# Java基础相关

## 基本数据类型



## **String， Stringbuffer， StringBuilder 的区别**

String 字符串常量(final修饰，不可被继承)，String是常量，当创建之后即不能更改。(可以通过StringBuffer和StringBuilder创建String对象(常用的两个字符串操作类)。)

StringBuffer 字符串变量（线程安全）,其也是final类别的，不允许被继承，其中的绝大多数方法都进行了同步处理，包括常用的Append方法也做了同步处理(synchronized修饰)。其自jdk1.0起就已经出现。其toString方法会进行对象缓存，以减少元素复制开销。 StringBuilder 字符串变量（非线程安全）其自jdk1.5起开始出现。与StringBuffer一样都继承和实现了同样的接口和类，方法除了没使用synch修饰以外基本一致，不同之处在于最后toString的时候，会直接返回一个新对象。

## ArrayList 和 LinkedList 有什么区别

ArrayList和LinkedList都实现了List接口，有以下的不同点：

1、ArrayList是基于索引的数据接口，它的底层是数组。它可以以O(1)时间复杂度对元素进行随机访问。与此对应，LinkedList是以元素列表的形式存储它的数据，每一个元素都和它的前一个和后一个元素链接在一起，在这种情况下，查找某个元素的时间复杂度是O(n)。

2、相对于ArrayList，LinkedList的插入，添加，删除操作速度更快，因为当元素被添加到集合任意位置的时候，不需要像数组那样重新计算大小或者是更新索引。

3、LinkedList比ArrayList更占内存，因为LinkedList为每一个节点存储了两个引用，一个指向前一个元素，一个指向下一个元素。

## **类的实例化顺序**

父类静态代变量、父类静态代码块、子类静态变量、子类静态代码块、父类非静态变量（父类实例成员变量）、父类构造函数、子类非静态变量（子类实例成员变量）、子类构造函数。

## 继承和聚合的区别在哪。

继承指的是一个类（称为子类、子接口）继承另外的一个类（称为父类、父接口）的功能，并可以增加它自己的新功能的能力，继承是类与类或者接口与接口之间最常见的关系；在Java中此类关系通过关键字extends明确标识，在设计时一般没有争议性；

聚合是关联关系的一种特例，他体现的是整体与部分、拥有的关系，即has-a的关系，此时整体与部分之间是可分离的，他们可以具有各自的生命周期，部分可以属于多个整体对象，也可以为多个整体对象共享；比如计算机与CPU、公司与员工的关系等；表现在代码层面，和关联关系是一致的，只能从语义级别来区分；

## Nio

内存与存储设备之间的传输数据的通道

### 按方向分：

- InputStream/Reader: 所有的输入流的基类，前者是字节输入流，后者是字符输入流。方法read

- OutputStream/Writer: 所有输出流的基类，前者是字节输出流，后者是字符输出流。方法write

### 按单位：字节流和字符流

File对象本身可以是目录。

调用file.mkdirs()即可创建目录。

如果是文件或者空目录，可以直接删除。

但如果目录中有文件或者子目录，则必须递归删除。

- 字节数组char[] 作为输入源的InputStream类是————ByteArrayInputStream

- 用文件作为输入源的InputStream类是？————FileInputStream

- 用字符串作为输入源的是？————StringBufferInputStream

- 用于多线程之间管道通信的输入源是————PipeInputStream

BufferedInputStream字节缓冲流，将字节流传入缓冲流，减少磁盘的的访问次数

BufferedReder字符缓冲流，高效读取，支持换行符，可以读取一行写一行

> 使用FileReader和FileWriter复制文本文件，不能复制图片和二进制文件

>

> 使用字节流能复制任意文件。

InputStreamReader/OutputStreamWriter，桥转换流

字节流和字符流的转化，硬盘上是字节流，内存中是字符

### 三种IO

IO(BIO)是面向流的，NIO是面向缓冲区的，BIO里用户最关心“我要读”，NIO里用户最关心"我可以读了"

BIO：Block IO 同步阻塞式 IO，就是我们平常使用的传统 IO，它的特点是模式简单使用方便，并发处理能力低。

NIO：New IO 同步非阻塞 IO，是传统 IO 的升级，客户端和服务器端通过 Channel（通道）通讯，实现了多路复用。

AIO：Asynchronous IO 是 NIO 的升级，也叫 NIO2，实现了异步非堵塞 IO ，异步 IO 的操作基于事件和回调机制。

### 选择器（Selectors）

Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

Selector与Channel是相互配合使用的，将Channel注册在Selector上之后，才可以正确的使用Selector，但此时Channel必须为非阻塞模式。Selector可以监听Channel的四种状态（Connect、Accept、Read、Write），当监听到某一Channel的某个状态时，才允许对Channel进行相应的操作。

- Connect：某一个客户端连接成功后

- Accept：准备好进行连接

- Read:可读

- Write:可写

### 数据处理

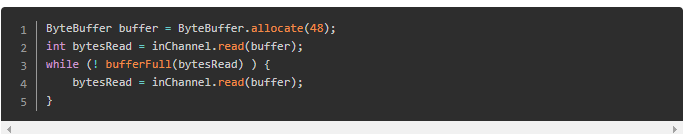
使用NIO的API调用时看起来与使用IO时有所不同，但这并不意外，因为并不是仅从一个InputStream逐字节读取，而是数据必须先读入缓冲区再处理。

（Java IO: 从一个阻塞的流中读数据） 而一个NIO的实现会有所不同，下面是一个简单的例子：



注意第二行，从通道读取字节到ByteBuffer。当这个方法调用返回时，你不知道你所需的所有数据是否在缓冲区内。你所知道的是，该缓冲区包含一些字节，这使得处理有点困难。假设第一次read(buffer)调用后，读入缓冲区的数据只有半行，例如，“Name:An”，你能处理数据吗？显然不能，需要等待，直到整行数据读入缓存，在此之前，对数据的任何处理毫无意义。

所以，你怎么知道是否该缓冲区包含足够的数据可以处理呢？好了，你不知道。发现的方法只能查看缓冲区中的数据。其结果是，在你知道所有数据都在缓冲区里之前，你必须检查几次缓冲区的数据。这不仅效率低下，而且可以使程序设计方案杂乱不堪。例如：



bufferFull()方法必须跟踪有多少数据读入缓冲区，并返回真或假，这取决于缓冲区是否已满。换句话说，如果缓冲区准备好被处理，那么表示缓冲区满了。

bufferFull()方法扫描缓冲区，但必须保持在bufferFull()方法被调用之前状态相同。如果没有，下一个读入缓冲区的数据可能无法读到正确的位置。这是不可能的，但却是需要注意的又一问题。

Java NIO:从一个通道里读数据，直到所有的数据都读到缓冲区里.

## 反射的原理，反射创建类实例的三种方式是什么

Java反射是Java被视为动态（或准动态）语言的一个关键性质。这个机制允许程序在运行时透过Reflection APIs取得任何一个已知名称的class的内部信息，包括其modifiers（诸如public, static 等）、superclass（例如Object）、实现之interfaces（例如Cloneable），也包括fields和methods的所有信息，并可于运行时改变fields内容或唤起methods。

Java反射机制容许程序在运行时加载、探知、使用编译期间完全未知的classes。

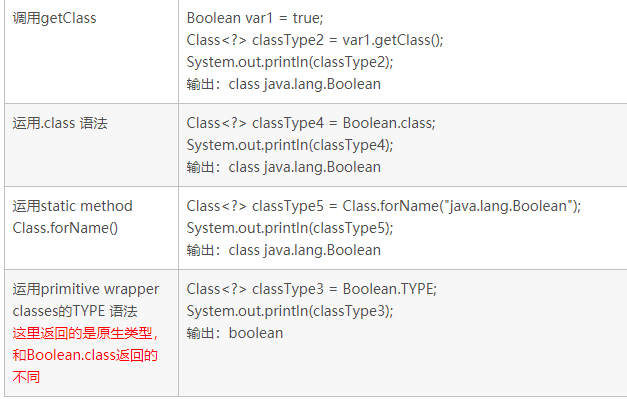
换言之，Java可以加载一个运行时才得知名称的class，获得其完整结构。

Class.forName(className)方法，其实调用的方法是Class.forName(className,true,classloader);注意看第2个boolean参数，它表示的意思，在loadClass后必须初始化。比较下我们前面准备jvm加载类的知识，我们可以清晰的看到在执行过此方法后，目标对象的 static块代码已经被执行，static参数也已经被初始化。

再看ClassLoader.loadClass(className)方法，其实他调用的方法是ClassLoader.loadClass(className,false);还是注意看第2个 boolean参数，该参数表示目标对象被装载后不进行链接，这就意味这不会去执行该类静态块中间的内容。因此2者的区别就显而易见了。

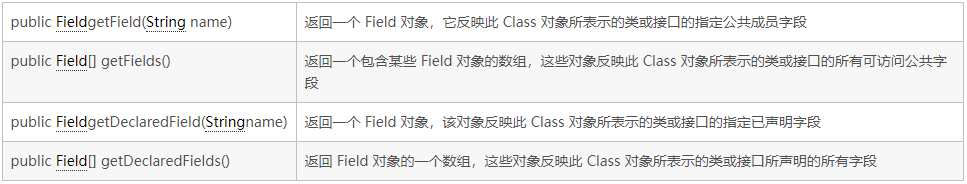
### 获取类的Class对象

Class 类的实例表示正在运行的 Java 应用程序中的类和接口。获取类的Class对象有多种方式：



### 获取类的Fields

可以通过反射机制得到某个类的某个属性，然后改变对应于这个类的某个实例的该属性值。JAVA 的Class<T>类提供了几个方法获取类的属性。



通过Java的反射机制，可以在运行期间调用对象的任何方法;

### 获取类的Method

通过反射机制得到某个类的某个方法，然后调用对应于这个类的某个实例的该方法

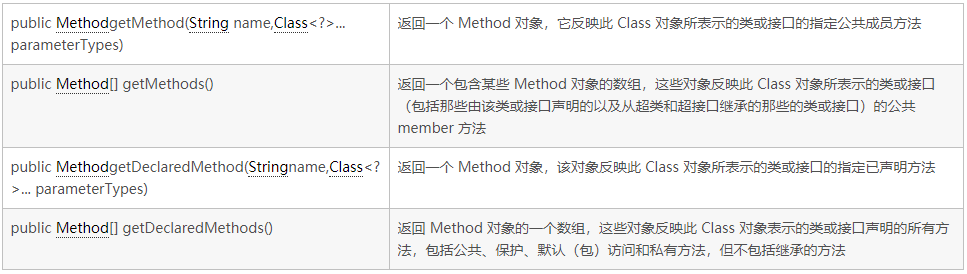
Class<T>类提供了几个方法获取类的方法。



### 获取类的Constructor

通过反射机制得到某个类的构造器，然后调用该构造器创建该类的一个实例

Class<T>类提供了几个方法获取类的构造器。



### 新建类的实例

通过反射机制创建新类的实例，有几种方法可以创建



### 调用类的函数

通过反射获取类Method对象，调用Field的Invoke方法调用函数。

### 设置/获取类的属性值

通过反射获取类的Field对象，调用Field方法设置或获取值

## 描述动态代理的几种实现方式，分别说出相应的优缺点

Jdk cglib jdk底层是利用反射机制，需要基于接口方式，这是由于



Cglib则是基于asm框架，实现了无反射机制进行代理，利用空间来换取了时间，代理效率高于jdk,

总的来说，反射机制在生成类的过程中比较高效，而asm在生成类之后的相关执行过程中比较高效（可以通过将asm生成的类进行缓存，这样解决asm生成类过程低效问题）。还有一点必须注意：jdk动态代理的应用前提，必须是目标类基于统一的接口。如果没有上述前提，jdk动态代理不能应用。由此可以看出，jdk动态代理有一定的局限性，cglib这种第三方类库实现的动态代理应用更加广泛，且在效率上更有优势。

Cglib是一个优秀的动态代理框架，它的底层使用ASM(ASM是一个通用的Java字节码操作和分析框架。 它可以用于修改现有类或直接以二进制形式动态生成类。)在内存中动态的生成被代理类的子类，使用CGLIB即使代理类没有实现任何接口也可以实现动态代理功能。CGLIB具有简单易用，它的运行速度要远远快于JDK的Proxy动态代理：

cglib有两种可选方式，继承和引用。第一种是基于继承实现的动态代理，所以可以直接通过super调用target方法，但是这种方式在spring中是不支持的，因为这样的话，这个target对象就不能被spring所管理，所以cglib还是才用类似jdk的方式，通过持有target对象来达到拦截方法的效果。

## final 的用途

类、变量、方法

final 修饰的类叫最终类，该类不能被继承。

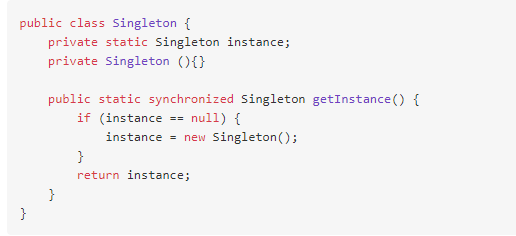
final 修饰的方法不能被重写。

final 修饰的变量叫常量，常量必须初始化，初始化之后值就不能被修改。

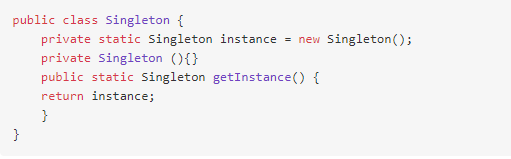
## 单例模式

保证一个类仅有一个实例，并提供一个访问它的全局访问点！

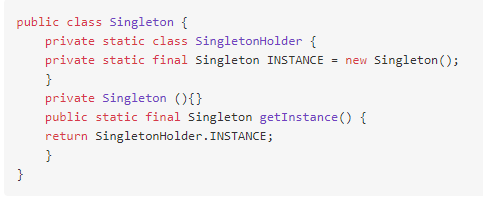
懒汉，线程不安全

这种写法能够在多线程中很好的工作，而且看起来它也具备很好的lazy loading，但是，遗憾的是，效率很低，99%情况下不需要同步。

饿汉

这种方式基于classloder机制避免了多线程的同步问题，instance在类装载时就实例化。目前java单例是指一个虚拟机的范围，因为装载类的功能是虚拟机的，所以一个虚拟机在通过自己的ClassLoader装载饿汉式实现单例类的时候就会创建一个类的实例。这就意味着一个虚拟机里面有很多ClassLoader，而这些classloader都能装载某个类的话，就算这个类是单例，也能产生很多实例。当然如果一台机器上有很多虚拟机，那么每个虚拟机中都有至少一个这个类的实例的话，那这样 就更不会是单例了。(这里讨论的单例不适合集群！)

静态内部类

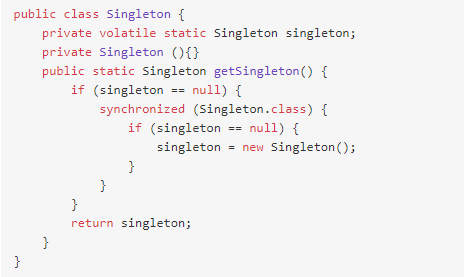
 这种方式同样利用了classloder的机制来保证初始化instance时只有一个线程，这种方式是Singleton类被装载了，instance不一定被初始化。因为SingletonHolder类没有被主动使用，只有显示通过调用getInstance方法时，才会显示装载SingletonHolder类，从而实例化instance。想象一下，如果实例化instance很消耗资源，我想让他延迟加载！这个时候，这种方式相比第2种方式就显得很合理。

4.枚举



这种方式是Effective Java作者Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象，可谓是很坚强的壁垒啊，不过，个人认为由于1.5中才加入enum特性，用这种方式写不免让人感觉生疏，在实际工作中，我也很少看见有人这么写过。

5.双重校验锁(jdk1.5)

这样方式实现线程安全地创建实例，而又不会对性能造成太大影响。它只是第一次创建实例的时候同步，以后就不需要同步了。

由于volatile关键字屏蔽了虚拟机中一些必要的代码优化，所以运行效率并不是很高，因此建议没有特别的需要不要使用。双重检验锁方式的单例不建议大量使用，根据情况决定。

## hashcode 和 equals 实现

hashcode

hashcode（）方法提供了对象的hashCode值，是一个native方法，返回的默认值与System.identityHashCode(obj)一致。

通常这个值是对象头部的一部分二进制位组成的数字，具有一定的标识对象的意义存在，但绝不定于地址。

作用是：用一个数字来标识对象。比如在HashMap、HashSet等类似的集合类中，如果用某个对象本身作为Key，即要基于这个对象实现Hash的写入和查找，那么对象本身如何实现这个呢？就是基于hashcode这样一个数字来完成的，只有数字才能完成计算和对比操作。

### hashcode是否唯一

hashcode只能说是标识对象，在hash算法中可以将对象相对离散开，这样就可以在查找数据的时候根据这个key快速缩小数据的范围，但hashcode不一定是唯一的，所以hash算法中定位到具体的链表后，需要循环链表，然后通过equals方法来对比Key是否是一样的。

### equals与hashcode的关系

equals相等两个对象，则hashcode一定要相等。但是hashcode相等的两个对象不一定equals相等。

### 覆盖equals时总要覆盖hashCode

一个很常见的错误根源在于没有覆盖hashCode方法。在每个覆盖了equals方法的类中，也必须覆盖hashCode方法。如果不这样做的话，就会违反Object.hashCode的通用约定，从而导致该类无法结合所有基于散列的集合一起正常运作，这样的集合包括HashMap、HashSet和Hashtable。

在应用程序的执行期间，只要对象的equals方法的比较操作所用到的信息没有被修改，那么对这同一个对象调用多次，hashCode方法都必须始终如一地返回同一个整数。在同一个应用程序的多次执行过程中，每次执行所返回的整数可以不一致。

如果两个对象根据equals()方法比较是相等的，那么调用这两个对象中任意一个对象的hashCode方法都必须产生同样的整数结果。

如果两个对象根据equals()方法比较是不相等的，那么调用这两个对象中任意一个对象的hashCode方法，则不一定要产生相同的整数结果。但是程序员应该知道，给不相等的对象产生截然不同的整数结果，有可能提高散列表的性能。

## 数组与链表的优缺点和区别

数组　是将元素在内存中**连续存放**，由于每个元素占用内存相同，可以通过下标迅速访问数组中任何元素。但是如果要在数组中增加一个元素，需要移动大量元素，在内存中空出一个元素的空间，然后将要增加的元素放在其中。同样的道理，如果想删除一个元素，同样需要移动大量元素去填掉被移动的元素。如果应用需要**快速访问数据，很少插入和删除元素，就应该用数组。**

链表　中的元素在内存中不是顺序存储的，而是通过存在元素中的指针联系到一起，每个结点包括两个部分：一个是存储 数据元素 的　数据域，另一个是存储下一个结点地址的 指针。   
　　如果要访问链表中一个元素，需要从第一个元素开始，一直找到需要的元素位置。但是增加和删除一个元素对于链表数据结构就非常简单了，只要修改元素中的指针就可以了。如果应用需要**经常插入和删除元素你就需要用链表**。

### 内存存储区别

数组从栈中分配空间, 对于程序员方便快速,但自由度小。

链表从堆中分配空间, 自由度大但申请管理比较麻烦.

### 总结

1、存取方式上，数组可以顺序存取或者随机存取，而链表只能顺序存取；

2、存储位置上，数组逻辑上相邻的元素在物理存储位置上也相邻，而链表不一定；

3、存储空间上，链表由于带有指针域，存储密度不如数组大；

4、按序号查找时，数组可以随机访问，时间复杂度为O(1)，而链表不支持随机访问，平均需要O(n)；

5、按值查找时，若数组无序，数组和链表时间复杂度均为O(1)，但是当数组有序时，可以采用折半查找将时间复杂度降为O(logn)；

6、插入和删除时，数组平均需要移动n/2个元素，而链表只需修改指针即可；

7、空间分配方面：

　　数组在静态存储分配情形下，存储元素数量受限制，动态存储分配情形下，虽然存储空间可以扩充，但需要移动大量元素，导致操作效率降低，而且如果内存中没有更大块连续存储空间将导致分配失败；

　　链表存储的节点空间只在需要的时候申请分配，只要内存中有空间就可以分配，操作比较灵活高效；

## 异常分类

java.lang.Throwable是所有异常的根

java.lang.Error是错误信息

java.lang.Exception是异常信息

### Checked异常

1 当前方法知道如何处理该异常，则用try...catch块来处理该异常。

2 当前方法不知道如何处理，则在定义该方法是声明抛出该异常。

我们比较熟悉的Checked异常有

InterruptedException

NoSuchFieldException

Java.lang.ClassNotFoundException

Java.lang.NoSuchMetodException

java.io.IOException

### RuntimeException

Runtime如除数是0和数组下标越界等，其产生频繁，处理麻烦，若显示申明或者捕获将会对程序的可读性和运行效率影响很大。所以由系统自动检测并将它们交给缺省的异常处理程序。当然如果你有处理要求也可以显示捕获它们。

Java.lang.ClassCastException

Java.lang.IndexOutOfBoundsException

Java.lang.NullPointerException

java.lang.NumberFormatException

ArithmeticException

IllegalArgumentException

### Error

当程序发生不可控的错误时，通常做法是通知用户并中止程序的执行。与异常不同的是Error及其子类的对象不应被抛出。

Error 类型的错误通常为虚拟机相关错误，如系统崩溃，内存不足，堆栈溢出等，编译器不会对这类错误进行检测，JAVA 应用程序也不应对这类错误进行捕获，一旦这类错误发生，通常应用程序会被终止，仅靠应用程序本身无法恢复；

Exception 类的错误是可以在应用程序中进行捕获并处理的，通常遇到这种错误，应对其进行处理，使应用程序可以继续正常运行。

Error是throwable的子类，代表编译时间和系统错误，用于指示合理的应用程序不应该试图捕获的严重问题。

Error由Java虚拟机生成并抛出，包括动态链接失败，虚拟机错误等。程序对其不做处理。

### JVM 是如何处理异常的？

在一个方法中如果发生异常，这个方法会创建一个异常对象，并转交给 JVM，该异常对象包含异常名称，异常描述以及异常发生时应用程序的状态。创建异常对象并转交给 JVM 的过程称为抛出异常。可能有一系列的方法调用，最终才进入抛出异常的方法，这一系列方法调用的有序列表叫做调用栈。

JVM 会顺着调用栈去查找看是否有可以处理异常的代码，如果有，则调用异常处理代码。当 JVM 发现可以处理异常的代码时，会把发生的异常传递给它。如果 JVM 没有找到可以处理该异常的代码块，JVM 就会将该异常转交给默认的异常处理器（默认处理器为 JVM 的一部分），默认异常处理器打印出异常信息并终止应用程序。

### Java异常处理最佳实践

1. 在 finally 块中清理资源或者使用 try-with-resource 语句

2. 优先明确的异常

3. 对异常进行文档说明

4. 使用描述性消息抛出异常

5. 优先捕获最具体的异常

6. 不要捕获 Throwable 类 JVM 抛出错误，指出不应该由应用程序处理的严重问题。

7. 不要忽略异常 合理的做法是至少要记录异常的信息。

8. 不要记录并抛出异常 这经常会给同一个异常输出多条日志

9. 包装异常时不要抛弃原始的异常 否则，你将会丢失堆栈跟踪和原始异常的消息，这将会使分析导致异常的异常事件变得困难。

10. 不要使用异常控制程序的流程 会严重影响应用的性能

11. 使用标准异常 内建的异常可以解决问题，就不要定义自己的异常

12. 异常会影响性能 成本非常高

## 泛型

泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数，泛型的好处是在编译的时候检查类型安全，并且所有的强制转换都是自动和隐式的，以提高代码的重用率

Java泛型是J2 SE1.5中引入的一个新特性，其本质是参数化类型，也就是说所操作的数据类型被指定为一个参数（type parameter）这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口、泛型方法。

Java语言中引入泛型是一个较大的功能增强。不仅语言、类型系统和编译器有了较大的变化，以支持泛型，而且类库也进行了很大的改动，许多重要的类，比如集合框架，都已经成为泛型化的了。这带来了很多好处：

1、类型安全

泛型的主要目标是提高Java程序的类型安全。通过知道使用泛型定义的变量的类型限制，编译器可以在非常高的层次上验证类型假设。没有泛型，这些假设就只存在于系统开发人员的头脑中。

通过在变量声明中捕获这一附加的类型信息，泛型允许编译器实施这些附加的类型约束。类型错误就可以在编译时被捕获了，而不是在运行时当作ClassCastException展示出来。将类型检查从运行时挪到编译时有助于Java开发人员更早、更容易地找到错误，并可提高程序的可靠性。

2、消除强制类型转换

泛型的一个附带好处是，消除源代码中的许多强制类型转换。这使得代码更加可读，并且减少了出错机会。尽管减少强制类型转换可以提高使用泛型类的代码的累赞程度，但是声明泛型变量时却会带来相应的累赞程度。在简单的程序中使用一次泛型变量不会降低代码累赞程度。但是对于多次使用泛型变量的大型程序来说，则可以累积起来降低累赞程度。所以泛型消除了强制类型转换之后，会使得代码加清晰和筒洁。

3、更高的运行效率

在非泛型编程中，将筒单类型作为Object传递时会引起Boxing（装箱）和Unboxing（拆箱）操作，这两个过程都是具有很大开销的。引入泛型后，就不必进行Boxing和Unboxing操作了，所以运行效率相对较高，特别在对集合操作非常频繁的系统中，这个特点带来的性能提升更加明显。

4、潜在的性能收益

泛型为较大的优化带来可能。在泛型的初始实现中，编译器将强制类型转换（没有泛型的话，Java系统开发人员会指定这些强制类型转换）插入生成的字节码中。但是更多类型信息可用于编译器这一事实，为未来版本的JVM的优化带来可能

## Lambda表达式

简化接口的实现，只能实现函数式接口，如果一个接口中要求实现类必须实现的抽象方法只有一个，这样的接口就是函数式接口

@FunctionalInterface，用来验证这是不是一个函数式接口，如果不是会报错。

重点关注的参数和返回值。

基础语法

`（参数）->{`

方法体

`}`

如果参数只有一个小括号可以省略

如果方法体只有一行大括号可以省略，return关键字也不用写

方法体可以直接是一个方法函数，函数引用时，参数列表和接口实现必须一样

静态方法的引用

​ 类名：：静态方法名

非静态方法的引用

​ 对象：：方法名

构造方法的引用

​ 类名：：new

特殊的方法引用，方法的参数列表和函数式接口一样

​ 类名：：非静态方法名

注意的问题：

局部变量默认为final

## 流式编程：

对集合操作的增强：

将数据源转换成流（Stream）进行处理，得到新的Stream

流式编程可以大幅度简化代码的数量，进行流式操作。（删除，过滤，映射）

### 流式编程步骤

1. 获取数据源；

​ 使用collection的steam方法，获取一个同步流

​ 使用collection的parallelStream方法，获取一个并发流，并发处理效率要高

​ 使用Arrays.stream获取流

2. 对流中的方法进行处理，所有的操作都是函数式接口，可以使用lambda表达式来实现。简化代码量

3. 对流中的数据进行整合最终处理

### 最终操作

关闭流，将流转成集合或者遍历，关闭后不能使用

1. collect 将数据收集到一起，参数一个接口（非函数式接口），可以使用Collectors中的方法，toList,toMap,toSet

2. reduce 将流中的数据按照一定的规则聚合起来。流中的元素两两操作。.get获取结果

3. count 获取流中的数据

4. forEach迭代、遍历流中的数据

5. mac&min获取最大或最小值，可以指定规则。

6. Matching 做匹配，allMatch，anyMatch，noneMatch 返回值为Boolean，判断流中是否有我们想要的元素

7. find 查找元素，findFirst串行流和并行流都是首元素 findAny并行流获取到的可能不是首元素

instream 方法summaryStatistics，对流进行分析

### 中间操作

- filter 条件过滤

- distinct 去重先比较hashcode，然后使用equles，需要重写两个方法

- sorted 排序根据对应的类实现的comparaable接口进行排序，需要实现Comparable接口，或者传入自定义比较规则

- limit&skip 从0位置开始截取流中指定数量的数据，skip指从0位置开始跳过指定数量的数据

- map对流中的数据进行映射，用新的数据替换旧的数据

- flatmap 扁平化映射，一般用于map映射完成后流中的数据是一个容器，需要对容器中的数据进行处理，扁平化处理，可以将集合中的容器数据直接读到流中

### Collectors工具类

Collectors.toList（），Collectors.toSet（），Collectors.toMap（）

maxBy（）找到流中最大的元素，等同于max

joining（）将字符串拼接，可以传入分隔符，前缀和尾缀，只适用于Stream<String>

summingInt 将流中的数据映射成intstream，并求和

averagingInt（），aummarizingInt

## Java 网络编程

网络编程是指编写运行在多个设备（计算机）的程序，这些设备都通过网络连接起来。

java.net 包中 J2SE 的 API 包含有类和接口，它们提供低层次的通信细节。你可以直接使用这些类和接口，来专注于解决问题，而不用关注通信细节。

java.net 包中提供了两种常见的网络协议的支持：

- \*\*TCP\*\*：TCP 是传输控制协议的缩写，它保障了两个应用程序之间的可靠通信。通常用于互联网协议，被称 TCP / IP。

- \*\*UDP\*\*：UDP 是用户数据报协议的缩写，一个无连接的协议。提供了应用程序之间要发送的数据的数据包。

### 三次握手与四次挥手

TCP是面向连接的协议，因此每个TCP连接都有3个阶段：连接建立、数据传送和连接释放。连接建立经历三个步骤，通常称为“三次握手”。

TCP三次握手过程如下：

1. 第一次握手

客户机发送连接请求报文段到服务器，并进入SYN\_SENT状态，等待服务器确认。（SYN = 1,seq=x）

2. 第二次握手

服务器收到连接请求报文，如果同意建立连接，向客户机发回确认报文段，并为该TCP连接分配TCP缓存和变量。(SYN=1,ACK=1,seq=y,ack=x+1)。

3. 第三次握手

客户机收到服务器的确认报文段后，向服务器给出确认报文段，并且也要给该连接分配缓存和变量。此包发送完毕，客户端和服务器进入ESTABLISHED（TCP连接成功）状态，完成三次握手。(ACK=1,seq=x+1,ack=y+1)。

TCP四次挥手过程如下：

由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

1. TCP客户端发送一个FIN，用来关闭客户到服务器的数据传送。

2. 服务器收到这个FIN，它发回一个ACK，确认序号为收到的序号加1。和SYN一样，一个FIN将占用一个序号。

3. 服务器关闭客户端的连接，发送一个FIN给客户端。

4. 客户端发回ACK报文确认，并将确认序号设置为收到序号加1。

### HTTP状态码

1xx：消息

2xx: 成功

3xx: 重定向

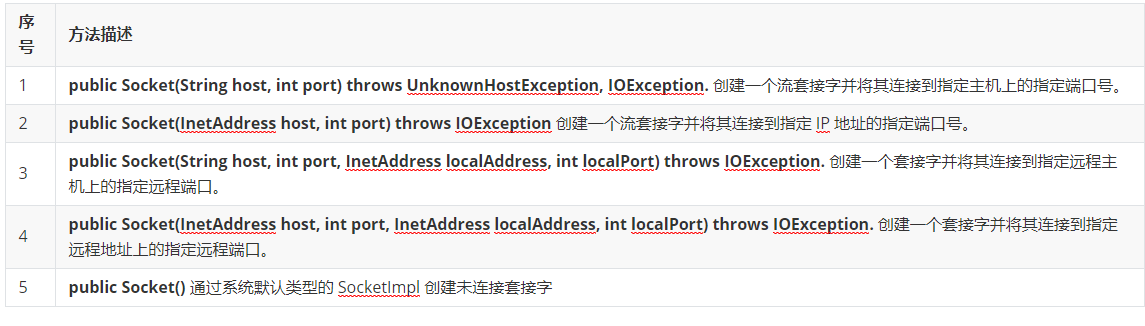
4xx: 请求出错 404 NotFound，403 Forbidden

5xx: 服务器出错 502Bad GateWay

### Socket 类的方法

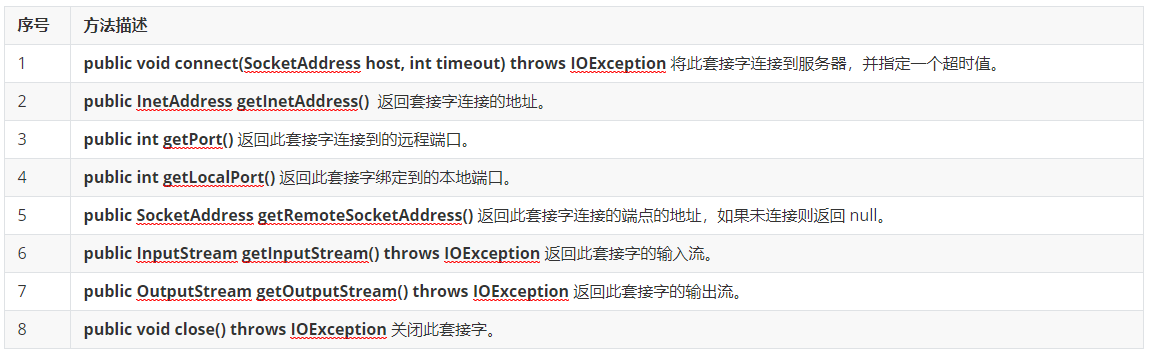
java.net.Socket 类代表客户端和服务器都用来互相沟通的套接字。客户端要获取一个 Socket 对象通过实例化 ，而 服务器获得一个 Socket 对象则通过 accept() 方法的返回值。

Socket 类有五个构造方法.



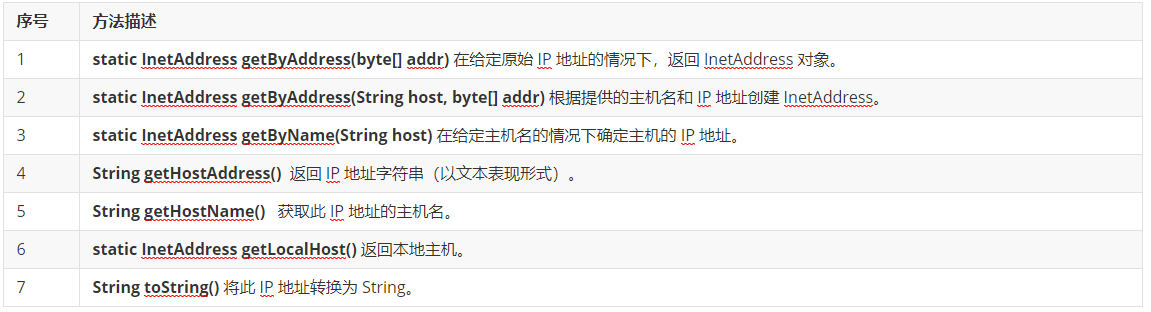
当 Socket 构造方法返回，并没有简单的实例化了一个 Socket 对象，它实际上会尝试连接到指定的服务器和端口。

下面列出了一些感兴趣的方法，注意客户端和服务器端都有一个 Socket 对象，所以无论客户端还是服务端都能够调用这些方法。



InetAddress 类的方法

这个类表示互联网协议(IP)地址。下面列出了 Socket 编程时比较有用的方法：



# Java多线程

java程序至少有三个线程，主线程，GC线程，异常处理线程，发生异常会影响主线程。

单核是一个假的多线程

并行是多个cpu同时执行多个任务。

并发是一个cpu采用时间片同时执行多个任务。

## 并发编程的优点

充分利用多核CPU的计算能力：通过并发编程的形式可以将多核CPU的计算能力发挥到极致，性能得到提升

方便进行业务拆分，提升系统并发能力和性能：在特殊的业务场景下，先天的就适合于并发编程。现在的系统动不动就要求百万级甚至千万级的并发量，而多线程并发编程正是开发高并发系统的基础，利用好多线程机制可以大大提高系统整体的并发能力以及性能。面对复杂业务模型，并行程序会比串行程序更适应业务需求，而并发编程更能吻合这种业务拆分 。

并发编程的目的就是为了能提高程序的执行效率，提高程序运行速度，但是并发编程并不总是能提高程序运行速度的，而且并发编程可能会遇到很多问题，比如\*\*：内存泄漏、上下文切换、线程安全、死锁\*\*等问题。

1. 上下文切换

多线程编程中一般线程的个数都大于 CPU 核心的个数，而一个 CPU 核心在任意时刻只能被一个线程使用，为了让这些线程都能得到有效执行，CPU 采取的策略是为每个线程分配时间片并轮转的形式。当一个线程的时间片用完的时候就会重新处于就绪状态让给其他线程使用，这个过程就属于一次上下文切换。

2. 死锁是指两个或两个以上的进程（线程）在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程（线程）称为死锁进程（线程）。

## 死锁的四个必要条件是什么

互斥条件：线程(进程)对于所分配到的资源具有排它性，即一个资源只能被一个线程(进程)占用，直到被该线程(进程)释放

请求与保持条件：一个线程(进程)因请求被占用资源而发生阻塞时，对已获得的资源保持不放。

不剥夺条件：线程(进程)已获得的资源在末使用完之前不能被其他线程强行剥夺，只有自己使用完毕后才释放资源。

循环等待条件：当发生死锁时，所等待的线程(进程)必定会形成一个环路（类似于死循环），造成永久阻塞

## JMM

JMM定义了Java 虚拟机(JVM)在计算机内存(RAM)中的工作方式。JVM是整个计算机虚拟模型，所以JMM是隶属于JVM的。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（Main Memory）中，每个线程都有一个私有的本地内存（Local Memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存、写缓冲区、寄存器以及其他的硬件和编译器优化。

JMM(Java内存模型Java Memory Model,简称JMM)本身是一种抽象的概念 并不真实存在,它描述的是一组规则或规范通过规范定制了程序中各个变量(包括实例字段,静态字段和构成数组对象的元素)的访问方式.

JMM关于同步规定:

1. 线程解锁前,必须把共享变量的值刷新回主内存

2. 线程加锁前,必须读取主内存的最新值到自己的工作内存

3. 加锁解锁是同一把锁

由于JVM运行程序的实体是线程,而每个线程创建时JVM都会为其创建一个工作内存(有些地方成为栈空间),工作内存是每个线程的私有数据区域,而Java内存模型中规定所有变量都存储在主内存,主内存是共享内存区域,所有线程都可访问,但线程对变量的操作(读取赋值等)必须在工作内存中进行,首先要将变量从主内存拷贝到自己的工作空间,然后对变量进行操作,操作完成再将变量写回主内存,不能直接操作主内存中的变量,各个线程中的工作内存储存着主内存中的变量副本拷贝,因此不同的线程无法访问对方的工作内存,此案成间的通讯(传值) 必须通过主内存来完成

## 并发编程三要素：

原子性：原子，即一个不可再被分割的颗粒。原子性指的是一个或多个操作要么全部执行成功要么全部执行失败。n++存在原子性问题，通过JUC里面的原子整型包装类可以解决。

可见性：一个线程对共享变量的修改,另一个线程能够立刻看到。（synchronized,volatile）

有序性：程序执行的顺序按照代码的先后顺序执行。（处理器可能会对指令进行重排序）处理器在进行重新排序是必须要考虑指令之间的数据依赖性

## 创建线程有四种方式：

- 继承 Thread 类；

- 实现 Runnable 接口；

- 实现 Callable 接口；

- 使用 Executors 工具类创建线程池

## 线程的状态和基本操作

新建(new)：新创建了一个线程对象。

可运行(runnable)：线程对象创建后，当调用线程对象的 start()方法，该线程处于就绪状态，等待被线程调度选中，获取cpu的使用权。

运行(running)：可运行状态(runnable)的线程获得了cpu时间片（timeslice），执行程序代码。注：就绪状态是进入到运行状态的唯一入口，也就是说，线程要想进入运行状态执行，首先必须处于就绪状态中；

阻塞(block)：处于运行状态中的线程由于某种原因，暂时放弃对 CPU的使用权，停止执行，此时进入阻塞状态，直到其进入到就绪状态，才 有机会再次被 CPU 调用以进入到运行状态。

阻塞的情况分三种：

(一). 等待阻塞：运行状态中的线程执行 wait()方法，JVM会把该线程放入等待队列(waitting queue)中，使本线程进入到等待阻塞状态；

(二). 同步阻塞：线程在获取 synchronized 同步锁失败(因为锁被其它线程所占用)，，则JVM会把该线程放入锁池(lock pool)中，线程会进入同步阻塞状态；

(三). 其他阻塞: 通过调用线程的 sleep()或 join()或发出了 I/O 请求时，线程会进入到阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O 处理完毕时，线程重新转入就绪状态。

死亡(dead)：线程run()、main()方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。

### Java 中用到的线程调度算法是什么？

有两种调度模型：分时调度模型和抢占式调度模型

Java虚拟机采用抢占式调度模型，是指优先让可运行池中优先级高的线程占用CPU，如果可运行池中的线程优先级相同，那么就随机选择一个线程，使其占用CPU。处于运行状态的线程会一直运行，直至它不得不放弃 CPU。

### sleep() 和 wait() 有什么区别？

两者都可以暂停线程的执行

类的不同：sleep() 是 Thread线程类的静态方法，wait() 是 Object类的方法。

是否释放锁：sleep() 不释放锁；wait() 释放锁。

用途不同：Wait 通常被用于线程间交互/通信，sleep 通常被用于暂停执行。

用法不同：wait() 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 notify() 或者 notifyAll() 方法。sleep() 方法执行完成后，线程会自动苏醒。或者可以使用wait(long timeout)超时后线程会自动苏醒。

wait() 方法应该在循环调用，因为当线程获取到 CPU 开始执行的时候，其他条件可能还没有满足，所以在处理前，循环检测条件是否满足会更好。

### wait(), notify()和 notifyAll()被定义在 Object 类里

Java中，任何对象都可以作为锁，并且 wait()，notify()等方法用于等待对象的锁或者唤醒线程，在 Java 的线程中并没有可供任何对象使用的锁，所以任意对象调用方法一定定义在Object类中。

### Thread 类的 sleep()和 yield ()方法

Thread 类的 sleep()和 yield()方法将在当前正在执行的线程上运行

（1） sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；

（2） 线程执行 sleep()方法后转入阻塞（blocked）状态，而执行 yield()方法后转入就绪（ready）状态；

（3）sleep()方法声明抛出 InterruptedException，而 yield()方法没有声明任何异常；

（4）sleep()方法比 yield()方法（跟操作系统 CPU 调度相关）具有更好的可移植性，通常不建议使用yield()方法来控制并发线程的执行。

## synchronized 和 volatile ？

volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取； synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

volatile仅能使用在变量级别；synchronized则可以使用在变量、方法、和类级别的。

volatile仅能实现变量的修改可见性，不能保证原子性；而synchronized则可以保证变量的修改可见性和原子性。

volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。

volatile标记的变量不会被编译器优化；synchronized标记的变量可以被编译器优化。

### synchronized 关键字

在 Java 中，synchronized 关键字是用来控制线程同步的，就是在多线程的环境下，控制 synchronized 代码段不被多个线程同时执行。synchronized 可以修饰类、方法、变量。

- synchronized关键字最主要的三种使用方式：

修饰实例方法: 作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁

修饰静态方法: 也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（ static 表明这是该类的一个静态资源，不管new了多少个对象，只有一份）。所以如果一个线程A调用一个实例对象的非静态 synchronized 方法，而线程B需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁。

修饰代码块: 指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

总结： synchronized 关键字加到 static 静态方法和 synchronized(class)代码块上都是是给 Class 类上锁。synchronized 关键字加到实例方法上是给对象实例上锁。尽量不要使用 synchronized(String a) 因为JVM中，字符串常量池具有缓存功能！

- synchronized、volatile、CAS 比较

（1）synchronized 是悲观锁，属于抢占式，会引起其他线程阻塞。

（2）volatile 提供多线程共享变量可见性和禁止指令重排序优化。

（3）CAS 是基于冲突检测的乐观锁（非阻塞）

- synchronized 和 Lock 有什么区别？

- 首先synchronized是Java内置关键字，在JVM层面，Lock是个Java类；

- synchronized 可以给类、方法、代码块加锁；而 lock 只能给代码块加锁。

- synchronized 不需要手动获取锁和释放锁，使用简单，发生异常会自动释放锁，不会造成死锁；而 lock 需要自己加锁和释放锁，如果使用不当没有 unLock()去释放锁就会造成死锁。

- 通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。

- lock可以中断，synchronized要么出异常要么正常结束

- synchronized是非公平锁，Lock两者都可以

- Lock可以精确唤醒需要唤醒的线程，synchronized要么唤醒一个要么唤醒全部

### volatile 关键字

- 提供多线程共享变量可见性

- 不保证原子性

- 禁止指令重排序优化。

对于可见性，Java 提供了 volatile 关键字来保证可见性和禁止指令重排。 volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。当一个共享变量被 volatile 修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

从实践角度而言，volatile 的一个重要作用就是和 CAS 结合，保证了原子性，详细的可以参见 java.util.concurrent.atomic 包下的类，比如 AtomicInteger。

volatile 常用于多线程环境下的单次操作(单次读或者单次写)。

- volatile 修饰符的有过什么实践？

单例模式

是否 Lazy 初始化：是

是否多线程安全：是

实现难度：较复杂

描述：对于Double-Check这种可能出现的问题（当然这种概率已经非常小了，但毕竟还是有的嘛~），解决方案是：只需要给instance的声明加上volatile关键字即可volatile关键字的一个作用是禁止指令重排，把instance声明为volatile之后，对它的写操作就会有一个内存屏障（什么是内存屏障？），这样，在它的赋值完成之前，就不用会调用读操作。注意：volatile阻止的不是singleton = newSingleton()这句话内部[1-2-3]的指令重排，而是保证了在一个写操作（[1-2-3]）完成之前，不会调用读操作（if (instance == null)）。

uniqueInstance 采用 volatile 关键字修饰也是很有必要的， uniqueInstance = new Singleton(); 这段代码其实是分为三步执行：

为 uniqueInstance 分配内存空间

初始化 uniqueInstance

将 uniqueInstance 指向分配的内存地址

## Lock体系

### 公平锁和非公平锁,

公平锁：多个线程按照申请锁的顺序来获取锁，先进先出原则

非公平锁：多个线程的顺序不按照申请顺序来获得，可能造成优先级反转或者饥饿现象，没有按照顺序执行就是优先级反转。排队的线程一直拿不到锁而保持等待就是饥饿，优点是吞吐量比公平锁大

ReentrantLock默认为非公平锁，synchronized也是一种非公平锁。非公平锁会直接占有锁如果尝试失败，就采用类似公平锁的机制去等待锁。

### 乐观锁和悲观锁

悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。再比如 Java 里面的同步原语 synchronized 关键字的实现也是悲观锁。

乐观锁：顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 write\_condition 机制，其实都是提供的乐观锁。在 Java中 java.util.concurrent.atomic 包下面的原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。

乐观锁的实现方式：

1、使用版本标识来确定读到的数据与提交时的数据是否一致。提交后修改版本标识，不一致时可以采取丢弃和再次尝试的策略。

2、java 中的 Compare and Swap 即 CAS ，当多个线程尝试使用 CAS 同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。 CAS 操作中包含三个操作数 —— 需要读写的内存位置（V）、进行比较的预期原值（A）和拟写入的新值(B)。如果内存位置 V 的值与预期原值 A 相匹配，那么处理器会自动将该位置值更新为新值 B。否则处理器不做任何操作。

### 自旋锁

很多 synchronized 里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。既然 synchronized 里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在 synchronized 的边界做忙循环，这就是自旋。

尝试获取锁的线程不会立即阻塞，而是采用循环的方式去获取锁，

- 好处是不用阻塞减少上下文切换，

- 缺点是循环会消耗CPU，锁被长期占有的情况下不能使用自旋锁，消耗性能

可以使用原子引用类的CAS实现自旋锁，将当前线程放到原子引用中去，循环比较期望值为null就成功获取锁，解锁的时候如果期望值是当前线程就解锁。

### 死锁，活锁，饥饿

当线程 A 持有独占锁a，并尝试去获取独占锁 b 的同时，线程 B 持有独占锁 b，并尝试获取独占锁 a 的情况下，就会发生 AB 两个线程由于互相持有对方需要的锁，而发生的阻塞现象，我们称为死锁。

死锁与活锁的区别，死锁与饥饿的区别？

死锁：是指两个或两个以上的进程（或线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。

活锁：任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。

活锁和死锁的区别在于，处于活锁的实体是在不断的改变状态，这就是所谓的“活”， 而处于死锁的实体表现为等待；活锁有可能自行解开，死锁则不能。

饥饿：一个或者多个线程因为种种原因无法获得所需要的资源，导致一直无法执行的状态。

Java 中导致饥饿的原因：

1、高优先级线程吞噬所有的低优先级线程的 CPU 时间。

2、线程被永久堵塞在一个等待进入同步块的状态，因为其他线程总是能在它之前持续地对该同步块进行访问。

3、线程在等待一个本身也处于永久等待完成的对象(比如调用这个对象的 wait 方法)，因为其他线程总是被持续地获得唤醒。

### 独占锁和共享锁

独占锁：一个锁只能被一个线程所有ReentrantLock和synchronized都是独占锁

共享锁： 该锁能被多个线程持有，ReentrantReadWriteLock，读是共享，写是独占。

### 可重入锁/递归锁（ReentrantLock）

ReentrantLock重入锁，是实现Lock接口的一个类，支持重入性，

如果线程已经拥有锁，那么它可以进入这个锁所同步着的代码块

表示能够对共享资源能够重复加锁，即当前线程获取该锁再次获取不会被阻塞。

Synchronized也是一个典型的可重入锁

重入性的实现原理

1. 在线程获取锁的时候，如果已经获取锁的线程是当前线程的话则直接再次获取成功；

2. 由于锁会被获取n次，那么只有锁在被释放同样的n次之后，该锁才算是完全释放成功。

ReentrantLock支持两种锁：公平锁和非公平锁。何谓公平性，是针对获取锁而言的，如果一个锁是公平的，那么锁的获取顺序就应该符合请求上的绝对时间顺序，满足FIFO。

作用是避免死锁

锁和解锁要同时出现，否则会一直等待，哪怕加两层锁也可以正常进行代码

## CAS

CAS 是 compare and swap 的缩写，即我们所说的比较交换。它是一条CPU并发原语.原语属于操作系统用于范畴,是由若干条指令组成,用于完成某个功能的一个过程,并且原语的执行必须是连续的,在执行过程中不允许中断,也即是说CAS是一条原子指令,不会造成所谓的数据不一致的问题.

cas 是一种基于锁的操作，而且是乐观锁。在 java 中锁分为乐观锁和悲观锁。悲观锁是将资源锁住，等一个之前获得锁的线程释放锁之后，下一个线程才可以访问。而乐观锁采取了一种宽泛的态度，通过某种方式不加锁来处理资源，比如通过给记录加 version 来获取数据，性能较悲观锁有很大的提高。

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存地址里面的值和 A 的值是一样的，那么就将内存里面的值更新成 B。CAS是通过无限循环来获取数据的，若果在第一轮循环中，a 线程获取地址里面的值被b 线程修改了，那么 a 线程需要自旋，到下次循环才有可能机会执行。

底层实现是UnSafe类

是CAS的核心类 由于Java 方法无法直接访问底层 ,需要通过本地(native)方法来访问,UnSafe相当于一个后面,基于该类可以直接操作特额定的内存数据.UnSafe类在于sun.misc包中,其内部方法操作可以向C的指针一样直接操作内存,因为Java中CAS操作的助兴依赖于UNSafe类的方法.

注意UnSafe类中所有的方法都是native修饰的,也就是说UnSafe类中的方法都是直接调用操作底层资源执行响应的任务

java.util.concurrent.atomic 包的类大多使用 CAS 操作来实现(AtomicInteger,AtomicBoolean,AtomicLong)。

CAS 的会产生什么问题？

1、ABA 问题：

比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但可能存在潜藏的问题。从 Java1.5 开始 JDK 的 atomic包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。增加版本

2、循环时间长开销大：

对于资源竞争严重（线程冲突严重）的情况，CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 synchronized。

3、只能保证一个共享变量的原子操作：

当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁。

## AQS 原理概览

指的是AbstractQueuedSynchronizer，它提供了一种实现阻塞锁和一系列依赖FIFO等待队列的同步器的框架，ReentrantLock、Semaphore、CountDownLatch、CyclicBarrier等并发类均是基于AQS来实现的，具体用法是通过继承AQS实现其模板方法，然后将子类作为同步组件的内部类。

AQS(AbstractQueuedSynchronizer)核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

AQS使用一个int成员变量来表示同步状态，通过内置的FIFO队列来完成获取资源线程的排队工作。AQS使用CAS对该同步状态进行原子操作实现对其值的修改。

AQS 对资源的共享方式

AQS定义两种资源共享方式

Exclusive（独占）：只有一个线程能执行，如ReentrantLock。又可分为公平锁和非公平锁：

公平锁：按照线程在队列中的排队顺序，先到者先拿到锁

非公平锁：当线程要获取锁时，无视队列顺序直接去抢锁，谁抢到就是谁的

Share（共享）：多个线程可同时执行，如Semaphore/CountDownLatch。Semaphore、CountDownLatch、 CyclicBarrier、ReadWriteLock 我们都会在后面讲到。

### CycliBarriar 和 CountdownLatch

\*\*CountDownLatch(倒计时器)：\*\* CountDownLatch是一个同步工具类，用来协调多个线程之间的同步。这个工具通常用来控制线程阻塞等待，它可以让某一个线程等待直到倒计时结束，再开始执行。

两个方法：

1. 计数方法：countDownLath.countDown();

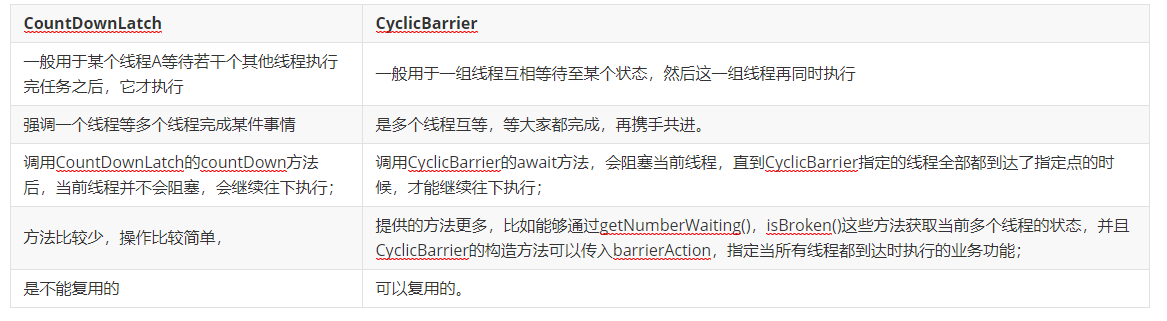
2. 等待方法：await（）；

CyclicBarrier(循环栅栏)： CyclicBarrier 和 CountDownLatch 非常类似，它也可以实现线程间的技术等待，但是它的功能比 CountDownLatch 更加复杂和强大。

主要应用场景和 CountDownLatch 类似。 CyclicBarrier 的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。

CyclicBarrier默认的构造方法是 CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用await()方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。

CountDownLatch与CyclicBarrier都是用于控制并发的工具类，都可以理解成维护的就是一个计数器，但是这两者还是各有不同侧重点的：



### Semaphore （信号量）

Semaphore 就是一个信号量，它的作用是限制某段代码块的并发数。多个共享资源的互斥使用，类似于停车位

Semaphore有一个构造函数，可以传入一个 int 型整数 n，表示某段代码最多只有 n 个线程可以访问，如果超出了 n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果 Semaphore 构造函数中传入的 int 型整数 n=1，相当于变成了一个 synchronized 了。

两个方法：1、acquire（）获取锁 2、release（）释放锁

Semaphore(信号量)-允许多个线程同时访问： synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，Semaphore(信号量)可以指定多个线程同时访问某个资源。

## 阻塞队列

当阻塞队列为空时，从队列里获取元素的操作会被阻塞

当阻塞队列满时，向队列里添加元素的操作会被阻塞

好处是我们不需要关心什么时候需要阻塞线程,什么时候需要唤醒线程,因为BlockingQueue都一手给你包办好了



- 抛出异常 当阻塞队列满时,再往队列里面add插入元素会抛IllegalStateException: Queue full

当阻塞队列空时,再往队列Remove元素时候回抛出NoSuchElementException

- 特殊值 插入方法,成功返回true 失败返回false

移除方法,成功返回元素,队列里面没有就返回null

- 一直阻塞 当阻塞队列满时,生产者继续往队列里面put元素,队列会一直阻塞直到put数据or响应中断退出

当阻塞队列空时,消费者试图从队列take元素,队列会一直阻塞消费者线程直到队列可用.

- 超时退出 当阻塞队列满时,队列会阻塞生产者线程一定时间,超过后限时后生产者线程就会退出

### ## 常见阻塞队列：

ArrayBlockingQueue: 由数组结构组成的有界阻塞队列.

LinkedBlockingDeque: 由链表结构组成的有界(但大小默认值Integer>MAX\_VALUE)阻塞队列.

PriorityBlockingQueue:支持优先级排序的无界阻塞队列.

DelayQueue: 使用优先级队列实现的延迟无界阻塞队列.

SynchronousQueue:不存储元素的阻塞队列,也即是单个元素的队列.

LinkedTransferQueue:由链表结构组成的无界阻塞队列.

LinkedBlockingDeque:由了解结构组成的双向阻塞队列.

## 并发容器

### List

ArrayList使用多线程会产生并发修改异常，解决方法：

1. 使用Vector已被废弃

2. 使用Collections工具类使list同步

3. 使用写时复制List（CopyonWriteArrayList）写的时候复制，读写分离，在写的时候加锁（ReentrantLock）加锁，拷贝数组，插入数值，写回数组，成功解锁

### Set

HashSet不安全，底层数据结构是HashMap，只关心key，不关系value，Value的是常量对象PRESENT = new Object()

1、使用Collecions同步集合

2、使用CopyOnWriteArraySet，底层实现用的是读写复制List

### Map

HashMap不安全使用Collections.synchronizedMap使其同步。

ConcurrentHashMap

ConcurrentHashMap是Java中的一个线程安全且高效的HashMap实现。平时涉及高并发如果要用map结构，那第一时间想到的就是它。相对于hashmap来说，ConcurrentHashMap就是线程安全的map，其中利用了锁分段的思想提高了并发度。

JDK1.8后，ConcurrentHashMap抛弃了原有的\*\*Segment 分段锁，而采用了 CAS + synchronized 来保证并发安全性\*\*。

### ConcurrentHashMap 的并发度是什么？

ConcurrentHashMap 把实际 map 划分成若干部分来实现它的可扩展性和线程安全。这种划分是使用并发度获得的，它是 ConcurrentHashMap 类构造函数的一个可选参数，默认值为 16，这样在多线程情况下就能避免争用。

在 JDK8 后，它摒弃了 Segment（锁段）的概念，而是启用了一种全新的方式实现,利用 CAS 算法。

### 并发容器的实现？

何为同步容器：可以简单地理解为通过 synchronized 来实现同步的容器，如果有多个线程调用同步容器的方法，它们将会串行执行

并发容器使用了与同步容器完全不同的加锁策略来提供更高的并发性和伸缩性

分段锁，允许任意数量的读线程并发地访问 map，并且执行读操作的线程和写操作的线程也可以并发的访问 map，同时允许一定数量的写操作线程并发地修改 map，所以它可以在并发环境下实现更高的吞吐量。

### SynchronizedMap 和 ConcurrentHashMap 有什么区别？

SynchronizedMap 一次锁住整张表来保证线程安全，所以每次只能有一个线程来访为 map。

ConcurrentHashMap 使用分段锁来保证在多线程下的性能。

## 线程池

### Executors类创建四种常见线程池

1）newSingleThreadExecutor：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

（2）newFixedThreadPool：创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。如果希望在服务器上使用线程池，建议使用 newFixedThreadPool方法来创建线程池，这样能获得更好的性能。

（3） newCachedThreadPool：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60 秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说 JVM）能够创建的最大线程大小。

（4）newScheduledThreadPool：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

### 线程池有什么优点？

降低资源消耗：重用存在的线程，减少对象创建销毁的开销。

提高响应速度。可有效的控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。当任务到达时，任务可以不需要的等到线程创建就能立即执行。

提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

附加功能：提供定时执行、定期执行、单线程、并发数控制等功能。

综上所述使用线程池框架 Executor 能更好的管理线程、提供系统资源使用率。

### 线程池都有哪些状态？

RUNNING：这是最正常的状态，接受新的任务，处理等待队列中的任务。

SHUTDOWN：不接受新的任务提交，但是会继续处理等待队列中的任务。

STOP：不接受新的任务提交，不再处理等待队列中的任务，中断正在执行任务的线程。

TIDYING：所有的任务都销毁了，workCount 为 0，线程池的状态在转换为 TIDYING 状态时，会执行钩子方法 terminated()。

TERMINATED：terminated()方法结束后，线程池的状态就会变成这个。

### ThreadPoolExecutor

《阿里巴巴Java开发手册》中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险

Executors 各个方法的弊端：

newFixedThreadPool 和 newSingleThreadExecutor:

主要问题是堆积的请求处理队列可能会耗费非常大的内存，甚至 OOM。

newCachedThreadPool 和 newScheduledThreadPool:

主要问题是线程数最大数是 Integer.MAX\_VALUE，可能会创建数量非常多的线程，甚至 OOM。

ThreaPoolExecutor创建线程池方式只有一种，就是走它的构造函数，参数自己指定

### ThreadPoolExecutor构造函数重要参数分析

ThreadPoolExecutor 3 个最重要的参数：

corePoolSize ：核心线程数，线程数定义了最小可以同时运行的线程数量。当值窗口

maximumPoolSize ：线程池中允许存在的工作线程的最大数量，核心线程和阻塞队列满了后扩容执行，候客区

workQueue：当新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，任务就会被存放在队列中。

ThreadPoolExecutor其他常见参数:

keepAliveTime：线程池中的线程数量大于 corePoolSize 的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了 keepAliveTime才会被回收销毁；

unit ：keepAliveTime 参数的时间单位。

threadFactory：为线程池提供创建新线程的线程工厂

handler ：线程池任务队列超过 maxinumPoolSize 之后的拒绝策略

### ThreadPoolExecutor饱和策略

ThreadPoolExecutor 饱和策略定义:

如果当前同时运行的线程数量达到最大线程数量并且队列也已经被放满了任时，ThreadPoolTaskExecutor 定义一些策略:

ThreadPoolExecutor.AbortPolicy：抛出 RejectedExecutionException来拒绝新任务的处理。

ThreadPoolExecutor.CallerRunsPolicy：调用执行自己的线程运行任务。您不会任务请求。但是这种策略会降低对于新任务提交速度，影响程序的整体性能。另外，这个策略喜欢增加队列容量。如果您的应用程序可以承受此延迟并且你不能任务丢弃任何一个任务请求的话，你可以选择这个策略。

ThreadPoolExecutor.DiscardPolicy：不处理新任务，直接丢弃掉。

ThreadPoolExecutor.DiscardOldestPolicy： 此策略将丢弃最早的未处理的任务请求。

### 合理配置线程池

cpu密集型：任务需要大量运算没有阻塞，一般是CPU核数+1个线程的此案成池

IO密集型：不是一直执行任务，cpu核数\*2，CPU核数/（1-阻塞系数）

### execute和submit的区别

1、有没有返回值

2、 execute只能接受Runnable类型的任务

submit不管是Runnable还是Callable类型的任务都可以接受，但是Runnable返回值均为void，所以使用Future的get()获得的还是null

3、异常处理

execute中抛出异常

execute中的是Runnable接口的实现，所以只能使用try、catch来捕获CheckedException，通过实现UncaughtExceptionHande接口处理UncheckedException

即和普通线程的处理方式完全一致

​ submit中抛出异常

​ 不管提交的是Runnable还是Callable类型的任务，如果不对返回值Future调用get()方法，都会吃掉异常

## 原子操作类

### 什么是原子操作？在 Java Concurrency API 中有哪些原子类？

原子操作（atomic operation）意为”不可被中断的一个或一系列操作” 。

处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。在 Java 中可以通过锁和循环 CAS 的方式来实现原子操作。 CAS 操作——Compare & Set，或是 Compare & Swap，现在几乎所有的 CPU 指令都支持 CAS 的原子操作。

原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。

java.util.concurrent 这个包里面提供了一组原子类。其基本的特性就是在多线程环境下，当有多个线程同时执行这些类的实例包含的方法时，具有排他性，即当某个线程进入方法，执行其中的指令时，不会被其他线程打断，而别的线程就像自旋锁一样，一直等到该方法执行完成，才由 JVM 从等待队列中选择另一个线程进入，这只是一种逻辑上的理解。

原子类：AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference

原子数组：AtomicIntegerArray，AtomicLongArray，AtomicReferenceArray

原子属性更新器：AtomicLongFieldUpdater，AtomicIntegerFieldUpdater，AtomicReferenceFieldUpdater

解决 ABA 问题的原子类：AtomicMarkableReference（通过引入一个 boolean来反映中间有没有变过），AtomicStampedReference（通过引入一个 int 来累加来反映中间有没有变过）

### 说一下 atomic 的原理？

Atomic包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。

# Java虚拟机

## JDK 和 JRE 有什么区别？

JDK：Java Development Kit 的简称，java 开发工具包，提供了 java 的开发环境和运行环境。

JRE：Java Runtime Environment 的简称，java 运行环境，为 java 的运行提供了所需环境。

具体来说 JDK 其实包含了 JRE，同时还包含了编译 java 源码的编译器 javac，还包含了很多 java 程序调试和分析的工具。简单来说：如果你需要运行 java 程序，只需安装 JRE 就可以了，如果你需要编写 java 程序，需要安装 JDK。

## jvm 和 类的关系

当调用 java命令运行一个java程序时，必会启动一个jvm即java虚拟机。该java程序的所有线程，变量都处于jvm中，都使用该jvm的内存区

jvm终止的情况：

1.程序自然运行结束

2.遇到System.exit();Runtime.getRuntime.exit();

3.遇到未捕获异常或错误时

4.程序所在的平台强制结束了JVM进程

jvm终止，jvm内存中的数据全部丢失。

### 类的加载

Java中的所有类，都需要由类加载器装载到JVM中才能运行。类加载器本身也是一个类，而它的工作就是把class文件从硬盘读取到内存中。在写程序的时候，我们几乎不需要关心类的加载，因为这些都是隐式装载的，除非我们有特殊的用法，像是反射，就需要显式的加载所需要的类。

类装载方式，有两种 ：

1.隐式装载， 程序在运行过程中当碰到通过new 等方式生成对象时，隐式调用类装载器加载对应的类到jvm中，

2.显式装载， 通过class.forname()等方法，显式加载需要的类

Java类的加载是动态的，它并不会一次性将所有类全部加载后再运行，而是保证程序运行的基础类(像是基类)完全加载到jvm中，至于其他类，则在需要的时候才加载。这当然就是为了节省内存开销。

类的加载 又称为 类的初始化，实际上可细分为 类的 加载、连接、初始化。下面将讲述着三个阶段的过程!

类的加载 指.class文件读入内存，并为之创建一个 java.lang.Class对象

类加载，是通过类加载器来完成的，类加载器通常由JVM提供，通常称为系统类加载器（也可以是自己写的加载器，只要继承ClassLoader基类）。

类加载无须等到“首次使用该类”时加载，jvm允许预加载某些类。。。。

### 类的连接

负责把类的二进制数据合并到JRE（java运行环境）中

1.验证 检测被加载的类是否有正确的内部结构，并和其他类协调一致

2.准备 负责为类的类变量（非对象变量）分配内存，并设置默认初始值

3.解析 将类的二进制数据中的符号引用替换成直接引用。。(static final 好像跟这个有点关系？？？？5.6.处有联系）

### 类初始化

主要对类变量（而非对象变量）的初始化

声明类变量的初始值 = 静态初始化块 他们是相同的，等效的。都会被当成类的初始化语句，JVM会按照这些语句在程序中的顺序依次执行他们

### 类初始化的时机

1.创建类的实例。new,反射，反序列化

2.使用某类的类方法--静态方法

3.访问某类的类变量，或赋值类变量

4.反射创建某类或接口的Class对象。Class.forName("Hello");---注意：loadClass调用ClassLoader.loadClass(name,false)方法,没有link，自然没有initialize

5.初始化某类的子类

6.直接使用java.exe来运行某个主类。即cmd java 程序会先初始化该类。

### 类加载器

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器:

- 启动类加载器(Bootstrap ClassLoader)用来加载java核心类库，无法被java程序直接引用。是虚拟机自身的一部分，用来加载Java\_HOME/lib/目录中的，或者被 -Xbootclasspath 参数所指定的路径中并且被虚拟机识别的类库；

- 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。

- 系统类加载器（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。

- 用户自定义类加载器，通过继承 java.lang.ClassLoader类的方式实现。

类加载器 负责加载所有的类，为被加载如内存中的类生成一个java.lang.Class实例。一旦类被载入内存，同一个类就不会再加载第二次

如何判断是同一个类：

java中 一个类用其 全限定类名标示--包名+类名

jvm中 一个类用其 全限定类名+加载器标示---包名+类名+加载器名

加载器层次结构：

JVM启动时，姓曾的三个类加载器组成的机构

1.Bootstrap ClassLoader 根类 ------引导类加载器，加载java核心类。非java.lang.ClassLoader子类，而是JVM自身实现

2.Extension ClassLoader 扩展类-----加载JRE的扩展目录中的JAR包的类（%JAVA\_HOME%/jre/lib/ext或java.ext.dirs系统属性指定的目录）

3.System ClassLoader 系统类-----加载cmd java -cp \*\*,环境变量指定的jar包和类路径。ClassLoader.getSystemClassLoader获得 系统类加载器。

### 类的加载机制：

1.全盘负责。某类以及其所依赖的所有类，都由一个加载器负责加载。除非显示使用另外一个加载器。

2.父类委托。双亲委派模型：先父类加载器加载改Class，不行后，才尝试从自己的类路径中加载该类

3.缓存机制。缓存机制将会保证所有加载过的Class都会被缓存。。当程序需要Class时，先从缓存区中寻找Class对象，没有的话，才加载该类的.class对象。

### 加载Class9个步骤：

1.检测此Class 是否被载入过（即在缓存区中是否由此 Class），有，则进入第8步，否则执行第2步。

2.如果父类加载器不存在（要么parent 一定是根类加载器，要么本身就是根类加载器），则跳到第4步；如果父类加载器存在，则执行第3步。

3.请求使用父类加载器去载入目标类，如果成功则跳到第8步，否则执行第5步

4.请求使用 根类加载器 载入目标类，成功则跳到第8步，否则跳到第7步

5.当前类加载器 尝试寻找 Class文件（从与此ClassLoader相关的类路径中寻找），如果找到则执行第6步，否则跳到第7步。

6.从文件中载入Class，成功后跳到第8步。

7.抛出ClassNotFoundException异常。

8.返回对应的 java.lang.Class对象。

### 类装载分为以下 5 个步骤：

- 加载：根据查找路径找到相应的 class 文件然后导入；

- 验证：检查加载的 class 文件的正确性；

- 准备：给类中的静态变量分配内存空间；

- 解析：虚拟机将常量池中的符号引用替换成直接引用的过程。符号引用就理解为一个标示，而在直接引用直接指向内存中的地址；

- 初始化：对静态变量和静态代码块执行初始化工作。

## 对象的创建

1. 虚拟机遇到一条new指令时，先检查常量池是否已经加载相应的类，如果没有，必须先执行相应的类加载。

2. 类加载通过后，接下来分配内存。

3. 划分内存时还需要考虑一个问题-并发，也有两种方式: CAS同步处理，或者本地线程分配缓冲(Thread Local Allocation Buffer, TLAB)。

4. 然后内存空间初始化操作，接着是做一些必要的对象设置(元信息、哈希码…)，最后执行<init>方法。

### 为对象分配内存

类加载完成后，接着会在Java堆中划分一块内存分配给对象。内存分配根据Java堆是否规整，有两种方式：

- 指针碰撞：如果Java堆的内存是规整，即所有用过的内存放在一边，而空闲的的放在另一边。分配内存时将位于中间的指针指示器向空闲的内存移动一段与对象大小相等的距离，这样便完成分配内存工作。

- 空闲列表：如果Java堆的内存不是规整的，则需要由虚拟机维护一个列表来记录那些内存是可用的，这样在分配的时候可以从列表中查询到足够大的内存分配给对象，并在分配后更新列表记录。

选择哪种分配方式是由 Java 堆是否规整来决定的，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

### 处理并发安全问题

对象的创建在虚拟机中是一个非常频繁的行为，哪怕只是修改一个指针所指向的位置，在并发情况下也是不安全的，可能出现正在给对象 A 分配内存，指针还没来得及修改，对象 B 又同时使用了原来的指针来分配内存的情况。解决这个问题有两种方案：

- 对分配内存空间的动作进行同步处理（采用 CAS + 失败重试来保障更新操作的原子性）；

- 本地线程分配缓冲（Thread Local Allocation Buffer, TLAB）:把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在 Java 堆中预先分配一小块内存，。哪个线程要分配内存，就在哪个线程的 TLAB 上分配。只有 TLAB 用完并分配新的 TLAB 时，才需要同步锁。通过-XX:+/-UserTLAB参数来设定虚拟机是否使用TLAB。

### 对象的访问定位

Java程序需要通过 JVM 栈上的引用访问堆中的具体对象。对象的访问方式取决于 JVM 虚拟机的实现。目前主流的访问方式有 句柄 和 直接指针 两种方式。

指针： 指向对象，代表一个对象在内存中的起始地址。

句柄： 可以理解为指向指针的指针，维护着对象的指针。句柄不直接指向对象，而是指向对象的指针（句柄不发生变化，指向固定内存地址），再由对象的指针指向对象的真实内存地址。

- 句柄访问

`Java`堆中划分出一块内存来作为句柄池，引用中存储对象的句柄地址，而句柄中包含了对象实例数据与对象类型数据各自的具体地址信息：

引用中存储的是稳定的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而引用本身不需要修改。

- 直接指针

如果使用直接指针访问，引用 中存储的直接就是对象地址，那么`Java`堆对象内部的布局中就必须考虑如何放置访问类型数据的相关信息。

优势：速度更快，节省了一次指针定位的时间开销。由于对象的访问在`Java`中非常频繁，因此这类开销积少成多后也是非常可观的执行成本。HotSpot 中采用的就是这种方式。

## JVM 的主要组成部分及其作用

两个子系统和两个组件：

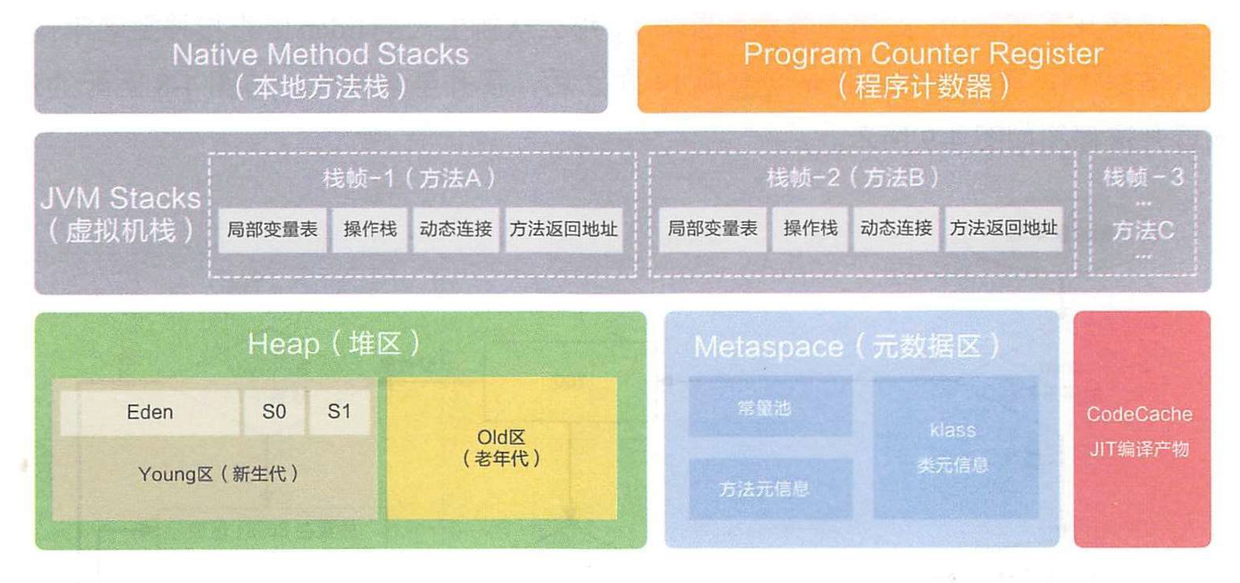
- Class loader(类装载)：根据给定的全限定名类名(如：java.lang.Object)来装载class文件到Runtime data area中的method area。

- Execution engine（执行引擎）：执行classes中的指令。

- Native Interface(本地接口)：与native libraries交互，是其它编程语言交互的接口。

- Runtime data area(运行时数据区域)：这就是我们常说的JVM的内存。

作用 ：首先通过编译器把 Java 代码转换成字节码，类加载器（ClassLoader）再把字节码加载到内存中，将其放在运行时数据区（Runtime data area）的方法区内，而字节码文件只是 JVM 的一套指令集规范，并不能直接交给底层操作系统去执行，因此需要特定的命令解析器执行引擎（Execution Engine），将字节码翻译成底层系统指令，再交由 CPU 去执行，而这个过程中需要调用其他语言的本地库接口（Native Interface）来实现整个程序的功能。



- 程序计数器（Program Counter Register）：当前线程所执行的字节码的行号指示器，字节码解析器的工作是通过改变这个计数器的值，来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能，都需要依赖这个计数器来完成；

- Java 虚拟机栈（Java Virtual Machine Stacks）：用于存储局部变量表、操作数栈、动态链接、方法出口等信息；

- 本地方法栈（Native Method Stack）：与虚拟机栈的作用是一样的，只不过虚拟机栈是服务 Java 方法的，而本地方法栈是为虚拟机调用 Native 方法服务的；

- Java 堆（Java Heap）：Java 虚拟机中内存最大的一块，是被所有线程共享的，几乎所有的对象实例都在这里分配内存；通过new关键字和构造器创建的对象

- 方法区（Methed Area）：用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译后的代码等数据。JDK8 之前，Hotspot 中方法区的实现是永久代（Perm），JDK8 开始使用元空间（Metaspace），以前永久代所有内容的字符串常量移至堆内存，其他内容移至元空间，元空间直接在本地内存分配。

## 说一下堆栈的区别？

物理地址

堆的物理地址分配对对象是不连续的。因此性能慢些。

栈物理地址分配是连续的。所以性能快。

内存分别

堆因为是不连续的，所以分配的内存是在运行期确认的，因此大小不固定。一般堆大小远远大于栈。

栈是连续的，所以分配的内存大小要在编译期就确认，大小是固定的。

存放的内容

堆存放的是对象的实例和数组。因此该区更关注的是数据的存储

栈存放：局部变量，操作数栈，返回结果。该区更关注的是程序方法的执行。

PS：

1. 静态变量放在方法区

2. 静态的对象还是放在堆。

程序的可见度

堆对于整个应用程序都是共享、可见的。

栈只对于线程是可见的。所以也是线程私有。他的生命周期和线程相同。

## 垃圾收集器

内存中已经不再被使用到的空间就是垃圾

java中，程序员是不需要显示的去释放一个对象的内存的，而是由虚拟机自行执行。

内存泄漏是指不再被使用的对象或者变量一直被占据在内存中。

理论上来说，Java是有GC垃圾回收机制的，也就是说，不再被使用的对象，会被GC自动回收掉，自动从内存中清除。

由于有这个垃圾回收机制，java中的对象不再有“作用域”的概念，只有引用的对象才有“作用域”。

垃圾回收器通常作为一个单独的低级别的线程运行，在不可预知的情况下对内存堆中已经死亡的或很长时间没有用过的对象进行清除和回收。

程序员不能实时的对某个对象或所有对象调用垃圾回收器进行垃圾回收。

GC 是垃圾收集的意思（Gabage Collection）,

### Java 中都有哪些引用类型？

- 强引用：无论如何都不会被回收的对象，发生 gc 的时候不会被回收。

- 软引用：在内存不足时会被回收。SoftReference类

- 弱引用：有用但不是必须的对象，在下一次GC时会被回收。weakReakReference，需要读取大量的本地图片，每次读图片都会影响性能，一次性全部加载又会造成内存溢出，所以使用软引用或者弱引用，，weakHashMap用来做缓存当发生gc时回收,

- 虚引用（幽灵引用/幻影引用）：无法通过虚引用获得对象，用 PhantomReference 实现虚引用，虚引用的用途是在 gc 时返回一个通知。配合引用队列使用，ReferenceQueue

### 判断对象是否可以被回收？

垃圾收集器在做垃圾回收的时候，首先需要判定的就是哪些内存是需要被回收的，哪些对象是「存活」的，是不可以被回收的；哪些对象已经「死掉」了，需要被回收。

一般有两种方法来判断：

- 引用计数器法：java不使用此方法。为每个对象创建一个引用计数，有对象引用时计数器 +1，引用被释放时计数 -1，当计数器为 0 时就可以被回收。它有一个缺点不能解决循环引用的问题；

- 可达性分析算法：从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是可以被回收的。

GCRoot对象：

1. 虚拟机栈中引用的对象（局部变量区）

2. 方法区中类静态属性引用的对象

3. 方法区中常量引用的对象

4. 本地方法栈中JNI （Natice方法）引用的对象

### JVM 的垃圾回收算法

- 引用计数，java不使用此方法

- 标记-清除算法：标记无用对象，然后进行清除回收。缺点：效率不高，无法清除垃圾碎片。

- 复制算法：按照容量划分二个大小相等的内存区域，当一块用完的时候将活着的对象复制到另一块上，然后再把已使用的内存空间一次清理掉。缺点：内存使用率不高，只有原来的一半。

- 标记-整理算法：标记无用对象，让所有存活的对象都向一端移动，然后直接清除掉端边界以外的内存。

- 分代算法：根据对象存活周期的不同将内存划分为几块，一般是新生代和老年代，新生代基本采用复制算法，老年代采用标记整理算法。

### 垃圾回收器

Serial收集器（复制算法): 新生代单线程收集器，标记和清理都是单线程，优点是简单高效；

ParNew收集器 (复制算法): 新生代收并行集器，实际上是Serial收集器的多线程版本，在多核CPU环境下有着比Serial更好的表现；

Parallel Scavenge收集器 (复制算法): 新生代并行收集器，追求高吞吐量，高效利用 CPU。吞吐量 = 用户线程时间/(用户线程时间+GC线程时间)，高吞吐量可以高效率的利用CPU时间，尽快完成程序的运算任务，适合后台应用等对交互相应要求不高的场景；

Serial Old收集器 (标记-整理算法): 老年代单线程收集器，Serial收集器的老年代版本；

Parallel Old收集器 (标记-整理算法)： 老年代并行收集器，吞吐量优先，Parallel Scavenge收集器的老年代版本；

CMS(Concurrent Mark Sweep)收集器（标记-清除算法）并发清除GC： 老年代并行收集器，以获取最短回收停顿时间为目标的收集器，具有高并发、低停顿的特点，追求最短GC回收停顿时间。

G1(Garbage First)收集器 (标记-整理算法)： Java堆并行收集器，G1收集器是JDK1.7提供的一个新收集器，G1收集器基于“标记-整理”算法实现，也就是说不会产生内存碎片。此外，G1收集器不同于之前的收集器的一个重要特点是：G1回收的范围是整个Java堆(包括新生代，老年代)，而前六种收集器回收的范围仅限于新生代或老年代。在停顿时间上加入预测机制。

内存区域不再是连续的，而是变成一个个大小一样的region，每个region1M到32M不等，还是分代收集器，逻辑上的分代概念

工作过程：

针对eden进行收集，数据移动到survivor区



### 分代垃圾回收器是怎么工作

分代回收器有两个分区：老生代和新生代，新生代默认的空间占比总空间的 1/3，老生代的默认占比是 2/3。

新生代使用的是复制算法，新生代里有 3 个分区：Eden、To Survivor、From Survivor，它们的默认占比是 8:1:1，它的执行流程如下：

- 把 Eden + From Survivor 存活的对象放入 To Survivor 区；

- 清空 Eden 和 From Survivor 分区；

- From Survivor 和 To Survivor 分区交换，From Survivor 变 To Survivor，To Survivor 变 From Survivor。

每次在 From Survivor 到 To Survivor 移动时都存活的对象，年龄就 +1，当年龄到达 15（默认配置是 15）时，升级为老生代。大对象也会直接进入老生代。

老生代当空间占用到达某个值之后就会触发全局垃圾收回，一般使用标记整理的执行算法。以上这些循环往复就构成了整个分代垃圾回收的整体执行流程。

## 内存分配策略

### 对象优先在 Eden 区分配

多数情况，对象都在新生代 Eden 区分配。当 Eden 区分配没有足够的空间进行分配时，虚拟机将会发起一次 Minor GC。如果本次 GC 后还是没有足够的空间，则将启用分配担保机制在老年代中分配内存。

这里我们提到 Minor GC，如果你仔细观察过 GC 日常，通常我们还能从日志中发现 Major GC/Full GC。

Minor GC 是指发生在新生代的 GC，因为 Java 对象大多都是朝生夕死，所有 Minor GC 非常频繁，一般回收速度也非常快；

Major GC/Full GC 是指发生在老年代的 GC，出现了 Major GC 通常会伴随至少一次 Minor GC。Major GC 的速度通常会比 Minor GC 慢 10 倍以上。

### 大对象直接进入老年代

所谓大对象是指需要大量连续内存空间的对象，频繁出现大对象是致命的，会导致在内存还有不少空间的情况下提前触发 GC 以获取足够的连续空间来安置新对象。

前面我们介绍过新生代使用的是标记-清除算法来处理垃圾回收的，如果大对象直接在新生代分配就会导致 Eden 区和两个 Survivor 区之间发生大量的内存复制。因此对于大对象都会直接在老年代进行分配。

### 长期存活对象将进入老年代

虚拟机采用分代收集的思想来管理内存，那么内存回收时就必须判断哪些对象应该放在新生代，哪些对象应该放在老年代。因此虚拟机给每个对象定义了一个对象年龄的计数器，如果对象在 Eden 区出生，并且能够被 Survivor 容纳，将被移动到 Survivor 空间中，这时设置对象年龄为 1。对象在 Survivor 区中每「熬过」一次 Minor GC 年龄就加 1，当年龄达到一定程度（默认 15） 就会被晋升到老年代。

## JVM调优

### OOM的常见异常

- Java.lang.StackOverflowError

- Java.lang.OutOfMemoryError:Java heap space

- Java.lang.OutOfMemeoryError:GC overhead limit exceeded

- Java.lang.OutOfMemeoryError:Direct buffer memory

- Java.lang.OutOfMemeoryError:unable to create new native thread

- Java.lang.OutOfMemeoryError:Metaspace

### JVM 调优的java工具？

JDK 自带了很多监控工具，都位于 JDK 的 bin 目录下，其中最常用的是 jconsole 和 jvisualvm 这两款视图监控工具。

- jconsole：用于对 JVM 中的内存、线程和类等进行监控；

- jvisualvm：JDK 自带的全能分析工具，可以分析：内存快照、线程快照、程序死锁、监控内存的变化、gc 变化等。

- jstack 全称： JVM Stack Trance 作用： 查看某个java进程的堆栈情况， 可用于确认死锁、IO等待、死循环等问题。

- jstat 作用： 查看进程中内存使用情况，但只能给出一些简单统计数据

- 统计加载了多少类以及占用空间 jstat -class pid

- 统计编译了多少文件 jstat -compiler 10

- jmap 全称： JVM Memory Map 作用：生成进程的内存堆快照 当需要看一下进程里是什么东西占用了过多内存时， 可以用jmap打印一下堆快照。

- 打印堆快照： jmap -dump:file=./dumpfile.dump 进程pid

- 查看特定类所占用的情况： jmap -histo:live 进程pid | grep 类名

- jhat 全称： JVM Heap Analysis Tool 和jmap配合， 可以解析jmap生成的堆快照， 支持生成1个web进程供我们分析和查看。

- jhat -J-Xmx515M dumpfile.dump

此时就会启动1个webServer，然后我们去访问就行了

- jdb 全称：Java Debugger 作用：用来对core文件和正在运行的Java进程进行实时地调试，类似于c++里的gdb

- 启动进程并调试: jdb -classpath . Test

- 至二级调试某进程: jdb -attach 8000 -sourcepath /Users/wefit/Development/study/java/jtest/src/

- jcmd 作用：多功能的工具，可以用它来导出堆、查看Java进程、导出线程信息、执行GC、还可以进行采样分析，可以理解为1个性能调优时用的工具。

- 查看 当前机器上所有的 jvm 进程信息: jcmd -l

- 查看指定进程的性能统计信息: jcmd pid PerfCounter.print

- 列出当前运行的 java 进程可以执行的操作: jcmd PID help

- 查看线程堆栈信息: jcmd PID Thread.print

- 查看堆内存信息： jcmd PID GC.heap\_dump FILE\_NAME

- jps 简单记法： JVM process status 全名：Java Virtual Machine Process Status Tool 作用： 显示 \*\*\*当前系统用户\*\*\* 的 \*\*\*所有\*\*\* Java进程情况及其进程号

- 查看进程jvm参数： jps -v

- 输出程序main class的完整package名或程序的jar文件完整路径名： jps -l

- 输出传递给main方法的参数: jps -m

- jinfo jvm infomation 作用：和jps功能类似， 但是支持根据指定pis查看指定进程

- 可以查看JVM参数、系统参数、调整jvm参数

- 但不支持查看java程序的内存使用情况

jinfo -flag xxx 查看xxx参数

- javap 把java字节码文件反汇编为Java源码文件。

### jvm参数类型

- 标配参数

-version、-help、-showversion

- X参数

-Xint：解释执行、 -Xcomp： 第一次使用就编译成本地代码、-Xmixed：混合模式

- XX参数

Boolean类型和KV设值类型

### 常用的 JVM 调优的参数都有哪些？

- -Xms2g：初始化推大小为 2g；等价于-XX:InitialHeapSize，默认为物理内存的1/64

- -Xmx2g：堆最大内存为 2g；等价于-XX:MaxHeapSize

- -Xss 设置单个线程的大小，一般默认为512K~1024K 等价于-XX:ThreadStackSize

- -Xmn 设置年轻代大小

- -XX:MetaspaceSize 设置元空间大小 -Xms10m -Xmx10m -XX:MetaspaceSize=1024m -XX:+PrintFlagsFinal

- -XX:NewRatio=4：设置年轻的和老年代的内存比例为 1:4；

- -XX:SurvivorRatio=8：设置新生代 Eden 和 Survivor 比例为 8:2；

- –XX:+UseParNewGC：指定使用 ParNew + Serial Old 垃圾回收器组合；

- -XX:+UseParallelOldGC：指定使用 ParNew + ParNew Old 垃圾回收器组合；

- -XX:+UseConcMarkSweepGC：指定使用 CMS + Serial Old 垃圾回收器组合；

- -XX:+PrintGC：开启打印 gc 信息；

- -XX:+PrintGCDetails：打印 gc 详细信息。

- -XX:+PrintFlagsInitial 查看初始默认值jvm参数

- -XX:+PrintFlagsFinal 查看修改更新参数

- -XX:+PrintCommandLineFlags

# Java集合框架

集合框架：用于存储数据的容器。

- 对象封装数据，对象多了也需要存储。集合用于存储对象。

- 对象的个数确定可以使用数组，对象的个数不确定的可以用集合。因为集合是可变长度的。

## 集合和数组的区别

- 数组是固定长度的；集合可变长度的。

- 数组可以存储基本数据类型，也可以存储引用数据类型；集合只能存储引用数据类型。

- 数组存储的元素必须是同一个数据类型；集合存储的对象可以是不同数据类型。

容量自增长；

提供了高性能的数据结构和算法，使编码更轻松，提高了程序速度和质量；

允许不同 API 之间的互操作，API之间可以来回传递集合；

可以方便地扩展或改写集合，提高代码复用性和可操作性。

通过使用JDK自带的集合类，可以降低代码维护和学习新API成本。

## List、Set、Map 之间的区别是什么？

Map接口和Collection接口是所有集合框架的父接口：

Collection接口的子接口包括：Set接口和List接口

Map接口的实现类主要有：HashMap、TreeMap、Hashtable、ConcurrentHashMap以及Properties等

Set接口的实现类主要有：HashSet、TreeSet、LinkedHashSet等

List接口的实现类主要有：ArrayList、LinkedList、Stack以及Vector等CopyonWriteArrayList

List 支持for循环，也就是通过下标来遍历，也可以用迭代器，但是set只能用迭代，因为他无序，无法用下标来取得想要的值。



Collection

List

Arraylist： Object数组

Vector： Object数组

LinkedList： 双向循环链表

CopyonWriteArrayList：写时复制List 读多写少且脏数据影响不大的并发情况下，选择CopyOnWriteArrayList

Set

HashSet（无序，唯一）：基于 HashMap 实现的，底层采用 HashMap 来保存元素

LinkedHashSet： LinkedHashSet 继承与 HashSet，并且其内部是通过 LinkedHashMap 来实现的。有点类似于我们之前说的LinkedHashMap 其内部是基于 Hashmap 实现一样，不过还是有一点点区别的。

TreeSet（有序，唯一）： 红黑树(自平衡的排序二叉树。)

CopyOnWriteArraySet：底层实现用的是读写复制ListMap

HashMap： JDK1.8之前HashMap由数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突）.JDK1.8以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间

LinkedHashMap：LinkedHashMap 继承自 HashMap，所以它的底层仍然是基于拉链式散列结构即由数组和链表或红黑树组成。另外，LinkedHashMap 在上面结构的基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。

HashTable： 数组+链表组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的

TreeMap： 红黑树（自平衡的排序二叉树）

## 怎么确保一个集合不能被修改？

可以使用 Collections. unmodifiableCollection(Collection c) 方法来创建一个只读集合，这样改变集合的任何操作都会抛出 Java. lang. UnsupportedOperationException 异常。

Arrays.asList()只会返回1个固定大小的列表， 其返回的List是AbstractList，无法调用add、remove和clear， 如果调用会直接抛异常。

异常名字为UnsupportedOperationException

## 迭代器 Iterator

Iterator 接口提供遍历任何 Collection 的接口。我们可以从一个 Collection 中使用迭代器方法来获取迭代器实例。迭代器取代了 Java 集合框架中的 Enumeration，迭代器允许调用者在迭代过程中移除元素。

Iterator 的特点是只能单向遍历，但是更加安全，因为它可以确保，在当前遍历的集合元素被更改的时候，就会抛出 ConcurrentModificationException 异常。

边遍历边修改 Collection 的唯一正确方式是使用 Iterator.remove()

Iterator 和 ListIterator 有什么区别？

Iterator 可以遍历 Set 和 List 集合，而 ListIterator 只能遍历 List。

Iterator 只能单向遍历，而 ListIterator 可以双向遍历（向前/后遍历）。

ListIterator 实现 Iterator 接口，然后添加了一些额外的功能，比如添加一个元素、替换一个元素、获取前面或后面元素的索引位置。

## ArrayList

ArrayList的优点如下：

ArrayList 底层以数组实现，是一种随机访问模式。ArrayList 实现了 RandomAccess 接口，因此查找的时候非常快。

ArrayList 在顺序添加一个元素的时候非常方便。

ArrayList 的缺点如下：

删除元素的时候，需要做一次元素复制操作。如果要复制的元素很多，那么就会比较耗费性能。

插入元素的时候，也需要做一次元素复制操作，缺点同上。

ArrayList 比较适合顺序添加、随机访问的场景。

ArrayList 不是线程安全的，如果遇到多线程场景，可以通过 Collections 的 synchronizedList 方法将其转换成线程安全的容器后再使用

### 数组和 List 之间的转换？

- 数组转 List：使用 Arrays. asList(array) 进行转换。

- List 转数组：使用 List 自带的 toArray() 方法。

### ArrayList 和 LinkedList 的区别是什么？

数据结构实现：ArrayList 是动态数组的数据结构实现，而 LinkedList 是双向链表的数据结构实现。

随机访问效率：ArrayList 比 LinkedList 在随机访问的时候效率要高，因为 LinkedList 是线性的数据存储方式，所以需要移动指针从前往后依次查找。

增加和删除效率：在非首尾的增加和删除操作，LinkedList 要比 ArrayList 效率要高，因为 ArrayList 增删操作要影响数组内的其他数据的下标。

内存空间占用：LinkedList 比 ArrayList 更占内存，因为 LinkedList 的节点除了存储数据，还存储了两个引用，一个指向前一个元素，一个指向后一个元素。

线程安全：ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；

综合来说，在需要频繁读取集合中的元素时，更推荐使用 ArrayList，而在插入和删除操作较多时，更推荐使用 LinkedList

## Map

hashMap是线程不安全的，HashMap是数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的，采用哈希表来存储的, 它根据键的hashCode值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序却是不确定的。 HashMap最多只允许一条记录的键为null，允许多条记录的值为null。HashMap非线程安全，即任一时刻可以有多个线程同时写HashMap，可能会导致数据的不一致。如果需要满足线程安全，可以用 Collections的synchronizedMap方法使HashMap具有线程安全的能力，或者使用ConcurrentHashMap。

Hashtable：Hashtable是遗留类，很多映射的常用功能与HashMap类似，不同的是它承自Dictionary类，并且是线程安全的，任一时间只有一个线程能写Hashtable，并发性不如ConcurrentHashMap，因为ConcurrentHashMap引入了分段锁。Hashtable不建议在新代码中使用，不需要线程安全的场合可以用HashMap替换，需要线程安全的场合可以用ConcurrentHashMap替换。

LinkedHashMap：LinkedHashMap是HashMap的一个子类，保存了记录的插入顺序，在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的，也可以在构造时带参数，按照访问次序排序。

TreeMap：TreeMap实现SortedMap接口，能够把它保存的记录根据键排序，默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator遍历TreeMap时，得到的记录是排过序的。如果使用排序的映射，建议使用TreeMap。在使用TreeMap时，key必须实现Comparable接口或者在构造TreeMap传入自定义的Comparator，否则会在运行时抛出java.lang.ClassCastException类型的异常。

对于上述四种Map类型的类，要求映射中的key是不可变对象。不可变对象是该对象在创建后它的哈希值不会被改变。如果对象的哈希值发生变化，Map对象很可能就定位不到映射的位置了。

## HashMap的内部实现

HashMap 是基于哈希表的 Map 接口的非同步实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

### 结构实现

从结构实现来讲，HashMap是数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的，(1) 从源码可知，HashMap类中有一个非常重要的字段，就是 Node[] table，即哈希桶数组，明显它是一个Node的数组。我们来看Node[]是何物。

static class Node<K,V> implements Map.Entry<K,V> {

final int hash; //用来定位数组索引位置

final K key;

V value;

Node<K,V> next; //链表的下一个node

}

Node是HashMap的一个内部类，实现了Map.Entry接口，本质是就是一个映射(键值对)。上图中的每个黑色圆点就是一个Node对象。

HashMap就是使用哈希表来存储的。哈希表为解决冲突，可以采用开放地址法和链地址法等来解决问题，Java中HashMap采用了链地址法。链地址法，简单来说，就是数组加链表的结合。在每个数组元素上都一个链表结构，当数据被Hash后，得到数组下标，把数据放在对应下标元素的链表上。

系统调用key的hashCode()方法得到其hashCode 值（该方法适用于每个Java对象），然后再通过Hash算法的后两步运算（高位运算和取模运算，下文有介绍）来定位该键值对的存储位置，有时两个key会定位到相同的位置，表示发生了Hash碰撞。当然Hash算法计算结果越分散均匀，Hash碰撞的概率就越小，map的存取效率就会越高。

如果哈希桶数组很大，即使较差的Hash算法也会比较分散，如果哈希桶数组数组很小，即使好的Hash算法也会出现较多碰撞，所以就需要在空间成本和时间成本之间权衡，其实就是在根据实际情况确定哈希桶数组的大小，并在此基础上设计好的hash算法减少Hash碰撞。那么通过什么方式来控制map使得Hash碰撞的概率又小，哈希桶数组（Node[] table）占用空间又少呢？答案就是好的Hash算法和扩容机制。

在理解Hash和扩容流程之前，我们得先了解下HashMap的几个字段。从HashMap的默认构造函数源码可知，构造函数就是对下面几个字段进行初始化，源码如下：

int threshold; // 所能容纳的key-value对极限

final float loadFactor; // 负载因子

int modCount;

int size;

首先，Node[] table的初始化长度length(默认值是16)，Load factor为负载因子(默认值是0.75)，threshold是HashMap所能容纳的最大数据量的Node(键值对)个数。threshold = length \* Load factor。也就是说，在数组定义好长度之后，负载因子越大，所能容纳的键值对个数越多。

结合负载因子的定义公式可知，threshold就是在此Load factor和length(数组长度)对应下允许的最大元素数目，超过这个数目就重新resize(扩容)，扩容后的HashMap容量是之前容量的两倍。默认的负载因子0.75是对空间和时间效率的一个平衡选择，建议大家不要修改，除非在时间和空间比较特殊的情况下，如果内存空间很多而又对时间效率要求很高，可以降低负载因子Load factor的值；相反，如果内存空间紧张而对时间效率要求不高，可以增加负载因子loadFactor的值，这个值可以大于1。

size这个字段其实很好理解，就是HashMap中实际存在的键值对数量。注意和table的长度length、容纳最大键值对数量threshold的区别。而modCount字段主要用来记录HashMap内部结构发生变化的次数，主要用于迭代的快速失败。强调一点，内部结构发生变化指的是结构发生变化，例如put新键值对，但是某个key对应的value值被覆盖不属于结构变化。

在HashMap中，哈希桶数组table的长度length大小必须为2的n次方(一定是合数)，这是一种非常规的设计，常规的设计是把桶的大小设计为素数。相对来说素数导致冲突的概率要小于合数，具体证明可以参考http://blog.csdn.net/liuqiyao\_01/article/details/14475159，Hashtable初始化桶大小为11，就是桶大小设计为素数的应用（Hashtable扩容后不能保证还是素数）。HashMap采用这种非常规设计，主要是为了在取模和扩容时做优化，同时为了减少冲突，HashMap定位哈希桶索引位置时，也加入了高位参与运算的过程。

这里存在一个问题，即使负载因子和Hash算法设计的再合理，也免不了会出现拉链过长的情况，一旦出现拉链过长，则会严重影响HashMap的性能。于是，在JDK1.8版本中，对数据结构做了进一步的优化，引入了红黑树。而当链表长度太长（默认超过8）时，链表就转换为红黑树，利用红黑树快速增删改查的特点提高HashMap的性能，其中会用到红黑树的插入、删除、查找等算法。

### 确定哈希桶数组索引位置

这里的Hash算法本质上就是三步：取key的hashCode值、高位运算、取模运算。

对于任意给定的对象，只要它的hashCode()返回值相同，那么程序调用方法一所计算得到的Hash码值总是相同的。我们首先想到的就是把hash值对数组长度取模运算，这样一来，元素的分布相对来说是比较均匀的。但是，模运算的消耗还是比较大的，在HashMap中是这样做的：调用方法二来计算该对象应该保存在table数组的哪个索引处。

方法一：

static final int hash(Object key) { //jdk1.8 & jdk1.7

int h;

// h = key.hashCode() 为第一步 取hashCode值

// h ^ (h >>> 16) 为第二步 高位参与运算

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

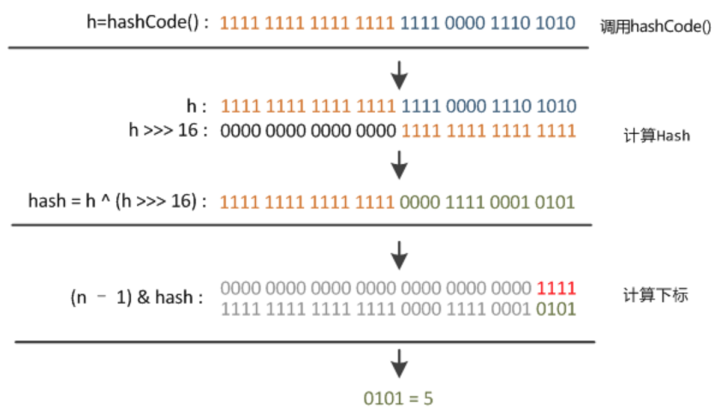
方法二：

static int indexFor(int h, int length) { //jdk1.7的源码，jdk1.8没有这个方法，但是实现原理一样的

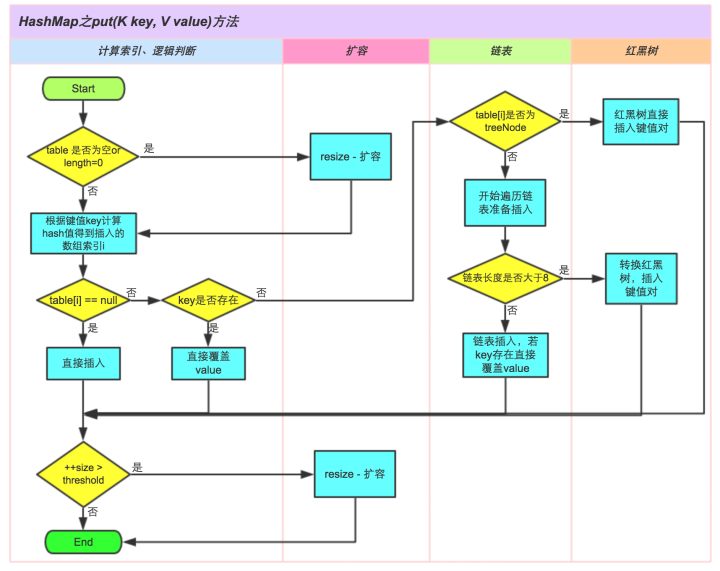
return h & (length-1); //第三步 取模运算

}

这个方法非常巧妙，它通过h & (table.length -1)来得到该对象的保存位，而HashMap底层数组的长度总是2的n次方，这是HashMap在速度上的优化。当length总是2的n次方时，h& (length-1)运算等价于对length取模，也就是h%length，但是&比%具有更高的效率。



### 分析HashMap的put方法



①.判断键值对数组table[i]是否为空或为null，否则执行resize()进行扩容；

②.根据键值key计算hash值得到插入的数组索引i，如果table[i]==null，直接新建节点添加，转向⑥，如果table[i]不为空，转向③；

③.判断table[i]的首个元素是否和key一样，如果相同直接覆盖value，否则转向④，这里的相同指的是hashCode以及equals；

④.判断table[i] 是否为treeNode，即table[i] 是否是红黑树，如果是红黑树，则直接在树中插入键值对，否则转向⑤；

⑤.遍历table[i]，判断链表长度是否大于8，大于8的话把链表转换为红黑树，在红黑树中执行插入操作，否则进行链表的插入操作；遍历过程中若发现key已经存在直接覆盖value即可；

⑥.插入成功后，判断实际存在的键值对数量size是否超多了最大容量threshold，如果超过，进行扩容。

### 扩容机制

扩容(resize)就是重新计算容量，向HashMap对象里不停的添加元素，而HashMap对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。当然Java里的数组是无法自动扩容的，方法是使用一个新的数组代替已有的容量小的数组，就像我们用一个小桶装水，如果想装更多的水，就得换大水桶。

JDK1.7的代码，好理解一些，本质上区别不大，具体区别后文再说。

1 void resize(int newCapacity) { //传入新的容量

2 Entry[] oldTable = table; //引用扩容前的Entry数组

3 int oldCapacity = oldTable.length;

4 if (oldCapacity == MAXIMUM\_CAPACITY) { //扩容前的数组大小如果已经达到最大(2^30)了

5 threshold = Integer.MAX\_VALUE; //修改阈值为int的最大值(2^31-1)，这样以后就不会扩容了

6 return;

7 }

8

9 Entry[] newTable = new Entry[newCapacity]; //初始化一个新的Entry数组

10 transfer(newTable); //！！将数据转移到新的Entry数组里

11 table = newTable; //HashMap的table属性引用新的Entry数组

12 threshold = (int)(newCapacity \* loadFactor);//修改阈值

13 }

1 void transfer(Entry[] newTable) {

2 Entry[] src = table; //src引用了旧的Entry数组

3 int newCapacity = newTable.length;

4 for (int j = 0; j < src.length; j++) { //遍历旧的Entry数组

5 Entry<K,V> e = src[j]; //取得旧Entry数组的每个元素

6 if (e != null) {

7 src[j] = null;//释放旧Entry数组的对象引用（for循环后，旧的Entry数组不再引用任何对象）

8 do {

9 Entry<K,V> next = e.next;

10 int i = indexFor(e.hash, newCapacity); //！！重新计算每个元素在数组中的位置

11 e.next = newTable[i]; //标记[1]

12 newTable[i] = e; //将元素放在数组上

13 e = next; //访问下一个Entry链上的元素

14 } while (e != null);

15 }

16 }

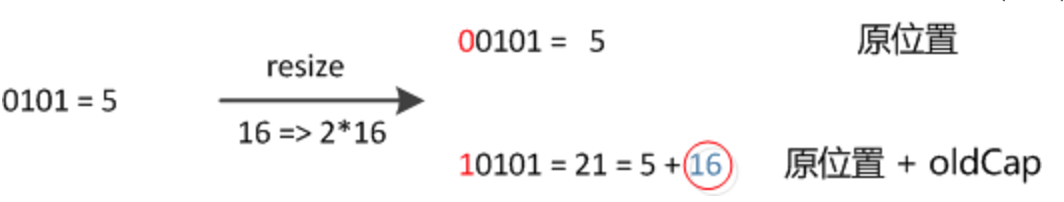
17 }

newTable[i]的引用赋给了e.next，也就是使用了单链表的头插入方式，同一位置上新元素总会被放在链表的头部位置；这样先放在一个索引上的元素终会被放到Entry链的尾部(如果发生了hash冲突的话），这一点和Jdk1.8有区别，下文详解。在旧数组中同一条Entry链上的元素，通过重新计算索引位置后，有可能被放到了新数组的不同位置上。

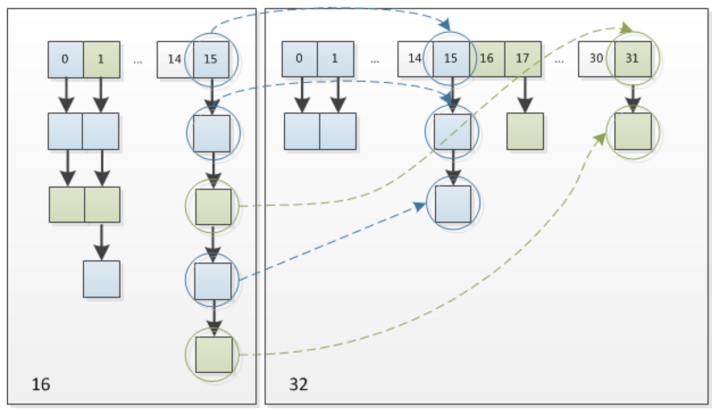
下面我们讲解下JDK1.8做了哪些优化。经过观测可以发现，我们使用的是2次幂的扩展(指长度扩为原来2倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。看下图可以明白这句话的意思，n为table的长度，图（a）表示扩容前的key1和key2两种key确定索引位置的示例，图（b）表示扩容后key1和key2两种key确定索引位置的示例，其中hash1是key1对应的哈希与高位运算结果。



元素在重新计算hash之后，因为n变为2倍，那么n-1的mask范围在高位多1bit(红色)，因此新的index就会发生这样的变化：



因此，我们在扩充HashMap的时候，不需要像JDK1.7的实现那样重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”，可以看看下图为16扩充为32的resize示意图：



这个设计确实非常的巧妙，既省去了重新计算hash值的时间，而且同时，由于新增的1bit是0还是1可以认为是随机的，因此resize的过程，均匀的把之前的冲突的节点分散到新的bucket了。这一块就是JDK1.8新增的优化点。有一点注意区别，JDK1.7中rehash的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置，但是从上图可以看出，JDK1.8不会倒置。有兴趣的同学可以研究下JDK1.8的resize源码，写的很赞:

### ****线程安全性****

在多线程使用场景中，应该尽量避免使用线程不安全的HashMap，而使用线程安全的ConcurrentHashMap。那么为什么说HashMap是线程不安全的，下面举例子说明在并发的多线程使用场景中使用HashMap可能造成死循环。代码例子如下(便于理解，仍然使用JDK1.7的环境)：

### Fail-Fast 机制

java.util.HashMap 不是线程安全的，因此如果在使用迭代器的过程中有其他线程修改了 map，那么将抛出 ConcurrentModificationException，这就是所谓 fail-fast 策略。一种错误机制。 当多个线程对同一个集合的内容进行操作时，就可能会产生 fail-fast 事件。某一个线程 A 通过 iterator去遍历某集合的过程中，若该集合的内容被其他线程所改变了；那么线程 A 访问集合时，就会抛出 ConcurrentModificationException 异常，产生 fail-fast 事件。

这一策略在源码中的实现是通过 modCount 域，modCount 顾名思义就是修改次数，对 HashMap 内容（当然不仅仅是 HashMap 才会有，其他例如 ArrayList 也会）的修改都将增加这个值（大家可以再回头看一下其源码，在很多操作中都有 modCount++ 这句），那么在迭代器初始化过程中会将这个值赋给迭代器的 expectedModCount。





### ****小结****

(1) 扩容是一个特别耗性能的操作，所以当程序员在使用HashMap的时候，估算map的大小，初始化的时候给一个大致的数值，避免map进行频繁的扩容。

(2) 负载因子是可以修改的，也可以大于1，但是建议不要轻易修改，除非情况非常特殊。

(3) HashMap是线程不安全的，不要在并发的环境中同时操作HashMap，建议使用ConcurrentHashMap。

(4) JDK1.8引入红黑树大程度优化了HashMap的性能。

## ConcurrentHashMap

Hashtable容器使用synchronized来保证线程安全，但在线程竞争激烈的情况下Hashtable的效率非常低下。因为当一个线程访问Hashtable的同步方法时，其他线程访问Hashtable的同步方法时，可能会进入阻塞或轮询状态。如线程1使用put进行添加元素，线程2不但不能使用put方法添加元素，并且也不能使用get方法来获取元素，所以竞争越激烈效率越低。

HashTable容器在竞争激烈的并发环境下表现出效率低下的原因，是因为所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。这里“按顺序”是很重要的，否则极有可能出现死锁，在ConcurrentHashMap内部，段数组是final的，并且其成员变量实际上也是final的，但是，仅仅是将数组声明为final的并不保证数组成员也是final的，这需要实现上的保证。这可以确保不会出现死锁，因为获得锁的顺序是固定的。

ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁ReentrantLock，在ConcurrentHashMap里扮演锁的角色，HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护者一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。

JDK1.8的实现已经抛弃了Segment分段锁机制，利用CAS+Synchronized来保证并发更新的安全。数据结构采用：数组+链表+红黑树。

Hashtable的任何操作都会把整个表锁住，是阻塞的。好处是总能获取最实时的更新，比如说线程A调用putAll写入大量数据，期间线程B调用get，线程B就会被阻塞，直到线程A完成putAll，因此线程B肯定能获取到线程A写入的完整数据。坏处是所有调用都要排队，效率较低。

ConcurrentHashMap 是设计为非阻塞的。在更新时会局部锁住某部分数据，但不会把整个表都锁住。同步读取操作则是完全非阻塞的。好处是在保证合理的同步前提下，效率很高。坏处是严格来说读取操作不能保证反映最近的更新。例如线程A调用putAll写入大量数据，期间线程B调用get，则只能get到目前为止已经顺利插入的部分数据。

应该根据具体的应用场景选择合适的HashMap。

## HashSet 的实现原理

默认情况下采用的是 initial capacity为16，load factor 为 0.75。

### HashSet 的实现

对于 HashSet 而言，它是基于 HashMap 实现的，HashSet 底层使用 HashMap 来保存所有元素，因此 HashSet 的实现比较简单，相关 HashSet 的操作，基本上都是直接调用底层 HashMap 的相关方法来完成，我们应该为保存到 HashSet 中的对象覆盖 hashCode() 和 equals()

### add 方法



如果此 set 中尚未包含指定元素，则添加指定元素。更确切地讲，如果此 set 没有包含满足(e==null ? e2==null : e.equals(e2)) 的元素 e2，则向此 set 添加指定的元素 e。如果此 set 已包含该元素，则该调用不更改 set 并返回 false。但底层实际将将该元素作为 key 放入 HashMap。

由于 HashMap 的 put() 方法添加 key-value 对时，当新放入 HashMap 的 Entry 中 key 与集合中原有 Entry 的 key 相同（hashCode()返回值相等，通过 equals 比较也返回 true），新添加的 Entry 的 value 会将覆盖原来 Entry 的 value（HashSet 中的 value 都是PRESENT），但 key 不会有任何改变，因此如果向 HashSet 中添加一个已经存在的元素时，新添加的集合元素将不会被放入 HashMap中，原来的元素也不会有任何改变，这也就满足了 Set 中元素不重复的特性。

该方法如果添加的是在 HashSet 中不存在的，则返回 true；如果添加的元素已经存在，返回 false。其原因在于我们之前提到的关于 HashMap 的 put 方法。该方法在添加 key 不重复的键值对的时候，会返回 null。



## Queue接口

Queue： 基本上，一个队列就是一个先入先出（FIFO）的数据结构

Queue接口与List、Set同一级别，都是继承了Collection接口。LinkedList实现了Deque接口

### 非阻塞队列

非阻塞队列不能阻塞，多线程时，当队列满或者队列空时，只能使用队列wait()，notify()进行队列消息传送。

1. PriorityQueue

PriorityQueue类实质上维护了一个有序列表。加入到 Queue 中的元素根据它们的天然排序（通过其 java.util.Comparable 实现）或者根据传递给构造函数的 java.util.Comparator 实现来定位。该队列不允许使用 null 元素也不允许插入不可比较的对象

PriorityQueue 队列的头指排序规则最小那个元素。如果多个元素都是最小值则随机选一个。

PriorityQueue 是一个无界队列，但是初始的容量(实际是一个Object[])，随着不断向优先级队列添加元素，其容量会自动扩容，无需指定容量增加策略的细节。

2. ConcurrentLinkedQueue

ConcurrentLinkedQueue 是基于链接节点的、线程安全的队列（CAS）。并发访问不需要同步。因为它在队列的尾部添加元素并从头部删除它们，所以只要不需要知道队列的大小，ConcurrentLinkedQueue 对公共集合的共享访问就可以工作得很好。收集关于队列大小的信息会很慢，需要遍历队列。

同样此队列不允许使用null元素，

3. 非阻塞队列的操作

add(E e)：将元素e插入到队列末尾，如果插入成功，则返回true；如果插入失败（即队列已满），则会抛出异常；

remove()：移除队首元素，若移除成功，则返回true；如果移除失败（队列为空），则会抛出异常；

remove(Object o)：移除指定的元素，若移除成功，则返回true；如果移除失败（队列为空），则会抛出异常

offer(E e)：将元素e插入到队列末尾，如果插入成功，则返回true；如果插入失败（即队列已满），则返回false；

poll()：移除并获取队首元素，若成功，则返回队首元素；否则返回null；

peek()：获取队首元素，若成功，则返回队首元素；否则返回null

### 阻塞队列

阻塞队列可以阻塞，当阻塞队列当队列里面没有值时，会阻塞直到有值输入。输入也一样，当队列满的时候，会阻塞，直到队列不为空。

1. 阻塞队列

ArrayBlockingQueue ：一个由数组支持的有界队列。

LinkedBlockingQueue ：一个由链接节点支持的可选有界队列。

PriorityBlockingQueue ：一个由优先级堆支持的无界优先级队列。

DelayQueue ：一个由优先级堆支持的、基于时间的调度队列。

SynchronousQueue ：一个利用 BlockingQueue 接口的简单聚集（rendezvous）机制。

2. 阻塞队列的操作

阻塞队列包括了非阻塞队列中的大部分方法，上面列举的5个方法在阻塞队列中都存在，但是要注意这5个方法在阻塞队列中都进行了同步措施。

除此之外，阻塞队列提供了另外4个非常有用的方法：

put(E e)：用来向队尾存入元素，如果队列满，则等待；

take()：用来从队首取元素，如果队列为空，则等待；

offer(E e,long timeout, TimeUnit unit)：用来向队尾存入元素，如果队列满，则等待一定的时间，当时间期限达到时，如果还没有插入成功，则返回false；否则返回true；

poll(long timeout, TimeUnit unit)：用来从队首取元素，如果队列空，则等待一定的时间，当时间期限达到时，如果取到，则返回null；否则返回取得的元素；

# Java设计模式

设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。设计模式是代码可用性的延伸

设计模式分类：创建型模式，结构型模式，行为型模式

## 设计模式原则

### - 单一职责原则

对类来说，即一个类应该只负责一项职责，如 A 类负责两个不同的职责：职责1、职责2,。当职责1 需求变更而改变 A 时，可能造成职责2 执行错误，所以需要将 A 类 的细粒度分解为 A1 和 A2

### - 接口隔离原则

客户端不应该依赖它不需要的接口，即一个类对另一个类的依赖应该建立在最小的接口

### - 依赖倒置原则：

高层模块不应该依赖低层模块，二者应该依赖其抽象

抽象不应该依赖细节，细节应该依赖抽象

依赖倒置的中心思想是面向接口编程

依赖倒置原则是基于这样的设计理念：相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建的架构比以细节为基础的架构要稳定的多。在 Java 中，抽象指的是接口或抽象类，细节就是具体的实现类

使用接口或抽象类的目的就是制定好规范，而不涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成

### - 里氏替换原则

在使用继承时，遵循里氏替换原则，在子类中 \*\*尽量\*\* 不要重写父类的方法

里氏替换原则告诉我们，继承实际上是让两个类耦合性增强了，\*\*在适当的情况下，可以通过聚合，组合，依赖来解决问题\*\*

通用的做法是：原来的父类和子类都继承一个更通俗的基类，原有的继承关系去掉，采用依赖，聚合，组合等关系代替

### - 开闭原则

一个软件实体，如类、模块和函数应该对扩展开放（对提供方），对修改关闭（对使用方）。用抽象构建框架，用实现扩展细节

当软件需要变化时，尽量 \*\*通过扩展\*\* 软件实体的行为来实现变化，而 \*\*不是通过修改\*\* 已有的代码来实现变化

编程中遵循其它原则，以及使用设计模式的目的就是遵循 \*\*开闭原则\*\*

### - 迪米特法则

一个对象应该对其他对象保持最少的了解

类与类关系关系越密切，耦合度越大

迪米特法则又叫 最少知道原则，即一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类不管多么复杂，都尽量将逻辑封装在类的内部。对外除了提供的 public 方法，不对外泄露任何消息

迪米特法则还有个更简单的定义：只与直接的朋友通信

直接的朋友：每个对象都会与其他的对象有耦合关系，只要两个对象之间有耦合关系，我们就可以说这两个对象之间的朋友关系。耦合的方式有很多，依赖、关联、组合，聚合等。其中，我们称出现成员变量，方法参数，方法返回值中的类为直接的朋友，而出现在局部变量中的类不是直接的朋友。也就是说，陌生的类最好不要以局部变量的形式出现在类的内部

### - 合成复用原则

尽量使用合成/聚合的方式，而不是使用继承

找出应用中可能需要变化之处，把它们独立出来，不要和那些不需要变化的代码混在一起

针对接口编程，而不是针对实现编程

为了交互对象之间的 \*\*松耦合设计\*\* 而努力

## 静态代理、JDK动态代理以及CGLIB动态代理

### - 静态代理

静态代理其实就是在程序运行之前，提前写好被代理方法的代理类，编译后运行。

静态代理需要针对被代理的方法提前写好代理类，如果被代理的方法非常多则需要编写很多代码，因此，对于上述缺点，通过动态代理的方式进行了弥补。

### - 动态代理

动态代理主要是通过反射机制，在运行时动态生成所需代理的class.

无论是动态代理还是静态带领，都需要定义接口，然后才能实现代理功能。这同样存在局限性，因此，为了解决这个问题，出现了第三种代理方式：cglib代理。

cglib代理

CGLib采用了非常底层的字节码技术，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。

JDK动态代理与CGLib动态代理均是实现Spring AOP的基础。

代理对象的生成过程由Enhancer类实现，大概步骤如下：

1. 生成代理类Class的二进制字节码；

2. 通过Class.forName加载二进制字节码，生成Class对象；

3. 通过反射机制获取实例构造，并初始化代理类对象

Spring 框架使用 Spring AOP 模块中的代理模式。在 Spring AOP 中，笔者可以创建对象的代理来实现横切关注点。

## 单例模式

程序中只有一个实例

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

主要解决：一个全局使用的类频繁地创建与销毁。

- 构造器私有化

- 自行创建实例

- 向外提供实例

懒汉式，线程安全 延迟创建对象

1. 线程不安全，先判断有没有。

2. 线程安全，使用synchronized锁定getInstance

3. 静态内部类去创建

饿汉式，线程安全 直接创建对象

1. 直接实例化

2. 枚举

3. 静态代码快饿汉式（复杂实例化）

Spring 框架提供了一个单例的 bean 来实现单例模式。

## 工厂模式

工厂模式（Factory Pattern）是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

意图：定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。

主要解决：主要解决接口选择的问题。

实现创建者和调用者的分离

类很复杂初始化由工厂来实现，简化代码量，减少繁琐的实例化过程

一个类有很多的子类可以使用工厂类来创建不同的子类

Spring 框架使用工厂模式来实现 Spring 容器的 BeanFactory 和 ApplicationContext 接口。Spring 容器基于工厂模式为 Spring 应用程序创建 bean，并管理着每一个 bean 的生命周期。BeanFactory 和 ApplicationContext 是工厂接口，并且在 Spring 中存在有很多实现类。getBean() 方法是相对应的 bean 的工厂方法。

### 抽象工厂模式

定义了一个接口用于创建相关或有依赖关系的对象簇，而无需明确指定具体类。

在 Spring 框架中，FactoryBean 接口是基于抽象工厂模式设计的。Spring 提供了很多这个接口的实现，比如 ProxyFactoryBean、JndiFactoryBean、LocalSessionFactoryBean、LocalContainerEntityManagerFactoryBean 等。FactoryBean 帮助 Spring 构建它自己无法轻松构建的对象。通常这是用来构造具有许多依赖关系的复杂对象。它也可以根据配置构造高易变的逻辑。例如，在 Spring 框架中，LocalSessionFactoryBean 是 FactoryBean 的一个实现，它用于获取 Hibernate 配置的关联的 bean 的引用。这是一个数据源的特定配置，它在得到 SessionFactory 的对象之前被使用。对此，在一致的情况下可以用 LocalSessionFactoryBean 获取特定的数据源配置。读者可以将 FactoryBean 的 getObject() 方法的返回结果注入到任何其他属性中。

## 观察者模式

当对象间存在一对多关系时，则使用观察者模式（Observer Pattern）。比如，当一个对象被修改时，则会自动通知它的依赖对象。观察者模式属于行为型模式。

意图：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依 赖于它的对象都得到通知并被自动更新。

主要解决：一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。

在 Spring 框架中，观察者模式用于实现 ApplicationContext 的事件处理功能。Spring 为我们提供了 ApplicationEvent 类和 ApplicationListener 接口来启用事件处理。Spring 应用程序中的任何 bean 实现 ApplicationListener 接口，都会接收到 ApplicationEvent 作为事件发布者推送的消息。在这里，事件发布者是主题(Subject) 和实现 ApplicationListener 的 bean 的观察者(Observer)。

## 秒杀系统设计

### 什么是秒杀

通俗一点讲就是网络商家为促销等目的组织的网上限时抢购活动

业务特点

- 高并发：秒杀的特点就是这样时间极短、 瞬间用户量大。

- 库存量少：一般秒杀活动商品量很少，这就导致了只有极少量用户能成功购买到。

- 业务简单：流程比较简单，一般都是下订单、扣库存、支付订单

- 恶意请求，数据库压力大

解决方案

- 前端：页面资源静态化，按钮控制，使用答题校验码可以防止秒杀器的干扰，让更多用户有机会抢到

- nginx：校验恶意请求，转发请求，负载均衡；动静分离，不走tomcat获取静态资源；gzip压缩，减少静态文件传输的体积，节省带宽，提高渲染速度

- 业务层：集群，多台机器处理，提高并发能力

- redis：集群保证高可用，持久化数据；分布式锁（悲观锁）；缓存热点数据（库存）

- mq：削峰限流，MQ堆积订单，保护订单处理层的负载，Consumer根据自己的消费能力来取Task，实际上下游的压力就可控了。重点做好路由层和MQ的安全

- 数据库：读写分离，拆分事务提高并发度

### 秒杀系统设计小结

秒杀系统就是一个“三高”系统，即高并发、高性能和高可用的分布式系统

秒杀设计原则：前台请求尽量少，后台数据尽量少，调用链路尽量短，尽量不要有单点

秒杀高并发方法：访问拦截、分流、动静分离

秒杀数据方法：减库存策略、热点、异步、限流降级

访问拦截主要思路：通过CDN和缓存技术，尽量把访问拦截在离用户更近的层，尽可能地过滤掉无效请求。

分流主要思路：通过分布式集群技术，多台机器处理，提高并发能力。

## 分布式系统设计

布式系统的目标是提升系统的整体性能和吞吐量另外还要尽量保证分布式系统的容错性（假如增加10台服务器才达到单机运行效果2倍左右的性能，那么这个分布式系统就根本没有存在的意义）。

即使采用了分布式系统，我们也要尽力运用并发编程、高性能网络框架等等手段提升单机上的程序性能。

### 分布式系统设计两大思路：中心化和去中心化

\*\*中心化设计\*\*

- 两个角色： 中心化的设计思想很简单，分布式集群中的节点机器按照角色分工，大体上分为两种角色： “领导” 和 “干活的”

- 角色职责： “领导”通常负责分发任务并监督“干活的”，发现谁太闲了，就想发设法地给其安排新任务，确保没有一个“干活的”能够偷懒，如果“领导”发现某个“干活的”因为劳累过度而病倒了，则是不会考虑先尝试“医治”他的，而是一脚踢出去，然后把他的任务分给其他人。其中微服务架构 Kubernetes 就恰好采用了这一设计思路。

- 中心化设计的问题

- 中心化的设计存在的最大问题是“领导”的安危问题，如果“领导”出了问题，则群龙无首，整个集群就奔溃了。但我们难以同时安排两个“领导”以避免单点问题。

- 中心化设计还存在另外一个潜在的问题，既“领导”的能力问题：可以领导10个人高效工作并不意味着可以领导100个人高效工作，所以如果系统设计和实现得不好，问题就会卡在“领导”身上。

- 领导安危问题的解决办法： 大多数中心化系统都采用了主备两个“领导”的设计方案，可以是热备或者冷备，也可以是自动切换或者手动切换，而且越来越多的新系统都开始具备自动选举切换“领导”的能力，以提升系统的可用性。

\*\*去中心化设计\*\*

- 众生地位平等： 在去中心化的设计里，通常没有“领导”和“干活的”这两种角色的区分，大家的角色都是一样的，地位是平等的，全球互联网就是一个典型的去中心化的分布式系统，联网的任意节点设备宕机，都只会影响很小范围的功能。

- “去中心化”不是不要中心，而是由节点来自由选择中心。 （集群的成员会自发的举行“会议”选举新的“领导”主持工作。最典型的案例就是ZooKeeper及Go语言实现的Etcd）

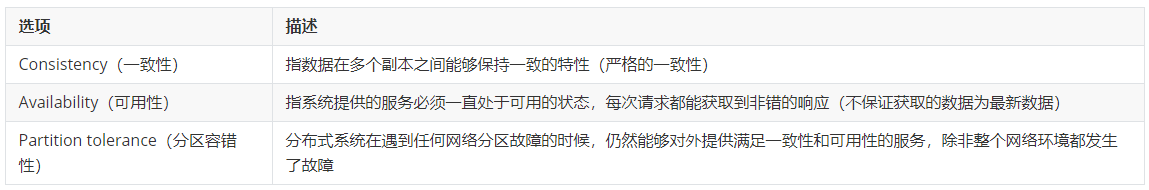
- 去中心化设计的问题： 去中心化设计里最难解决的一个问题是 “脑裂”问题 ，这种情况的发生概率很低，但影响很大。脑裂指一个集群由于网络的故障，被分为至少两个彼此无法通信的单独集群，此时如果两个集群都各自工作，则可能会产生严重的数据冲突和错误。一般的设计思路是，当集群判断发生了脑裂问题时，规模较小的集群就“自杀”或者拒绝服务。

### 分布式与集群的区别是什么？

- 分布式： 一个业务分拆多个子业务，部署在不同的服务器上

- 集群： 同一个业务，部署在多个服务器上。比如之前做电商网站搭的redis集群以及solr集群都是属于将redis服务器提供的缓存服务以及solr服务器提供的搜索服务部署在多个服务器上以提高系统性能、并发量解决海量存储问题

### CAP定理



Spring Cloud在CAP法则上主要满足的是A和P法则，Dubbo和Zookeeper在CAP法则主要满足的是C和P法则

# Mybatis

ORM框架：数据持久化 Object Relational Mapping 用于实现面向对象编程语言里不同类型系统的数据之间的转换。

JDBC链接数据库的问题：

1. 没有使用到连接池

2. sql写在代码中

3. 硬编码

## 技术本质

MyBatis的核心就在于配置文件解析和sql语句映射这一块，也是精髓所在，尤其是利用jdk的代理，达到crud的操作，堪称画龙点睛之笔。

- 数据库源 Driver、URL、username、password

- 执行语句 增删改查

- 操作 Connection、Parparestatement、ResultSet

一、获取数据库源：

#### SqlSessionFactoryBuilder().build(inputStream)

XMLConfigBuilder.parse() 调用 parseConfiguration(parser.evalNode("/configuration"));

XMLConfigBuilder.environmentsElement

Configuration.setEnvironment

二、获取数据库执行语句

mybatis加载mapper的方式有几种？

XMLConfigBuilder

Mapper

三、mybatis是如何操作的

mybatis的执行器有几种？Simple、Reuse、 Batch

获取数据库源

获取执行语句

通过jdbc执行。

反射到对象中返回结果集

## MyBatis的工作原理？

关于MyBatis的工作原理，JDBC有四个核心对象：

（1）DriverManager，用于注册数据库连接

（2）Connection，与数据库连接对象

（3）Statement/PrepareStatement，操作数据库SQL语句的对象

（4）ResultSet，结果集或一张虚拟表

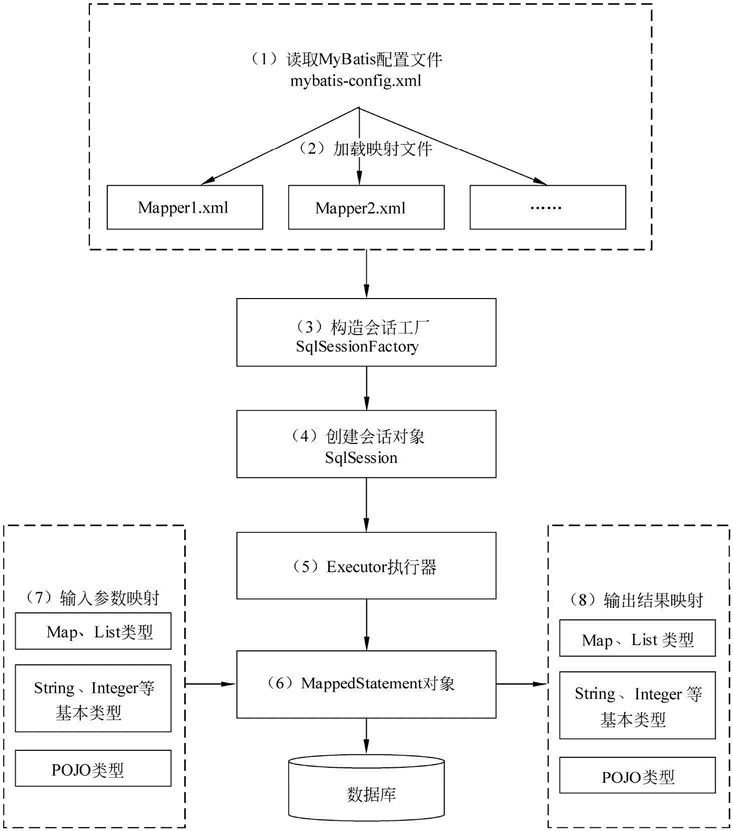
而MyBatis也有四大核心对象：

（1）SqlSession对象，该对象中包含了执行SQL语句的所有方法【1】。类似于JDBC里面的Connection 【2】。

（2）Executor接口，它将根据SqlSession传递的参数动态地生成需要执行的SQL语句，同时负责查询缓存的维护。类似于JDBC里面的Statement/PrepareStatement。

（3）MappedStatement对象，该对象是对映射SQL的封装，用于存储要映射的SQL语句的id、参数等信息。

（4）ResultHandler对象，用于对返回的结果进行处理，最终得到自己想要的数据格式或类型。可以自定义返回类型。



上面中流程就是MyBatis内部核心流程，每一步流程的详细说明如下文所述：

（1）读取MyBatis的配置文件。mybatis-config.xml为MyBatis的全局配置文件，用于配置数据库连接信息。

（2）加载映射文件。映射文件即SQL映射文件，该文件中配置了操作数据库的SQL语句，需要在MyBatis配置文件mybatis-config.xml中加载。mybatis-config.xml 文件可以加载多个映射文件，每个文件对应数据库中的一张表。

（3）构造会话工厂。通过MyBatis的环境配置信息构建会话工厂SqlSessionFactory。

（4）创建会话对象。由会话工厂创建SqlSession对象，该对象中包含了执行SQL语句的所有方法。

（5）Executor执行器。MyBatis底层定义了一个Executor接口来操作数据库，它将根据SqlSession传递的参数动态地生成需要执行的SQL语句，同时负责查询缓存的维护。

（6）MappedStatement对象。在Executor接口的执行方法中有一个MappedStatement类型的参数，该参数是对映射信息的封装，用于存储要映射的SQL语句的id、参数等信息。

（7）输入参数映射。输入参数类型可以是Map、List等集合类型，也可以是基本数据类型和POJO类型。输入参数映射过程类似于JDBC对preparedStatement对象设置参数的过程。

（8）输出结果映射。输出结果类型可以是Map、List等集合类型，也可以是基本数据类型和POJO类型。输出结果映射过程类似于JDBC对结果集的解析过程。

## mybatis 的核心组件：

- SqlSessionFactoryBuilder 它会根据配置或者代码来生成SqlSessionFactory，采用的是分步构建的Builder模式

- SqlSessionFactory 依靠它生成SqlSession，使用的是工厂模式。

- SqlSession ①可以发送SQL执行返回结构，②获取Mapper的接口。而我们现在一般会让其在业务代码中消失，使用Mapper接口编程

- SQL Mapper 接口和XML文件（或注解）构成，给出对应的SQL和映射规则，去执行并返回结果。



### SqlSessionFactory(工厂接口)

使用MyBatis首先就是去配置或者代码去生成SqlSessionFactory。

创建SqlSessionFactory,需要用到SqlSessionFactoryBuilder和一个类org.apache.itatis.session.Configuraton，采用Builder模式，具体分步在Configuration中完成。

### SqlSession

是MyBatis的核心接口，类似于JDBC中的Connection对象，代表着一个链接资源的启用。具体作用有三个：

- 获取Mapper接口

- 发送SQL给数据库

- 控制数据库事务

创建一个SqlSession：

SqlSession sqlSession = SqlSessionFactory.openSession();

注意：SqlSession是个门面接口，但实际上底层工作的是Executor

SqlSession事务：

sqlSession.commit();//提交事务

sqlSession.rollback();//回滚事务

sqlSession.close();

### 映射器

映射器是MyBatis中最重要、最复杂的组件，由一个接口和对应的XML文件（或注解组成）,他可以配置：

- 描述映射规则

- 提供SQL语句，并可以配置SQL参数类型、返回类型、缓存刷新等信息

- 配置缓存

- 提供动态SQL

\*\*用XML实现映射器与注解实现映射器\*\*

xml与pojo是自动映射的（如：role\_name <==> roleName）,只要列名一一对应即可。

与注解映射比较：

- XML和注解同时使用，XML会覆盖到注解的映射

- XML可读性高，动态SQL更容易使用

- XML可以相互引入，而注解不可以

### 其他类

#### Configuration

存储了MyBatis的所有配置项和解析后的信息，极为重要的一个类。MyBatis初始化时，将解析出的信息一股脑的扔给Configuration进行存储。执行sql语句或者其它操作时，从此类里取出相关的配置信息进行操作。

两个重要属性：

- MapperRegistry mapperRegistry; 映射注册机实例。

- Map<String, MappedStatement> mappedStatements，存储了 dao接口完全限定名+方法名 - MappedStatement实例的键值对。

#### XMLConfigBuilder

XML配置构建器，建造者模式。解析MyBatis的xml配置文件并以Document的形式保存下来，方便其内部解析方法解析出MyBatis的具体配置信息。

#### MapperRegistry

映射注册机，其属性 `Map<Class<?>, MapperProxyFactory<?>> knownMappers` 存储了 Class对象 - 映射器代理工厂实例 的键值对，MyBatis初始化时存入，使用时取出

#### MapperProxyFactory

映射器代理工厂，映射器代理类的工厂方法，生产一个dao接口的代理实例提供给用户。

#### MapperProxy

映射器代理类，利用jdk动态代理返回给用户代理类实例，除Object通用方法（toString()、hashcode()等等）、接口默认方法（java8 新增默认方法）外，皆走代理方法，在代理方法内部执行数据库的crud。

#### MappedStatement

映射语句，存储了解析注解、xml后的sql语句相关信息。

#### MapperMethod

映射器方法。负责sql语句参数填充及语句的分类执行。

两个极为重要的属性：

SqlCommand一个内部类 封装了SQL标签的类型 insert delete update select

MethodSignature一个内部类 封装了方法的参数信息 返回类型信息等

一个execute方法，负责执行sql语句，并将结果处理好后返回给用户。

# Spring相关

## Spring框架的优点

1.方便解耦，简化开发

通过Spring提供的IoC容器，我们可以将对象之间的依赖关系交由Spring进行控制，避免硬编码所造成的过度程序耦合。有了Spring，用户不必再为单实例模式类、属性文件解析等这些很底层的需求编写代码，可以更专注于上层的应用。

2.AOP编程的支持

通过Spring提供的AOP功能，方便进行面向切面的编程，许多不容易用传统OOP实现的功能可以通过AOP轻松应付。

3.声明事务的支持

在Spring中，我们可以从单调烦闷的事务管理代码中解脱出来，通过声明式方式灵活地进行事务的管理，提高开发效率和质量。 支持 POJO(Plain Old Java Object) 编程，从而具备持续集成和可测试性。

4.方便程序的测试

可以用非容器依赖的编程方式进行几乎所有的测试工作，在Spring里，测试不再是昂贵的操作，而是随手可做的事情。例如：Spring对Junit4支持，可以通过注解方便的测试Spring程序。

5.方便集成各种优秀框架

Spring不排斥各种优秀的开源框架，相反，Spring可以降低各种框架的使用难度，Spring提供了对各种优秀框架（如Struts,Hibernate、Hessian、Quartz）等的直接支持。它是开源免费的。

- \*\*轻量级\*\* - Spring 在代码量和透明度方面都很轻便。

- \*\*IOC\*\* - 控制反转

- \*\*AOP\*\* - 面向切面编程可以将应用业务逻辑和系统服务分离，以实现高内聚。

- \*\*容器\*\* - Spring 负责创建和管理对象（Bean）的生命周期和配置。

- \*\*MVC\*\* - 对 web 应用提供了高度可配置性，其他框架的集成也十分方便。

- \*\*事务管理\*\* - 提供了用于事务管理的通用抽象层。Spring 的事务支持也可用于容器较少的环境。

- \*\*JDBC 异常\*\* - Spring 的 JDBC 抽象层提供了一个异常层次结构，简化了错误处理策略。

缺点

Spring明明一个很轻量级的框架，却给人感觉大而全

Spring依赖反射，反射影响性能

使用门槛升高，入门Spring需要较长时间

## BeanFactory和ApplicationContext的关系详解

BeanFactory 简单粗暴，可以理解为就是个 HashMap，Key 是 BeanName，Value 是 Bean 实例。通常只提供注册（put），获取（get）这两个功能。我们可以称之为 “低级容器”。

ApplicationContext 可以称之为 “高级容器”。因为他比 BeanFactory 多了更多的功能。他继承了多个接口。因此具备了更多的功能。例如资源的获取，支持多种消息（例如 JSP tag 的支持），对 BeanFactory 多了工具级别的支持等待。所以你看他的名字，已经不是 BeanFactory 之类的工厂了，而是 “应用上下文”， 代表着整个大容器的所有功能。该接口定义了一个 refresh 方法，此方法是所有阅读 Spring 源码的人的最熟悉的方法，用于刷新整个容器，即重新加载/刷新所有的 bean。

## IOC和依赖注入（DI）

IoC（Inverse of Control:控制反转）是一种\*\*设计思想\*\*，就是 \*\*将原本在程序中手动创建对象的控制权，交由Spring框架来管理。\*\* IoC 在其他语言中也有应用，并非 Spirng 特有。 \*\*IoC 容器是 Spring 用来实现 IoC 的载体， IoC 容器实际上就是个Map（key，value）,Map 中存放的是各种对象。\*\*

将对象之间的相互依赖关系交给 IOC 容器来管理，并由 IOC 容器完成对象的注入。这样可以很大程度上简化应用的开发，把应用从复杂的依赖关系中解放出来。 \*\*IOC 容器就像是一个工厂一样，当我们需要创建一个对象的时候，只需要配置好配置文件/注解即可，完全不用考虑对象是如何被创建出来的。\*\* 在实际项目中一个 Service 类可能有几百甚至上千个类作为它的底层，假如我们需要实例化这个 Service，你可能要每次都要搞清这个 Service 所有底层类的构造函数，这可能会把人逼疯。如果利用 IOC 的话，你只需要配置好，然后在需要的地方引用就行了，这大大增加了项目的可维护性且降低了开发难度

在依赖注入中，您不必创建对象，但必须描述如何创建它们。您不是直接在代码中将组件和服务连接在一起，而是描述配置文件中哪些组件需要哪些服务。由 IoC 容器将它们装配在一起。

- 构造函数注入

- setter 注入

- 接口注入

创建对象和注入属性

实现两种方式：

1. XML文件方式，

在spring配置文件中使用bean标签

\- id属性，对象的标识，用于获取对象

\- class属性，类的全路径

默认执行无参方法

DI：依赖注入，就是注入属性，是IOC的一个实现，创建对象时使用

\- set方法注入

使用property标签注入，ref属性可以注入外部bean

<array><list><map><set>支持集合类型注入

### IoC 的一些好处是：

- 它将最小化应用程序中的代码量。

- 它将使您的应用程序易于测试，因为它不需要单元测试用例中的任何单例或 JNDI 查找机制。

- 它以最小的影响和最少的侵入机制促进松耦合。

- 它支持即时的实例化和延迟加载服务。

Spring 中的 IoC 的实现原理就是工厂模式加反射机制。

1、xml解析

2、工厂模式，降低耦合度，需要什么对象通过工厂的方法去创建，从而不需要和对象的类耦合

3、反射

## 面向切面的程序设计（AOP）：

AOP(Aspect-Oriented Programming:面向切面编程)能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码\*\*，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。

AOP：利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用行，同时提高开发效率

不通过修改源代码的方式，在主干功能里面添加新功能

底层使用动态代理方式

### 什么是 切面Aspect？

`aspect` 由 `pointcount` 和 `advice` 组成, 它既包含了横切逻辑的定义, 也包括了连接点的定义. Spring AOP 就是负责实施切面的框架, 它将切面所定义的横切逻辑编织到切面所指定的连接点中.

把通知应用到切入点的过程

AOP 的工作重心在于如何将增强编织目标对象的连接点上, 这里包含两个工作:

1. 如何通过 pointcut 和 advice 定位到特定的 joinpoint 上

2. 如何在 advice 中编写切面代码.

可以简单地认为, 使用 @Aspect 注解的类就是切面.

### 什么是切点（JoinPoint）

程序运行中的一些时间点, 例如一个方法的执行, 或者是一个异常的处理.

在 Spring AOP 中, join point 总是方法的执行点。

### 什么是通知（Advice）？

特定 JoinPoint 处的 Aspect 所采取的动作称为 Advice。Spring AOP 使用一个 Advice 作为拦截器，在 JoinPoint “周围”维护一系列的拦截器。

实际增强的逻辑部分被称为通知。

### 切入点

实际被真正增强的方法，成为切入点

execution（[权限修饰符][返回类型][类全路径][方法名称][参数列表]）

### 有哪些类型的通知（Advice）？

- \*\*Before\*\* - 这些类型的 Advice 在 joinpoint 方法之前执行，并使用 @Before 注解标记进行配置。

- \*\*After Returning\*\* - 这些类型的 Advice 在连接点方法正常执行后执行，并使用@AfterReturning 注解标记进行配置。

- \*\*After Throwing\*\* - 这些类型的 Advice 仅在 joinpoint 方法通过抛出异常退出并使用 @AfterThrowing 注解标记配置时执行。

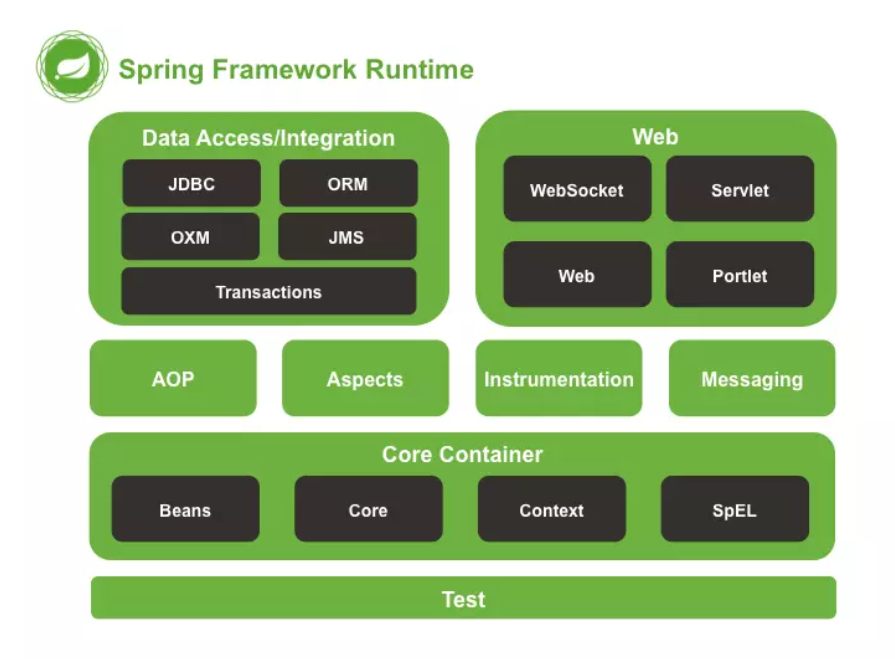
- \*\*After (finally)\*\* - 这些类型的 Advice 在连接点方法之后执行，无论方法退出是正常还是异常返回，并使用 @After 注解标记进行配置。

- \*\*Around\*\* - 这些类型的 Advice 在连接点之前和之后执行，并使用 @Around 注解标记进行配置。

相同的切入点可以进行抽取

@Order 设置切面的优先级，值越小越先执行。

## 核心组件



核心容器由\*\*spring-core，spring-beans，spring-context，spring-context-support和spring-expression\*\*（SpEL，Spring表达式语言，Spring Expression Language）等模块组成，它们的细节如下：

spring-core模块提供了框架的基本组成部分，包括 IoC 和依赖注入功能。

spring-beans 模块提供 BeanFactory，工厂模式的微妙实现，它移除了编码式单例的需要，并且可以把配置和依赖从实际编码逻辑中解耦。

context模块建立在由core和 beans 模块的基础上建立起来的，它以一种类似于JNDI注册的方式访问对象。Context模块继承自Bean模块，并且添加了国际化（比如，使用资源束）、事件传播、资源加载和透明地创建上下文（比如，通过Servelet容器）等功能。Context模块也支持Java EE的功能，比如EJB、JMX和远程调用等。\*\*ApplicationContext\*\*接口是Context模块的焦点。\*\*spring-context-support提供了对第三方库集成到Spring上下文的支持，比如缓存（EhCache, Guava, JCache）、邮件（JavaMail）、调度（CommonJ, Quartz）、模板引擎（FreeMarker, JasperReports, Velocity）等。

spring-expression模块提供了强大的表达式语言，用于在运行时查询和操作对象图。它是JSP2.1规范中定义的统一表达式语言的扩展，支持set和get属性值、属性赋值、方法调用、访问数组集合及索引的内容、逻辑算术运算、命名变量、通过名字从Spring IoC容器检索对象，还支持列表的投影、选择以及聚合等。

- 数据访问/集成

– 该层提供与数据库交互的支持。它包含以下模块：

- JDBC (Java DataBase Connectivity)

- ORM (Object Relational Mapping)用于支持Hibernate等ORM工具。

- OXM (Object XML Mappers)

- JMS (Java Messaging Service)Java消息服务。

- Transaction

- Web

– 该层提供了创建 Web 应用程序的支持。它包含以下模块：

- Web

- Web – Servlet

- Web – Socket

- Web – Portlet

- \*\*AOP\*\* – 该层支持面向切面编程

- \*\*Instrumentation\*\* – 该层为类检测和类加载器实现提供支持。

- \*\*Test\*\* – 该层为使用 JUnit 和 TestNG 进行测试提供支持。

- 几个杂项模块:

- Messaging – 该模块为 STOMP 提供支持。它还支持注解编程模型，该模型用于从 WebSocket 客户端路由和处理 STOMP 消息。

- Aspects – 该模块为与 AspectJ 的集成提供支持。

## Spring IoC 容器

IOC的思想基于IOC容器来实现。IOC容器的本质就是所有对象的工厂，需要什么对象就从工厂中取

### Spring提供了IOC容器的两种方式

（1）BeanFactory 是Spring内部使用的接口，一般不提供给开发人员，

\\* 加载配置文件的时候不会创建对象，获取对象的时候才会创建对象。

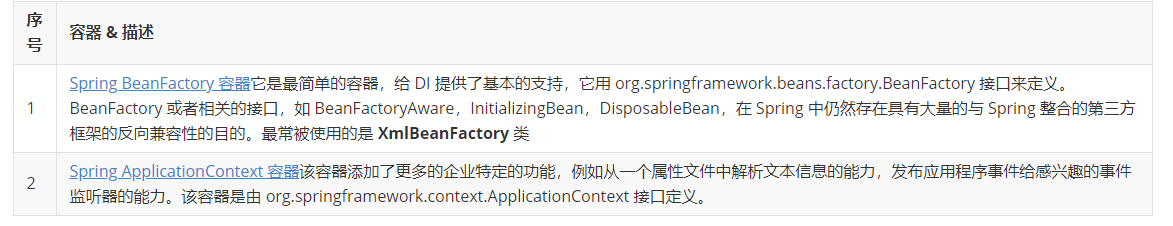
（2）ApplicationContext 是BeanFactory的一个子接口，提供更多更强大的功能，面向开发人员设计使用的

\\* 加载配置文件的时候就会把配置文件对象是进行创建

ApplicationContext的实现类：FileSystemXmlApplicationContext，ClassPathXmlApplicationContext

Spring 容器是 Spring 框架的核心。容器将创建对象，把它们连接在一起，配置它们，并管理他们的整个生命周期从创建到销毁。\*\*IOC 容器\*\*具有依赖注入功能的容器，它可以创建对象，IOC 容器负责实例化、定位、配置应用程序中的对象及建立这些对象间的依赖

Spring 框架的核心是 Spring 容器。容器创建对象，将它们装配在一起，配置它们并管理它们的完整生命周期。Spring 容器使用依赖注入来管理组成应用程序的组件。容器通过读取提供的配置元数据来接收对象进行实例化，配置和组装的指令。该元数据可以通过 XML，Java 注解或 Java 代码提供。



- \*\*FileSystemXmlApplicationContext\*\*：该容器从 XML 文件中加载已被定义的 bean。在这里，你需要提供给构造器 XML 文件的完整路径。

- \*\*ClassPathXmlApplicationContext\*\*：该容器从 XML 文件中加载已被定义的 bean。在这里，你不需要提供 XML 文件的完整路径，只需正确配置 CLASSPATH 环境变量即可，因为，容器会从 CLASSPATH 中搜索 bean 配置文件。

- \*\*WebXmlApplicationContext\*\*：该容器会在一个 web 应用程序的范围内加载在 XML 文件中已被定义的 bean。

### Spring Bean 作用域



### spring bean 容器的生命周期

spring bean 容器的生命周期流程如下：

1. Spring 容器根据配置中的 bean 定义中实例化 bean。

2. Spring 使用依赖注入填充所有属性，如 bean 中所定义的配置。

- 如果 bean 实现 BeanNameAware 接口，则工厂通过传递 bean 的 ID 来调用 setBeanName()。

- 如果 bean 实现 BeanFactoryAware 接口，工厂通过传递自身的实例来调用 setBeanFactory()。

3. 如果存在与 bean 关联的任何 BeanPostProcessors，则调用 preProcessBeforeInitialization() 方法。

4. 如果为 bean 指定了 init 方法（`<bean>` 的 init-method 属性），那么将调用它。

5. 最后，如果存在与 bean 关联的任何 BeanPostProcessors，则将调用 postProcessAfterInitialization() 方法。

6. bean的获取使用

​ - 如果 bean 实现 DisposableBean 接口，当 spring 容器关闭时，会调用 destory()。

7. 如果为 bean 指定了 destroy 方法（`<bean>` 的 destroy-method 属性），那么将调用它

1）、创建Bean的实例

2）、populateBean；给bean的各种属性赋值

3）、initializeBean：初始化bean；

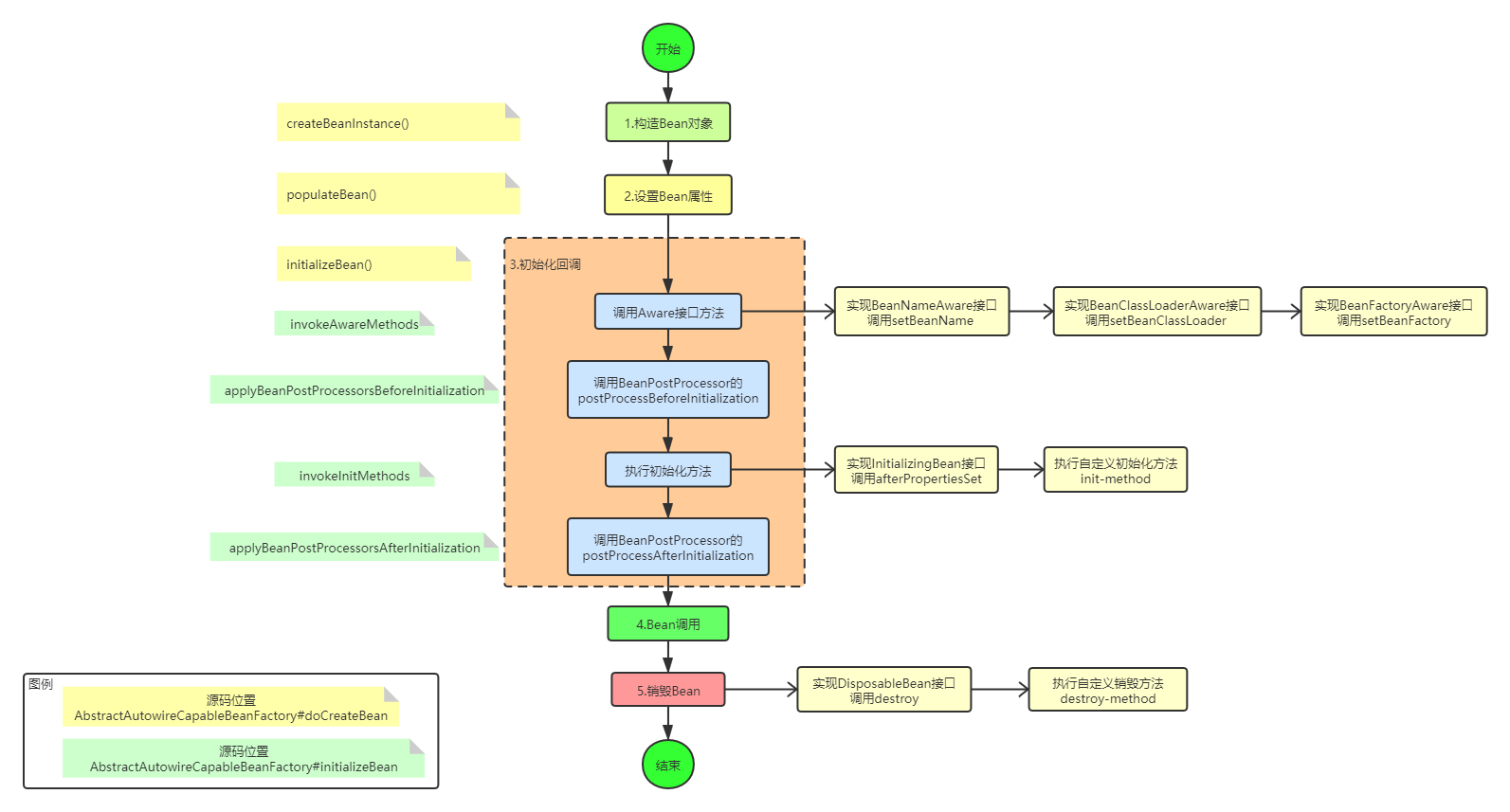
1）、invokeAwareMethods()：处理Aware接口的方法回调

2）、applyBeanPostProcessorsBeforeInitialization()：应用后置处理器的postProcessBeforeInitialization（）

3）、invokeInitMethods()；执行自定义的初始化方法

4）、applyBeanPostProcessorsAfterInitialization()；执行后置处理器的postProcessAfterInitialization（）；

4）、Bean创建成功



## spring事务

### Spring事务的配置方式

Spring支持编程式事务管理以及声明式事务管理两种方式。

是数据库操作最基本的单元，一组操作，要么都成功要么都失败：四个特性ACID

一般使用声明式事务管理。

底层使用的是AOP原理。

### 编程式事务管理

不推荐使用，编程式事务管理是侵入性事务管理，使用TransactionTemplate或者直接使用PlatformTransactionManager，对于编程式事务管理，Spring推荐使用TransactionTemplate。

### 声明式事务管理

必须标注在public方法上。

声明式事务管理建立在AOP之上，其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，执行完目标方法之后根据执行的情况提交或者回滚。

编程式事务每次实现都要单独实现，但业务量大功能复杂时，使用编程式事务无疑是痛苦的，而声明式事务不同，声明式事务属于无侵入式，不会影响业务逻辑的实现，只需要在配置文件中做相关的事务规则声明或者通过注解的方式，便可以将事务规则应用到业务逻辑中。

显然声明式事务管理要优于编程式事务管理，这正是Spring倡导的非侵入式的编程方式。唯一不足的地方就是声明式事务管理的粒度是方法级别，而编程式事务管理是可以到代码块的，但是可以通过提取方法的方式完成声明式事务管理的配置。

### 事务的传播机制

事务的传播性一般用在事务嵌套的场景，比如一个事务方法里面调用了另外一个事务方法，那么两个方法是各自作为独立的方法提交还是内层的事务合并到外层的事务一起提交，这就是需要事务传播机制的配置来确定怎么样执行。

常用的事务传播机制如下：

- PROPAGATION\_REQUIRED

Spring默认的传播机制，能满足绝大部分业务需求，如果外层有事务，则当前事务加入到外层事务，一块提交，一块回滚。如果外层没有事务，新建一个事务执行

- PROPAGATION\_REQUES\_NEW

该事务传播机制是每次都会新开启一个事务，同时把外层事务挂起，当当前事务执行完毕，恢复上层事务的执行。如果外层没有事务，执行当前新开启的事务即可

- PROPAGATION\_SUPPORT

如果外层有事务，则加入外层事务，如果外层没有事务，则直接使用非事务方式执行。完全依赖外层的事务

- PROPAGATION\_NOT\_SUPPORT

该传播机制不支持事务，如果外层存在事务则挂起，执行完当前代码，则恢复外层事务，无论是否异常都不会回滚当前的代码

- PROPAGATION\_NEVER

该传播机制不支持外层事务，即如果外层有事务就抛出异常

- PROPAGATION\_MANDATORY

与NEVER相反，如果外层没有事务，则抛出异常

- PROPAGATION\_NESTED

该传播机制的特点是可以保存状态保存点，当前事务回滚到某一个点，从而避免所有的嵌套事务都回滚，即各自回滚各自的，如果子事务没有把异常吃掉，基本还是会引起全部回滚的。

> 传播规则回答了这样一个问题：一个新的事务应该被启动还是被挂起，或者是一个方法是否应该在事务性上下文中运行。

### 事务的隔离级别

事务的隔离级别定义一个事务可能受其他并发务活动活动影响的程度，可以把事务的隔离级别想象为这个事务对于事物处理数据的自私程度。

在一个典型的应用程序中，多个事务同时运行，经常会为了完成他们的工作而操作同一个数据。并发虽然是必需的，但是会导致以下问题：

1. 脏读（Dirty read）

一个未提交事务读取到另一个未提交事务的数据：：：脏读发生在一个事务读取了被另一个事务改写但尚未提交的数据时。如果这些改变在稍后被回滚了，那么第一个事务读取的数据就会是无效的。

2. 不可重复读（Nonrepeatable read）

一个未提交事务读到了一个提交事务修改的数据：：：不可重复读发生在一个事务执行相同的查询两次或两次以上，但每次查询结果都不相同时。这通常是由于另一个并发事务在两次查询之间更新了数据。

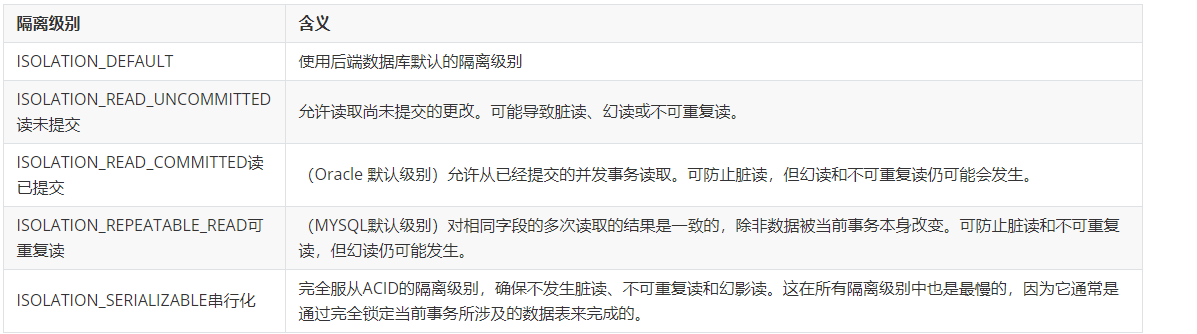
> 不可重复读重点在修改。

1. 幻读（Phantom reads）

一个未提交事务读到了一个提交事务的添加的数据：：：幻读和不可重复读相似。当一个事务（T1）读取几行记录后，另一个并发事务（T2）插入了一些记录时，幻读就发生了。在后来的查询中，第一个事务（T1）就会发现一些原来没有的额外记录。

> 幻读重点在新增或删除。

在理想状态下，事务之间将完全隔离，从而可以防止这些问题发生。然而，完全隔离会影响性能，因为隔离经常涉及到锁定在数据库中的记录（甚至有时是锁表）。完全隔离要求事务相互等待来完成工作，会阻碍并发。因此，可以根据业务场景选择不同的隔离级别。



### Spring声明式事务配置参考

事物配置中有哪些属性可以配置?以下只是简单的使用参考

1. 事务的传播性：

@Transactional(propagation=Propagation.REQUIRED)

2. 事务的隔离级别：

@Transactional(isolation = Isolation.READ\_UNCOMMITTED)

> 读取未提交数据(会出现脏读, 不可重复读) 基本不使用

1. 只读：

@Transactional(readOnly=true)

该属性用于设置当前事务是否为只读事务，设置为true表示只读，false则表示可读写，默认值为false。

2. 事务的超时性：

@Transactional(timeout=30)

3. 回滚：

指定单一异常类：@Transactional(rollbackFor=RuntimeException.class)

指定多个异常类：@Transactional(rollbackFor={RuntimeException.class, Exception.class})

### 源码

环境搭建：

1、导入相关依赖

数据源、数据库驱动、Spring-jdbc模块

2、配置数据源、JdbcTemplate（Spring提供的简化数据库操作的工具）操作数据

3、给方法上标注 @Transactional 表示当前方法是一个事务方法；

4、 @EnableTransactionManagement 开启基于注解的事务管理功能；

@EnableXXX

5、配置事务管理器来控制事务;

@Bean

public PlatformTransactionManager transactionManager()

原理：

1）、@EnableTransactionManagement

利用TransactionManagementConfigurationSelector给容器中会导入组件

导入两个组件

AutoProxyRegistrar

ProxyTransactionManagementConfiguration

2）、AutoProxyRegistrar：

给容器中注册一个 InfrastructureAdvisorAutoProxyCreator 组件；

InfrastructureAdvisorAutoProxyCreator：

利用后置处理器机制在对象创建以后，包装对象，返回一个代理对象（增强器），代理对象执行方法利用拦截器链进行调用；

3）、ProxyTransactionManagementConfiguration 做了什么？

1、给容器中注册事务增强器；

1）、事务增强器要用事务注解的信息，AnnotationTransactionAttributeSource解析事务注解

2）、事务拦截器：

TransactionInterceptor；保存了事务属性信息，事务管理器；

他是一个 MethodInterceptor；

在目标方法执行的时候；

执行拦截器链；

事务拦截器：

1）、先获取事务相关的属性

2）、再获取PlatformTransactionManager，如果事先没有添加指定任何transactionmanger

最终会从容器中按照类型获取一个PlatformTransactionManager；

3）、执行目标方法

如果异常，获取到事务管理器，利用事务管理回滚操作；

如果正常，利用事务管理器，提交事务

## 注解

### Component, Controller, Repository, Service 有何区别？

- @Component：这将 java 类标记为 bean。它是任何 Spring 管理组件的通用构造型。spring 的组件扫描机制现在可以将其拾取并将其拉入应用程序环境中。

- @Controller：这将一个类标记为 Spring Web MVC 控制器。标有它的 Bean 会自动导入到 IoC 容器中。

- @Service：此注解是组件注解的特化。它不会对 @Component 注解提供任何其他行为。您可以在服务层类中使用 @Service 而不是 @Component，因为它以更好的方式指定了意图。

- @Repository：这个注解是具有类似用途和功能的 @Component 注解的特化。它为 DAO 提供了额外的好处。它将 DAO 导入 IoC 容器，并使未经检查的异常有资格转换为 Spring DataAccessException。

- `@Component` ：通用的注解，可标注任意类为 `Spring` 组件。如果一个Bean不知道属于拿个层，可以使用`@Component` 注解标注。

`@Repository` : 对应持久层即 Dao 层，主要用于数据库相关操作。

`@Service` : 对应服务层，主要涉及一些复杂的逻辑，需要用到 Dao层。

`@Controller` : 对应 Spring MVC 控制层，主要用户接受用户请求并调用 Service 层返回数据给前端页面。

## Spring中的设计模式

工厂模式：BeanFactory就是简单工厂模式的体现，用来创建对象的实例；

单例模式：Bean默认为单例模式。

代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术；

模板方法：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。

观察者模式：定义对象键一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知被制动更新，如Spring中listener的实现–ApplicationListener。

# SpringMVC

Spring MVC 下我们一般把后端项目分为 Service层（处理业务）、Dao层（数据库操作）、Entity层（实体类）、Controller层(控制层，返回数据给前台页面)。

Spring Web MVC 框架提供 \*\*模型-视图-控制器\*\* 架构和随时可用的组件，用于开发灵活且松散耦合的 Web 应用程序。 MVC 模式有助于分离应用程序的不同方面，如输入逻辑，业务逻辑和 UI 逻辑，同时在所有这些元素之间提供松散耦合。

WebApplicationContext 是 ApplicationContext 的扩展。它具有 Web 应用程序所需的一些额外功能。它与普通的 ApplicationContext 在解析主题和决定与哪个 servlet 关联的能力方面有所不同。

## 五大核心组件

DispatcherServlet 请求的入口

HandlerMapping 请求的派发 负责让请求 和 控制器建立一一对应的关联

Controller（Handler） 真正的处理器

HandlerAdapter：处理器适配器

ModelAndView 封装模型信息和视图信息的

ViewResolver 视图处理器 最终定位页面的

### DispatcherServlet 中的九大核心属性

- MultipartResolver 内容类型( `Content-Type` )为 `multipart/\*` 的请求的解析器接口

- LocaleResolver \*\*本地化( 国际化 )解析器接口\*\*。

- ThemeResolver 主题解析器接口 因为现在的前端，基本和后端做了分离，所以这个功能已经越来越少用了。

- HandlerMapping 处理器匹配接口，\*\*根据请求( `handler` )获得其的处理器( `handler`)和拦截器们( HandlerInterceptor 数组 )。\*\*

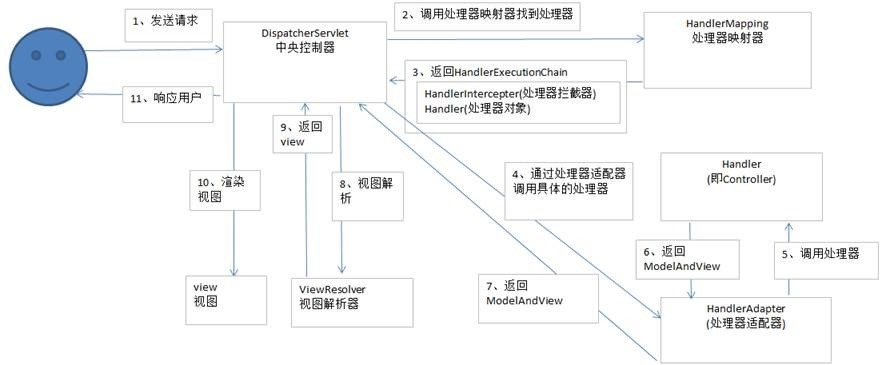
- HandlerAdapter 处理器适配器接口 因为，处理器 `handler` 的类型是 Object 类型，需要有一个调用者来实现 `handler` 是怎么被使用，怎么被执行。而 HandlerAdapter 的用途就在于此

- HandlerExceptionResolver \*\*处理器异常解析器接口，将处理器( `handler` )执行时发生的异常，解析( 转换 )成对应的 ModelAndView 结果。\*\*

- RequestToViewNameTranslator 请求到视图名的转换器接口

- ViewResolver \*\*实体解析器接口，根据视图名和国际化，获得最终的视图 View 对象\*\*

- FlashMapManager F\*\*lashMap 管理器接口，负责重定向时，保存参数到临时存储中\*\*。实际场景下，使用的非常少，特别是前后端分离之后



第一步：用户发起request请求，请求至DispatcherServlet前端控制器。

第二步：DispatcherServlet前端控制器请求HandlerMapping处理器映射器查找Handler。

DispatcherServlet：前端控制器，相当于中央调度器，各各组件都和前端控制器进行交互，降低了各组件之间耦合度。

第三步：HandlerMapping处理器映射器，根据url及一些配置规则（xml配置、注解配置）查找Handler，将Handler返回给 DispatcherServlet前端控制器。

第四步：DispatcherServlet前端控制器调用适配器执行Handler，有了适配器通过适配器去扩展对不同Handler执行方式（比如： 原始servlet开发，注解开发）

第五步：适配器执行Handler。Handler是后端控制器，当成模型。

第六步：Handler执行完成返回ModelAndView

ModelAndView是springmvc的一个对象，对Model和view进行封装。

第七步：适配器将ModelAndView返回给DispatcherServlet

第八步：DispatcherServlet调用视图解析器进行视图解析，解析后生成view。视图解析器根据逻辑视图名解析出真正的视图。

View：springmvc视图封装对象，提供了很多view，比如：jsp、freemarker、pdf、excel。

第九步：ViewResolver视图解析器给前端控制器返回view

第十步：DispatcherServlet调用view的渲染视图的方法，将模型数据填充到request域 。

第十一步：DispatcherServlet向用户响应结果(jsp页面、json数据。)

## SpringMVC 原理

使用javaConfig初始化项目：

实现WebApplicationInitializer接口，web容器启动的时候会调用onStartup方法。用java注解的方式去初始化spring上下文环境，AnnotationConfigWebApplicationContext ac=new AnnotationConfigWebApplicationContext（）；

ac.register(AppConfig.class);

DispatcherServlet servlet = new DispatcherServlet(ac);

spring帮我们实现了servlet的接口，把我们实现WebApplicationInitializer接口的类反射，并调用其onstartup方法并启动。

1、web容器在启动的时候，会扫描每个jar包下的META-INF/services/javax.servlet.ServletContainerInitializer

2、加载这个文件指定的类SpringServletContainerInitializer

3、spring的应用一启动会加载感兴趣的WebApplicationInitializer接口的下的所有组件；

4、并且为WebApplicationInitializer组件创建对象（组件不是接口，不是抽象类）

1）、AbstractContextLoaderInitializer：创建根容器；createRootApplicationContext()；

2）、AbstractDispatcherServletInitializer：

创建一个web的ioc容器；createServletApplicationContext();

创建了DispatcherServlet；createDispatcherServlet()；

将创建的DispatcherServlet添加到ServletContext中；

getServletMappings();

3）、AbstractAnnotationConfigDispatcherServletInitializer：注解方式配置的DispatcherServlet初始化器

创建根容器：createRootApplicationContext()

getRootConfigClasses();传入一个配置类

创建web的ioc容器： createServletApplicationContext();

获取配置类；getServletConfigClasses();

总结：

以注解方式来启动SpringMVC；继承AbstractAnnotationConfigDispatcherServletInitializer；

实现抽象方法指定DispatcherServlet的配置信息；

### SpringMVC源码分析

核心功能是请求分发

请求分发的核心类：DispatcherServlet 继承HTTPServlet

核心方法：doDispatch（HttpServletRequest request，HttpServletResponse response）处理分发逻辑。

HandlerMapping组件用于寻找获取到我们的Controller，实现类为RequestMappingHandlerMapping.getHandlerInternal

方法是getHandler(processedRequest) 获取到我们的Controller

HandlerAdapter 适配器，完成调用方法

HandlerMethodArgumentResolver 参数处理器，进行参数处理

1. supportsParameter判断你的参数是否需要有当前处理器来处理

2. resolveArgument 给这个参数赋值

除了加Controller注解来实现一个controller以外还有两种方式来实现

1. 实现HttpRequestHandler接口

2. 实现Controller接口

### 从发请求到Controller中间的过程

1. 扫描整个项目，定义一个map集合

2. 拿到所有加了Controller注解的类

3. 遍历类里面的所有方法对象

4. 判断方法是否加了RequestMapping注解

5. 把@RequestMapping注解的value作为map集合的Key给put进去，把method对象作为value放入map集合中

6. 根据用户请求拿到请求中的uri

7. 使用请求中的uri去map中匹配，匹配不到就404，匹配到了就得到了方法对象

## 相关注解

### @Controller

@Controller 注解，它将一个类标记为 Spring Web MVC 控制器 Controller 。

### @RestController 和 @Controller 的区别

@RestController 注解，在 @Controller 基础上，增加了 @ResponseBody 注解，更加适合目前前后端分离的架构下，提供 Restful API ，返回例如 JSON 数据格式。当然，返回什么样的数据格式，根据客户端的 "ACCEPT" 请求头来决定。

### @RequestMapping

@RequestMapping 注解，用于将特定 HTTP 请求方法映射到将处理相应请求的控制器中的特定类/方法。此注释可应用于两个级别：

类级别：映射请求的 URL。

方法级别：映射 URL 以及 HTTP 请求方法。

### ## @RequestMapping 和 @GetMapping 区别

@RequestMapping 可注解在类和方法上；@GetMapping 仅可注册在方法上。

@RequestMapping 可进行 GET、POST、PUT、DELETE 等请求方法；@GetMapping 是 @RequestMapping 的 GET 请求方法的特例，目的是为了提高清晰度。

### 返回 JSON 格式使用的注解：

可以使用 @ResponseBody 注解，或者使用包含 @ResponseBody 注解的 @RestController 注解。

当然，还是需要配合相应的支持 JSON 格式化的 HttpMessageConverter 实现类。例如，Spring MVC 默认使用 MappingJackson2HttpMessageConverter 。

### @PathVariable 注解与@RequestParam的区别

@PathVariable 注解，是 Spring MVC 中有用的注解之一，它允许您从URI 读取值，比如查询参数。它在使用 Spring 创建 RESTful Web 服务时特别有用，因为在 REST 中，资源标识符是 URI 的一部分。

@RequestParam注解和@PathVariable注解的区别，从字面上可以看出前者是获取请求里边携带的参数；后者是获取请求路径里边的变量参数。

(例如：127.0.0.1/user/{userId}?userName=zhangshan,userId是路径上的变量，userName才是请求参数信息)

1.@RequestParam注解

@RequestParam有三个参数：

value：参数名；

required：是否必需，默认为true，表示请求参数中必须包含该参数，如果不包含抛出异常。

defaultValue：默认参数值，如果设置了该值自动将required设置为false，如果参数中没有包含该参数则使用默认值。

示例：@RequestParam(value = "userId", required = false, defaultValue = "1")

2.@PathVariable注解

当使用@RequestMapping URI占位符映射时，Url中可以通过一个或多个{xxxx}占位符映射，通过@PathVariable可以绑定占位符参数到方法参数中。

例如：@PathVariable("userId") Long userId,@PathVariable("userName") String userName

(注：Long类型可以根据需求自己改变String或int，Spring会自动做转换)

@RequestMapping(“/user/{userId}/{userName}/query")

### @RequestBody

@requestBody注解常用来处理content-type不是默认的application/x-www-form-urlcoded编码的内容，比如说：application/json或者是application/xml等。一般情况下来说常用其来处理application/json类型。

通过@requestBody可以将请求体中的JSON字符串绑定到相应的bean上，当然，也可以将其分别绑定到对应的字符串上。

前端通过post传数据过来,要用RequestBody注解。

## SpringWebflux

\*\*Spring WebFlux 是一个异步非阻塞式的 Web 框架，它能够充分利用多核 CPU 的硬件资源去处理大量的并发请求。\*\*

WebFlux 内部使用的是响应式编程（Reactive Programming），以 Reactor 库为基础, 基于异步和事件驱动，可以让我们在不扩充硬件资源的前提下，提升系统的吞吐量和伸缩性。

\*\*WebFlux 并不能使接口的响应时间缩短，它仅仅能够提升吞吐量和伸缩性\*\*。

上面说到了， Spring WebFlux 是一个异步非阻塞式的 Web 框架，所以，它特别适合应用在 IO 密集型的服务中，比如微服务网关这样的应用中。

PS: IO 密集型包括：\*\*磁盘IO密集型\*\*, \*\*网络IO密集型\*\*，微服务网关就属于网络 IO 密集型，使用异步非阻塞式编程模型，能够显著地提升网关对下游服务转发的吞吐量。

\*\*在合适的场景中，选型最合适的技术\*\*。

- Spring MVC 因为是使用的同步阻塞式，更方便开发人员编写功能代码，Debug 测试等，一般来说，如果 Spring MVC 能够满足的场景，就尽量不要用 WebFlux;

- WebFlux 默认情况下使用 Netty 作为服务器;

- WebFlux 不支持 MySql;

1、新增的模块，用于web开发的，类似于SpringMVC，Webflux使用比较流行的响应式编程。

2、传统web框架，SpringMvc，这些事基于servlet容器，Webflux是一种异步非阻塞的框架，一部非阻塞在Servlet3.1以后才支持，核心是基于Reactor的相关API实现的

3、同步和异步，调用者发送请求等待对方的回应才去做其他事情就是同步

阻塞和非阻塞 针对被调用者，被调用者收到请求后先反馈在做事情就是非阻塞。

4、Webflux特点：异步非阻塞在有限的资源下，提高系统的吞吐量和伸缩性，实现响应式编程

使用java8的函数式编程实现路由请求

5、和SpringMvc的比较

1、都可以使用注解方式，都运行在Tomcat等容器中。

2、SpringMvc使用命令式编程

### 响应式编程和实现方式

面向数据流和变化传播的编程范式，自动将变化的值通过数据流进行传播

Java8：提供的观察者模式，Observer和Observable，响应式编程是观察者模式的一种

Reactor实现：

两个核心类Mono和Flux，这两个类都实现接口Publisher，提供丰富的操作符，Flux对象实现发布者，返回N个元素；Mono实现发布者，返回0个活1个元素，都可以发出三种数据信号，元素值，错误信号，完成信号

错误和完成信号不能共存

需要订阅才能出发数据流

操作符：map和flatMap

### Webflux的执行流程和核心API

1、基于Reactor，默认使用容器是Netty，Netty是高性能的NIO框架，异步非阻塞的框架。

2、执行过程与MVC相似。Webflux核心控制器是DispatchHandler，实现接口WebHandler，负责请求的处理

- HandlerMapping 请求查询到处理的方法

- HandlerAdapter 真正负责请求处理

- HandlerReultHandler 响应结果处理

实现函数式编程的两个接口

- RouterFunction 路由处理

- HandlerFunction 处理函数

两种实现方式

1、使用注解编程模式方式，和之前的SpringMVC使用相似的，只需要把相关依赖配置导入项目中，SpringBoot自动配置相关容器，默认情况下使用Netty服务器。

SpringMVC方式：同步阻塞的方式，基于SpringMVC+Servlet+Tomcat

SpringWebflux方式实现，异步非阻塞的方式，基于SpringWebflux+Reactor+Netty

2、函数式编程模型：

需要自己初始化服务器

核心任务是定义两个函数式接口的实现并启动需要的服务器（RouterFunction和HandlerFunction）

SpringWebflux请求和响应不在是ServletRequest和ServletResponse，而是ServerRequest和ServerResponse

# SpringBoot

## 概述

### 什么是 Spring Boot？

Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器，开发者能快速上手。

### Spring Boot 有哪些优点？

Spring Boot 主要有如下优点：

容易上手，提升开发效率，为 Spring 开发提供一个更快、更广泛的入门体验。

开箱即用，远离繁琐的配置。

提供了一系列大型项目通用的非业务性功能，例如：内嵌服务器、安全管理、运行数据监控、运行状况检查和外部化配置等。

没有代码生成，也不需要XML配置。

避免大量的 Maven 导入和各种版本冲突。

### Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的？

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下 3 个注解：

@SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。

@EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。

@ComponentScan：Spring组件扫描。

## 配置

### 什么是 JavaConfig？

Spring JavaConfig 是 Spring 社区的产品，它提供了配置 Spring IoC 容器的纯Java 方法。因此它有助于避免使用 XML 配置。使用 JavaConfig 的优点在于：

（1）面向对象的配置。由于配置被定义为 JavaConfig 中的类，因此用户可以充分利用 Java 中的面向对象功能。一个配置类可以继承另一个，重写它的@Bean 方法等。

（2）减少或消除 XML 配置。基于依赖注入原则的外化配置的好处已被证明。但是，许多开发人员不希望在 XML 和 Java 之间来回切换。JavaConfig 为开发人员提供了一种纯 Java 方法来配置与 XML 配置概念相似的 Spring 容器。从技术角度来讲，只使用 JavaConfig 配置类来配置容器是可行的，但实际上很多人认为将JavaConfig 与 XML 混合匹配是理想的。

（3）类型安全和重构友好。JavaConfig 提供了一种类型安全的方法来配置 Spring容器。由于 Java 5.0 对泛型的支持，现在可以按类型而不是按名称检索 bean，不需要任何强制转换或基于字符串的查找。

### Spring Boot 自动配置原理是什么？

注解 @EnableAutoConfiguration, @Configuration, @ConditionalOnClass 就是自动配置的核心，

@EnableAutoConfiguration 给容器导入META-INF/spring.factories 里定义的自动配置类。

筛选有效的自动配置类。

每一个自动配置类结合对应的 xxxProperties.java 读取配置文件进行自动配置功能

### 你如何理解 Spring Boot 配置加载顺序？

在 Spring Boot 里面，可以使用以下几种方式来加载配置。

1）properties文件；

2）YAML文件；

3）系统环境变量；

4）命令行参数；

等等……

### 什么是 YAML？

YAML 是一种人类可读的数据序列化语言。它通常用于配置文件。与属性文件相比，如果我们想要在配置文件中添加复杂的属性，YAML 文件就更加结构化，而且更少混淆。可以看出 YAML 具有分层配置数据。

### YAML 配置的优势在哪里 ?

YAML 现在可以算是非常流行的一种配置文件格式了，无论是前端还是后端，都可以见到 YAML 配置。那么 YAML 配置和传统的 properties 配置相比到底有哪些优势呢？

配置有序，在一些特殊的场景下，配置有序很关键

支持数组，数组中的元素可以是基本数据类型也可以是对象

简洁

相比 properties 配置文件，YAML 还有一个缺点，就是不支持 @PropertySource 注解导入自定义的 YAML 配置。

### Spring Boot 是否可以使用 XML 配置 ?

Spring Boot 推荐使用 Java 配置而非 XML 配置，但是 Spring Boot 中也可以使用 XML 配置，通过 @ImportResource 注解可以引入一个 XML 配置。

### spring boot 核心配置文件是什么？bootstrap.properties 和 application.properties 有何区别 ?

单纯做 Spring Boot 开发，可能不太容易遇到 bootstrap.properties 配置文件，但是在结合 Spring Cloud 时，这个配置就会经常遇到了，特别是在需要加载一些远程配置文件的时侯。

spring boot 核心的两个配置文件：

bootstrap (. yml 或者 . properties)：boostrap 由父 ApplicationContext 加载的，比 applicaton 优先加载，配置在应用程序上下文的引导阶段生效。一般来说我们在 Spring Cloud Config 或者 Nacos 中会用到它。且 boostrap 里面的属性不能被覆盖；

application (. yml 或者 . properties)： 由ApplicatonContext 加载，用于 spring boot 项目的自动化配置。

### 什么是 Spring Profiles？

Spring Profiles 允许用户根据配置文件（dev，test，prod 等）来注册 bean。因此，当应用程序在开发中运行时，只有某些 bean 可以加载，而在 PRODUCTION中，某些其他 bean 可以加载。假设我们的要求是 Swagger 文档仅适用于 QA 环境，并且禁用所有其他文档。这可以使用配置文件来完成。Spring Boot 使得使用配置文件非常简单。

### 如何在自定义端口上运行 Spring Boot 应用程序？

为了在自定义端口上运行 Spring Boot 应用程序，您可以在application.properties 中指定端口。server.port = 8090

## 安全

### 如何实现 Spring Boot 应用程序的安全性？

为了实现 Spring Boot 的安全性，我们使用 spring-boot-starter-security 依赖项，并且必须添加安全配置。它只需要很少的代码。配置类将必须扩展WebSecurityConfigurerAdapter 并覆盖其方法。

### 比较一下 Spring Security 和 Shiro 各自的优缺点 ?

由于 Spring Boot 官方提供了大量的非常方便的开箱即用的 Starter ，包括 Spring Security 的 Starter ，使得在 Spring Boot 中使用 Spring Security 变得更加容易，甚至只需要添加一个依赖就可以保护所有的接口，所以，如果是 Spring Boot 项目，一般选择 Spring Security 。当然这只是一个建议的组合，单纯从技术上来说，无论怎么组合，都是没有问题的。Shiro 和 Spring Security 相比，主要有如下一些特点：

Spring Security 是一个重量级的安全管理框架；Shiro 则是一个轻量级的安全管理框架

Spring Security 概念复杂，配置繁琐；Shiro 概念简单、配置简单

Spring Security 功能强大；Shiro 功能简单

Spring Boot 中如何解决跨域问题 ?

跨域可以在前端通过 JSONP 来解决，但是 JSONP 只可以发送 GET 请求，无法发送其他类型的请求，在 RESTful 风格的应用中，就显得非常鸡肋，因此我们推荐在后端通过 （CORS，Cross-origin resource sharing） 来解决跨域问题。这种解决方案并非 Spring Boot 特有的，在传统的 SSM 框架中，就可以通过 CORS 来解决跨域问题，只不过之前我们是在 XML 文件中配置 CORS ，现在可以通过实现WebMvcConfigurer接口然后重写addCorsMappings方法解决跨域问题。

@Configuration

public class CorsConfig implements WebMvcConfigurer {

@Override

public void addCorsMappings(CorsRegistry registry) {

registry.addMapping("/\*\*")

.allowedOrigins("\*")

.allowCredentials(true)

.allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")

.maxAge(3600);

}

}

项目中前后端分离部署，所以需要解决跨域的问题。

我们使用cookie存放用户登录的信息，在spring拦截器进行权限控制，当权限不符合时，直接返回给用户固定的json结果。

当用户登录以后，正常使用；当用户退出登录状态时或者token过期时，由于拦截器和跨域的顺序有问题，出现了跨域的现象。

我们知道一个http请求，先走filter，到达servlet后才进行拦截器的处理，如果我们把cors放在filter里，就可以优先于权限拦截器执行。

@Configuration

public class CorsConfig {

@Bean

public CorsFilter corsFilter() {

CorsConfiguration corsConfiguration = new CorsConfiguration();

corsConfiguration.addAllowedOrigin("\*");

corsConfiguration.addAllowedHeader("\*");

corsConfiguration.addAllowedMethod("\*");

corsConfiguration.setAllowCredentials(true);

UrlBasedCorsConfigurationSource urlBasedCorsConfigurationSource = new UrlBasedCorsConfigurationSource();

urlBasedCorsConfigurationSource.registerCorsConfiguration("/\*\*", corsConfiguration);

return new CorsFilter(urlBasedCorsConfigurationSource);

}

}

### 什么是 CSRF 攻击？

CSRF 代表跨站请求伪造。这是一种攻击，迫使最终用户在当前通过身份验证的Web 应用程序上执行不需要的操作。CSRF 攻击专门针对状态改变请求，而不是数据窃取，因为攻击者无法查看对伪造请求的响应。

## 监视器

### Spring Boot 中的监视器是什么？

Spring boot actuator 是 spring 启动框架中的重要功能之一。Spring boot 监视器可帮助您访问生产环境中正在运行的应用程序的当前状态。有几个指标必须在生产环境中进行检查和监控。即使一些外部应用程序可能正在使用这些服务来向相关人员触发警报消息。监视器模块公开了一组可直接作为 HTTP URL 访问的REST 端点来检查状态。

### 如何在 Spring Boot 中禁用 Actuator 端点安全性？

默认情况下，所有敏感的 HTTP 端点都是安全的，只有具有 ACTUATOR 角色的用户才能访问它们。安全性是使用标准的 HttpServletRequest.isUserInRole 方法实施的。 我们可以使用来禁用安全性。只有在执行机构端点在防火墙后访问时，才建议禁用安全性。

### 我们如何监视所有 Spring Boot 微服务？

Spring Boot 提供监视器端点以监控各个微服务的度量。这些端点对于获取有关应用程序的信息（如它们是否已启动）以及它们的组件（如数据库等）是否正常运行很有帮助。但是，使用监视器的一个主要缺点或困难是，我们必须单独打开应用程序的知识点以了解其状态或健康状况。想象一下涉及 50 个应用程序的微服务，管理员将不得不击中所有 50 个应用程序的执行终端。为了帮助我们处理这种情况，我们将使用位于的开源项目。 它建立在 Spring Boot Actuator 之上，它提供了一个 Web UI，使我们能够可视化多个应用程序的度量。

## 整合第三方项目

### 什么是 WebSockets？

WebSocket 是一种计算机通信协议，通过单个 TCP 连接提供全双工通信信道。

1、WebSocket 是双向的 -使用 WebSocket 客户端或服务器可以发起消息发送。

2、WebSocket 是全双工的 -客户端和服务器通信是相互独立的。

3、单个 TCP 连接 -初始连接使用 HTTP，然后将此连接升级到基于套接字的连接。然后这个单一连接用于所有未来的通信

4、Light -与 http 相比，WebSocket 消息数据交换要轻得多。

### 什么是 Spring Data ?

Spring Data 是 Spring 的一个子项目。用于简化数据库访问，支持NoSQL 和 关系数据存储。其主要目标是使数据库的访问变得方便快捷。Spring Data 具有如下特点：

SpringData 项目支持 NoSQL 存储：

MongoDB （文档数据库）

Neo4j（图形数据库）

Redis（键/值存储）

Hbase（列族数据库）

SpringData 项目所支持的关系数据存储技术：

JDBC

JPA

Spring Data Jpa 致力于减少数据访问层 (DAO) 的开发量. 开发者唯一要做的，就是声明持久层的接口，其他都交给 Spring Data JPA 来帮你完成！Spring Data JPA 通过规范方法的名字，根据符合规范的名字来确定方法需要实现什么样的逻辑。

### 什么是 Spring Batch？

Spring Boot Batch 提供可重用的函数，这些函数在处理大量记录时非常重要，包括日志/跟踪，事务管理，作业处理统计信息，作业重新启动，跳过和资源管理。它还提供了更先进的技术服务和功能，通过优化和分区技术，可以实现极高批量和高性能批处理作业。简单以及复杂的大批量批处理作业可以高度可扩展的方式利用框架处理重要大量的信息。

### 什么是 FreeMarker 模板？

FreeMarker 是一个基于 Java 的模板引擎，最初专注于使用 MVC 软件架构进行动态网页生成。使用 Freemarker 的主要优点是表示层和业务层的完全分离。程序员可以处理应用程序代码，而设计人员可以处理 html 页面设计。最后使用freemarker 可以将这些结合起来，给出最终的输出页面。

### 如何集成 Spring Boot 和 ActiveMQ？

对于集成 Spring Boot 和 ActiveMQ，我们使用依赖关系。 它只需要很少的配置，并且不需要样板代码。

### 什么是 Apache Kafka？

Apache Kafka 是一个分布式发布 - 订阅消息系统。它是一个可扩展的，容错的发布 - 订阅消息系统，它使我们能够构建分布式应用程序。这是一个 Apache 顶级项目。Kafka 适合离线和在线消息消费。

### 什么是 Swagger？你用 Spring Boot 实现了它吗？

Swagger 广泛用于可视化 API，使用 Swagger UI 为前端开发人员提供在线沙箱。Swagger 是用于生成 RESTful Web 服务的可视化表示的工具，规范和完整框架实现。它使文档能够以与服务器相同的速度更新。当通过 Swagger 正确定义时，消费者可以使用最少量的实现逻辑来理解远程服务并与其进行交互。因此，Swagger消除了调用服务时的猜测。

### 前后端分离，如何维护接口文档 ?

前后端分离开发日益流行，大部分情况下，我们都是通过 Spring Boot 做前后端分离开发，前后端分离一定会有接口文档，不然会前后端会深深陷入到扯皮中。一个比较笨的方法就是使用 word 或者 md 来维护接口文档，但是效率太低，接口一变，所有人手上的文档都得变。在 Spring Boot 中，这个问题常见的解决方案是 Swagger ，使用 Swagger 我们可以快速生成一个接口文档网站，接口一旦发生变化，文档就会自动更新，所有开发工程师访问这一个在线网站就可以获取到最新的接口文档，非常方便。

其他

如何重新加载 Spring Boot 上的更改，而无需重新启动服务器？Spring Boot项目如何热部署？

这可以使用 DEV 工具来实现。通过这种依赖关系，您可以节省任何更改，嵌入式tomcat 将重新启动。Spring Boot 有一个开发工具（DevTools）模块，它有助于提高开发人员的生产力。Java 开发人员面临的一个主要挑战是将文件更改自动部署到服务器并自动重启服务器。开发人员可以重新加载 Spring Boot 上的更改，而无需重新启动服务器。这将消除每次手动部署更改的需要。Spring Boot 在发布它的第一个版本时没有这个功能。这是开发人员最需要的功能。DevTools 模块完全满足开发人员的需求。该模块将在生产环境中被禁用。它还提供 H2 数据库控制台以更好地测试应用程序。

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

### Spring Boot 中的 starter 到底是什么 ?

首先，这个 Starter 并非什么新的技术点，基本上还是基于 Spring 已有功能来实现的。首先它提供了一个自动化配置类，一般命名为 XXXAutoConfiguration ，在这个配置类中通过条件注解来决定一个配置是否生效（条件注解就是 Spring 中原本就有的），然后它还会提供一系列的默认配置，也允许开发者根据实际情况自定义相关配置，然后通过类型安全的属性注入将这些配置属性注入进来，新注入的属性会代替掉默认属性。正因为如此，很多第三方框架，我们只需要引入依赖就可以直接使用了。当然，开发者也可以自定义 Starter

### spring-boot-starter-parent 有什么用 ?

我们都知道，新创建一个 Spring Boot 项目，默认都是有 parent 的，这个 parent 就是 spring-boot-starter-parent ，spring-boot-starter-parent 主要有如下作用：

定义了 Java 编译版本为 1.8 。

使用 UTF-8 格式编码。

继承自 spring-boot-dependencies，这个里边定义了依赖的版本，也正是因为继承了这个依赖，所以我们在写依赖时才不需要写版本号。

执行打包操作的配置。

自动化的资源过滤。

自动化的插件配置。

针对 application.properties 和 application.yml 的资源过滤，包括通过 profile 定义的不同环境的配置文件，例如 application-dev.properties 和 application-dev.yml。

Spring Boot 打成的 jar 和普通的 jar 有什么区别 ?

Spring Boot 项目最终打包成的 jar 是可执行 jar ，这种 jar 可以直接通过 java -jar xxx.jar 命令来运行，这种 jar 不可以作为普通的 jar 被其他项目依赖，即使依赖了也无法使用其中的类。

Spring Boot 的 jar 无法被其他项目依赖，主要还是他和普通 jar 的结构不同。普通的 jar 包，解压后直接就是包名，包里就是我们的代码，而 Spring Boot 打包成的可执行 jar 解压后，在 \BOOT-INF\classes 目录下才是我们的代码，因此无法被直接引用。如果非要引用，可以在 pom.xml 文件中增加配置，将 Spring Boot 项目打包成两个 jar ，一个可执行，一个可引用。

### 开启 Spring Boot 特性有哪几种方式？

1）继承spring-boot-starter-parent项目

2）导入spring-boot-dependencies项目依赖

### 如何使用 Spring Boot 实现异常处理？

Spring 提供了一种使用 ControllerAdvice 处理异常的非常有用的方法。 我们通过实现一个 ControlerAdvice 类，来处理控制器类抛出的所有异常。

如何使用 Spring Boot 实现分页和排序？

使用 Spring Boot 实现分页非常简单。使用 Spring Data-JPA 可以实现将可分页的传递给存储库方法。

### 微服务中如何实现 session 共享 ?

在微服务中，一个完整的项目被拆分成多个不相同的独立的服务，各个服务独立部署在不同的服务器上，各自的 session 被从物理空间上隔离开了，但是经常，我们需要在不同微服务之间共享 session ，常见的方案就是 Spring Session + Redis 来实现 session 共享。将所有微服务的 session 统一保存在 Redis 上，当各个微服务对 session 有相关的读写操作时，都去操作 Redis 上的 session 。这样就实现了 session 共享，Spring Session 基于 Spring 中的代理过滤器实现，使得 session 的同步操作对开发人员而言是透明的，非常简便。

### Spring Boot 中如何实现定时任务 ?

定时任务也是一个常见的需求，Spring Boot 中对于定时任务的支持主要还是来自 Spring 框架。

在 Spring Boot 中使用定时任务主要有两种不同的方式，一个就是使用 Spring 中的 @Scheduled 注解，另一个则是使用第三方框架 Quartz。

使用 Spring 中的 @Scheduled 的方式主要通过 @Scheduled 注解来实现。

使用 Quartz ，则按照 Quartz 的方式，定义 Job 和 Trigger 即可。

# SpringCloud

## 为什么需要学习Spring Cloud

不论是商业应用还是用户应用，在业务初期都很简单，我们通常会把它实现为单体结构的应用。但是，随着业务逐渐发展，产品思想会变得越来越复杂，单体结构的应用也会越来越复杂。这就会给应用带来如下的几个问题：

代码结构混乱：业务复杂，导致代码量很大，管理会越来越困难。同时，这也会给业务的快速迭代带来巨大挑战；

开发效率变低：开发人员同时开发一套代码，很难避免代码冲突。开发过程会伴随着不断解决冲突的过程，这会严重的影响开发效率；

排查解决问题成本高：线上业务发现 bug，修复 bug 的过程可能很简单。但是，由于只有一套代码，需要重新编译、打包、上线，成本很高。

由于单体结构的应用随着系统复杂度的增高，会暴露出各种各样的问题。近些年来，微服务架构逐渐取代了单体架构，且这种趋势将会越来越流行。Spring Cloud是目前最常用的微服务开发框架，已经在企业级开发中大量的应用。

## 什么是Spring Cloud

Spring Cloud是一系列框架的有序集合。它利用Spring Boot的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、智能路由、消息总线、负载均衡、断路器、数据监控等，都可以用Spring Boot的开发风格做到一键启动和部署。Spring Cloud并没有重复制造轮子，它只是将各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过Spring Boot风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

## 设计目标与优缺点

设计目标

协调各个微服务，简化分布式系统开发。

优缺点

微服务的框架那么多比如：dubbo、Kubernetes，为什么就要使用Spring Cloud的呢？

优点：

产出于Spring大家族，Spring在企业级开发框架中无人能敌，来头很大，可以保证后续的更新、完善

组件丰富，功能齐全。Spring Cloud 为微服务架构提供了非常完整的支持。例如、配置管理、服务发现、断路器、微服务网关等；

Spring Cloud 社区活跃度很高，教程很丰富，遇到问题很容易找到解决方案

服务拆分粒度更细，耦合度比较低，有利于资源重复利用，有利于提高开发效率

可以更精准的制定优化服务方案，提高系统的可维护性

减轻团队的成本，可以并行开发，不用关注其他人怎么开发，先关注自己的开发

微服务可以是跨平台的，可以用任何一种语言开发

适于互联网时代，产品迭代周期更短

缺点：

微服务过多，治理成本高，不利于维护系统

分布式系统开发的成本高（容错，分布式事务等）对团队挑战大

总的来说优点大过于缺点，目前看来Spring Cloud是一套非常完善的分布式框架，目前很多企业开始用微服务、Spring Cloud的优势是显而易见的。因此对于想研究微服务架构的同学来说，学习Spring Cloud是一个不错的选择。

## Spring Cloud发展前景

Spring Cloud对于中小型互联网公司来说是一种福音，因为这类公司往往没有实力或者没有足够的资金投入去开发自己的分布式系统基础设施，使用Spring Cloud一站式解决方案能在从容应对业务发展的同时大大减少开发成本。同时，随着近几年微服务架构和Docker容器概念的火爆，也会让Spring Cloud在未来越来越“云”化的软件开发风格中立有一席之地，尤其是在五花八门的分布式解决方案中提供了标准化的、全站式的技术方案，意义可能会堪比当年Servlet规范的诞生，有效推进服务端软件系统技术水平的进步。

整体架构

## 主要项目

Spring Cloud的子项目，大致可分成两类，一类是对现有成熟框架"Spring Boot化"的封装和抽象，也是数量最多的项目；第二类是开发了一部分分布式系统的基础设施的实现，如Spring Cloud Stream扮演的就是kafka, ActiveMQ这样的角色。

### Spring Cloud Config

集中配置管理工具，分布式系统中统一的外部配置管理，默认使用Git来存储配置，可以支持客户端配置的刷新及加密、解密操作。

### Spring Cloud Netflix

Netflix OSS 开源组件集成，包括Eureka、Hystrix、Ribbon、Feign、Zuul等核心组件。

Eureka：服务治理组件，包括服务端的注册中心和客户端的服务发现机制；

Ribbon：负载均衡的服务调用组件，具有多种负载均衡调用策略；

Hystrix：服务容错组件，实现了断路器模式，为依赖服务的出错和延迟提供了容错能力；

Feign：基于Ribbon和Hystrix的声明式服务调用组件；

Zuul：API网关组件，对请求提供路由及过滤功能。

Spring Cloud Bus

用于传播集群状态变化的消息总线，使用轻量级消息代理链接分布式系统中的节点，可以用来动态刷新集群中的服务配置。

### Spring Cloud Consul

基于Hashicorp Consul的服务治理组件。

### Spring Cloud Security

安全工具包，对Zuul代理中的负载均衡OAuth2客户端及登录认证进行支持。

### Spring Cloud Sleuth

Spring Cloud应用程序的分布式请求链路跟踪，支持使用Zipkin、HTrace和基于日志（例如ELK）的跟踪。

### Spring Cloud Stream

轻量级事件驱动微服务框架，可以使用简单的声明式模型来发送及接收消息，主要实现为Apache Kafka及RabbitMQ。

### Spring Cloud Task

用于快速构建短暂、有限数据处理任务的微服务框架，用于向应用中添加功能性和非功能性的特性。

### Spring Cloud Zookeeper

基于Apache Zookeeper的服务治理组件。

### Spring Cloud Gateway

API网关组件，对请求提供路由及过滤功能。

### Spring Cloud OpenFeign

基于Ribbon和Hystrix的声明式服务调用组件，可以动态创建基于Spring MVC注解的接口实现用于服务调用，在Spring Cloud 2.0中已经取代Feign成为了一等公民。

Spring Cloud的版本关系

Spring Cloud是一个由许多子项目组成的综合项目，各子项目有不同的发布节奏。 为了管理Spring Cloud与各子项目的版本依赖关系，发布了一个清单，其中包括了某个Spring Cloud版本对应的子项目版本。 为了避免Spring Cloud版本号与子项目版本号混淆，Spring Cloud版本采用了名称而非版本号的命名，这些版本的名字采用了伦敦地铁站的名字，根据字母表的顺序来对应版本时间顺序，例如Angel是第一个版本，Brixton是第二个版本。 当Spring Cloud的发布内容积累到临界点或者一个重大BUG被解决后，会发布一个"service releases"版本，简称SRX版本，比如Greenwich.SR2就是Spring Cloud发布的Greenwich版本的第2个SRX版本。目前Spring Cloud的最新版本是Hoxton。

Spring Cloud和SpringBoot版本对应关系

Spring Cloud Version SpringBoot Version

Hoxton 2.2.x

Greenwich 2.1.x

Finchley 2.0.x

Edgware 1.5.x

Dalston 1.5.x

Spring Cloud和各子项目版本对应关系

Component Edgware.SR6 Greenwich.SR2

spring-cloud-bus 1.3.4.RELEASE 2.1.2.RELEASE

spring-cloud-commons 1.3.6.RELEASE 2.1.2.RELEASE

spring-cloud-config 1.4.7.RELEASE 2.1.3.RELEASE

spring-cloud-netflix 1.4.7.RELEASE 2.1.2.RELEASE

spring-cloud-security 1.2.4.RELEASE 2.1.3.RELEASE

spring-cloud-consul 1.3.6.RELEASE 2.1.2.RELEASE

spring-cloud-sleuth 1.3.6.RELEASE 2.1.1.RELEASE

spring-cloud-stream Ditmars.SR5 Fishtown.SR3

spring-cloud-zookeeper 1.2.3.RELEASE 2.1.2.RELEASE

spring-boot 1.5.21.RELEASE 2.1.5.RELEASE

spring-cloud-task 1.2.4.RELEASE 2.1.2.RELEASE

spring-cloud-gateway 1.0.3.RELEASE 2.1.2.RELEASE

spring-cloud-openfeign 暂无 2.1.2.RELEASE

注意：Hoxton版本是基于SpringBoot 2.2.x版本构建的，不适用于1.5.x版本。随着2019年8月SpringBoot 1.5.x版本停止维护，Edgware版本也将停止维护。

## SpringBoot和SpringCloud的区别？

SpringBoot专注于快速方便的开发单个个体微服务。

SpringCloud是关注全局的微服务协调整理治理框架，它将SpringBoot开发的一个个单体微服务整合并管理起来，

为各个微服务之间提供，配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等集成服务

SpringBoot可以离开SpringCloud独立使用开发项目， 但是SpringCloud离不开SpringBoot ，属于依赖的关系

SpringBoot专注于快速、方便的开发单个微服务个体，SpringCloud关注全局的服务治理框架。

使用 Spring Boot 开发分布式微服务时，我们面临以下问题

（1）与分布式系统相关的复杂性-这种开销包括网络问题，延迟开销，带宽问题，安全问题。

（2）服务发现-服务发现工具管理群集中的流程和服务如何查找和互相交谈。它涉及一个服务目录，在该目录中注册服务，然后能够查找并连接到该目录中的服务。

（3）冗余-分布式系统中的冗余问题。

（4）负载平衡 --负载平衡改善跨多个计算资源的工作负荷，诸如计算机，计算机集群，网络链路，中央处理单元，或磁盘驱动器的分布。

（5）性能-问题 由于各种运营开销导致的性能问题。

（6）部署复杂性-Devops 技能的要求。

## 服务注册和发现是什么意思？Spring Cloud 如何实现？

当我们开始一个项目时，我们通常在属性文件中进行所有的配置。随着越来越多的服务开发和部署，添加和修改这些属性变得更加复杂。有些服务可能会下降，而某些位置可能会发生变化。手动更改属性可能会产生问题。 Eureka 服务注册和发现可以在这种情况下提供帮助。由于所有服务都在 Eureka 服务器上注册并通过调用 Eureka 服务器完成查找，因此无需处理服务地点的任何更改和处理。

## Spring Cloud 和dubbo区别?

（1）服务调用方式 dubbo是RPC springcloud Rest Api

（2）注册中心,dubbo 是zookeeper springcloud是eureka，也可以是zookeeper

（3）服务网关,dubbo本身没有实现，只能通过其他第三方技术整合，springcloud有Zuul路由网关，作为路由服务器，进行消费者的请求分发,springcloud支持断路器，与git完美集成配置文件支持版本控制，事物总线实现配置文件的更新与服务自动装配等等一系列的微服务架构要素。

## 负载平衡的意义什么？

在计算中，负载平衡可以改善跨计算机，计算机集群，网络链接，中央处理单元或磁盘驱动器等多种计算资源的工作负载分布。负载平衡旨在优化资源使用，最大化吞吐量，最小化响应时间并避免任何单一资源的过载。使用多个组件进行负载平衡而不是单个组件可能会通过冗余来提高可靠性和可用性。负载平衡通常涉及专用软件或硬件，例如多层交换机或域名系统服务器进程。

## 什么是 Hystrix？它如何实现容错？

Hystrix 是一个延迟和容错库，旨在隔离远程系统，服务和第三方库的访问点，当出现故障是不可避免的故障时，停止级联故障并在复杂的分布式系统中实现弹性。

通常对于使用微服务架构开发的系统，涉及到许多微服务。这些微服务彼此协作。

假设如果上图中的微服务 9 失败了，那么使用传统方法我们将传播一个异常。但这仍然会导致整个系统崩溃。

随着微服务数量的增加，这个问题变得更加复杂。微服务的数量可以高达 1000.这是 hystrix 出现的地方 我们将使用 Hystrix 在这种情况下的 Fallback 方法功能。我们有两个服务 employee-consumer 使用由 employee-consumer 公开的服务。

现在假设由于某种原因，employee-producer 公开的服务会抛出异常。我们在这种情况下使用 Hystrix 定义了一个回退方法。这种后备方法应该具有与公开服务相同的返回类型。如果暴露服务中出现异常，则回退方法将返回一些值。

## 什么是 Hystrix 断路器？我们需要它吗？

由于某些原因，employee-consumer 公开服务会引发异常。在这种情况下使用Hystrix 我们定义了一个回退方法。如果在公开服务中发生异常，则回退方法返回一些默认值。

如果 firstPage method() 中的异常继续发生，则 Hystrix 电路将中断，并且员工使用者将一起跳过 firtsPage 方法，并直接调用回退方法。 断路器的目的是给第一页方法或第一页方法可能调用的其他方法留出时间，并导致异常恢复。可能发生的情况是，在负载较小的情况下，导致异常的问题有更好的恢复机会 。

## 什么是 Netflix Feign？它的优点是什么？

Feign 是受到 Retrofit，JAXRS-2.0 和 WebSocket 启发的 java 客户端联编程序。

Feign 的第一个目标是将约束分母的复杂性统一到 http apis，而不考虑其稳定性。

在 employee-consumer 的例子中，我们使用了 employee-producer 使用 REST模板公开的 REST 服务。

但是我们必须编写大量代码才能执行以下步骤

（1）使用功能区进行负载平衡。

（2）获取服务实例，然后获取基本 URL。

（3）利用 REST 模板来使用服务。 前面的代码如下

@Controller

public class ConsumerControllerClient {

@Autowired

private LoadBalancerClient loadBalancer;

public void getEmployee() throws RestClientException, IOException {

ServiceInstance serviceInstance=loadBalancer.choose("employee-producer");

System.out.println(serviceInstance.getUri());

String baseUrl=serviceInstance.getUri().toString();

baseUrl=baseUrl+"/employee";

RestTemplate restTemplate = new RestTemplate();

ResponseEntity<String> response=null;

try{

response=restTemplate.exchange(baseUrl,

HttpMethod.GET, getHeaders(),String.class);

}

catch (Exception ex)

{

System.out.println(ex);

}

System.out.println(response.getBody());

}

之前的代码，有像 NullPointer 这样的例外的机会，并不是最优的。我们将看到如何使用 Netflix Feign 使呼叫变得更加轻松和清洁。如果 Netflix Ribbon 依赖关系也在类路径中，那么 Feign 默认也会负责负载平衡。

## 什么是 Spring Cloud Bus？我们需要它吗？

考虑以下情况：我们有多个应用程序使用 Spring Cloud Config 读取属性，而Spring Cloud Config 从 GIT 读取这些属性。

下面的例子中多个员工生产者模块从 Employee Config Module 获取 Eureka 注册的财产。

如果假设 GIT 中的 Eureka 注册属性更改为指向另一台 Eureka 服务器，会发生什么情况。在这种情况下，我们将不得不重新启动服务以获取更新的属性。

还有另一种使用执行器端点/刷新的方式。但是我们将不得不为每个模块单独调用这个 url。例如，如果 Employee Producer1 部署在端口 8080 上，则调用 http：// localhost：8080 / refresh。同样对于 Employee Producer2 http：//localhost：8081 / refresh 等等。这又很麻烦。这就是 Spring Cloud Bus 发挥作用的地方。

Spring Cloud Bus 提供了跨多个实例刷新配置的功能。因此，在上面的示例中，如果我们刷新 Employee Producer1，则会自动刷新所有其他必需的模块。如果我们有多个微服务启动并运行，这特别有用。这是通过将所有微服务连接到单个消息代理来实现的。无论何时刷新实例，此事件都会订阅到侦听此代理的所有微服务，并且它们也会刷新。可以通过使用端点/总线/刷新来实现对任何单个实例的刷新。

## Spring Cloud断路器的作用

当一个服务调用另一个服务由于网络原因或自身原因出现问题，调用者就会等待被调用者的响应 当更多的服务请求到这些资源导致更多的请求等待，发生连锁效应（雪崩效应）

断路器有完全打开状态:一段时间内 达到一定的次数无法调用 并且多次监测没有恢复的迹象 断路器完全打开 那么下次请求就不会请求到该服务

半开:短时间内 有恢复迹象 断路器会将部分请求发给该服务，正常调用时 断路器关闭

关闭：当服务一直处于正常状态 能正常调用

## 什么是Spring Cloud Config?

在分布式系统中，由于服务数量巨多，为了方便服务配置文件统一管理，实时更新，所以需要分布式配置中心组件。在Spring Cloud中，有分布式配置中心组件spring cloud config ，它支持配置服务放在配置服务的内存中（即本地），也支持放在远程Git仓库中。在spring cloud config 组件中，分两个角色，一是config server，二是config client。

使用：

（1）添加pom依赖

（2）配置文件添加相关配置

（3）启动类添加注解@EnableConfigServer

## 什么是Spring Cloud Gateway?

Spring Cloud Gateway是Spring Cloud官方推出的第二代网关框架，取代Zuul网关。网关作为流量的，在微服务系统中有着非常作用，网关常见的功能有路由转发、权限校验、限流控制等作用。

使用了一个RouteLocatorBuilder的bean去创建路由，除了创建路由RouteLocatorBuilder可以让你添加各种predicates和filters，predicates断言的意思，顾名思义就是根据具体的请求的规则，由具体的route去处理，filters是各种过滤器，用来对请求做各种判断和修改。

# Mysql相关

1）数据永久保存

2）使用SQL语句，查询方便效率高。

1. 管理数据方便

结构化查询语言(Structured Query Language)简称SQL，是一种数据库查询语言。

数据库三大范式是什么

第一范式：每个列都不可以再拆分。

第二范式：在第一范式的基础上，非主键列完全依赖于主键，而不能是依赖于主键的一部分。

第三范式：在第二范式的基础上，非主键列只依赖于主键，不依赖于其他非主键。

## mysql有关权限的表都有哪几个

user权限表：记录允许连接到服务器的用户帐号信息，里面的权限是全局级的。

db权限表：记录各个帐号在各个数据库上的操作权限。

table\_priv权限表：记录数据表级的操作权限。

columns\_priv权限表：记录数据列级的操作权限。

host权限表：配合db权限表对给定主机上数据库级操作权限作更细致的控制。这个权限表不受GRANT和REVOKE语句的影响。

## mysql有哪些数据类型

1、整数类型，包括TINYINT、SMALLINT、MEDIUMINT、INT、BIGINT，分别表示1字节、2字节、3字节、4字节、8字节整数。任何整数类型都可以加上UNSIGNED属性，表示数据是无符号的，即非负整数。

长度：整数类型可以被指定长度，例如：INT(11)表示长度为11的INT类型。长度在大多数场景是没有意义的，它不会限制值的合法范围，只会影响显示字符的个数，而且需要和UNSIGNED ZEROFILL属性配合使用才有意义。

例子，假定类型设定为INT(5)，属性为UNSIGNED ZEROFILL，如果用户插入的数据为12的话，那么数据库实际存储数据为00012。

2、实数类型，包括FLOAT、DOUBLE、DECIMAL。

DECIMAL可以用于存储比BIGINT还大的整型，能存储精确的小数。

而FLOAT和DOUBLE是有取值范围的，并支持使用标准的浮点进行近似计算。

计算时FLOAT和DOUBLE相比DECIMAL效率更高一些，DECIMAL你可以理解成是用字符串进行处理。

3、字符串类型，包括VARCHAR、CHAR、TEXT、BLOB

VARCHAR用于存储可变长字符串，它比定长类型更节省空间。

VARCHAR使用额外1或2个字节存储字符串长度。列长度小于255字节时，使用1字节表示，否则使用2字节表示。

VARCHAR存储的内容超出设置的长度时，内容会被截断。

CHAR是定长的，根据定义的字符串长度分配足够的空间。

CHAR会根据需要使用空格进行填充方便比较。

CHAR适合存储很短的字符串，或者所有值都接近同一个长度。

CHAR存储的内容超出设置的长度时，内容同样会被截断。

使用策略：

对于经常变更的数据来说，CHAR比VARCHAR更好，因为CHAR不容易产生碎片。

对于非常短的列，CHAR比VARCHAR在存储空间上更有效率。

使用时要注意只分配需要的空间，更长的列排序时会消耗更多内存。

尽量避免使用TEXT/BLOB类型，查询时会使用临时表，导致严重的性能开销。

4、枚举类型（ENUM），把不重复的数据存储为一个预定义的集合。

有时可以使用ENUM代替常用的字符串类型。

ENUM存储非常紧凑，会把列表值压缩到一个或两个字节。

ENUM在内部存储时，其实存的是整数。

尽量避免使用数字作为ENUM枚举的常量，因为容易混乱。

排序是按照内部存储的整数

5、日期和时间类型，尽量使用timestamp，空间效率高于datetime，

用整数保存时间戳通常不方便处理。

如果需要存储微妙，可以使用bigint存储。

看到这里，这道真题是不是就比较容易回答了。

## MyISAM与InnoDB区别

Innodb引擎：Innodb引擎提供了对数据库ACID事务的支持。并且还提供了行级锁和外键的约束。它的设计的目标就是处理大数据容量的数据库系统。

Myisam引擎叶子节点存储的是数据的地址值

## 索引

索引是一种特殊的文件(InnoDB数据表上的索引是表空间的一个组成部分)，它们包含着对数据表里所有记录的引用指针。

索引是一种数据结构。数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用B树及其变种B+树。

更通俗的说，索引就相当于目录。为了方便查找书中的内容，通过对内容建立索引形成目录。索引是一个文件，它是要占据物理空间的。

- 索引的基本原理

索引用来快速地寻找那些具有特定值的记录。如果没有索引，一般来说执行查询时遍历整张表。

索引的原理很简单，就是把无序的数据变成有序的查询

把创建了索引的列的内容进行排序

对排序结果生成倒排表

在倒排表内容上拼上数据地址链

在查询的时候，先拿到倒排表内容，再取出数据地址链，从而拿到具体数据

### 索引优缺点？

- 索引的优点

可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

- 索引的缺点

时间方面：创建索引和维护索引要耗费时间，具体地，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，会降低增/改/删的执行效率；

空间方面：索引需要占物理空间。

### 索引使用场景（重点）

where条件

order by

当我们使用order by将查询结果按照某个字段排序时，如果该字段没有建立索引，那么执行计划会将查询出的所有数据使用外部排序（将数据从硬盘分批读取到内存使用内部排序，最后合并排序结果），这个操作是很影响性能的，因为需要将查询涉及到的所有数据从磁盘中读到内存（如果单条数据过大或者数据量过多都会降低效率），更无论读到内存之后的排序了。

但是如果我们对该字段建立索引alter table 表名 add index(字段名)，那么由于索引本身是有序的，因此直接按照索引的顺序和映射关系逐条取出数据即可。而且如果分页的，那么只用取出索引表某个范围内的索引对应的数据，而不用像上述那取出所有数据进行排序再返回某个范围内的数据。（从磁盘取数据是最影响性能的）

- 哪些情况下适合建索引

　　1. 频繁作为where条件语句查询的字段

　　2. 关联字段需要建立索引，例如外键字段，student表中的classid, classes表中的schoolid 等(定义有外键的数据列一定要建立索引)

　　3. 排序字段可以建立索引

　　4. 分组字段可以建立索引，因为分组的前提是排序

　　5. 统计字段可以建立索引，例如count(),max()

- 哪些情况下不适合建索引

　　1. 频繁更新的字段不适合建立索引

　　2. where条件中用不到的字段不适合建立索引

　3. 表数据可以确定比较少的不需要建索引

　　4. 数据重复且发布比较均匀的的字段不适合建索引（唯一性太差的字段不适合建立索引），例如性别，真假值

　5. 参与列计算的列不适合建索引

　　6. 对于定义为text、image和bit的数据类型的列不要建立索引。

### 索引有哪几种类型？

- 主键索引: 数据列不允许重复，不允许为NULL，一个表只能有一个主键。主键索引叶子节点存放的是数据，非主键索引叶子节点存储的是主键值。这是一种时间换空间的思路

- 唯一索引: 数据列不允许重复，允许为NULL值，一个表允许多个列创建唯一索引。

- 普通索引: 基本的索引类型，没有唯一性的限制，允许为NULL值。

- 全文索引： 是目前搜索引擎使用的一种关键技术。

### 创建索引时的注意

- 非空字段：应该指定列为NOT NULL，除非你想存储NULL。在mysql中，含有空值的列很难进行查询优化，因为它们使得索引、索引的统计信息以及比较运算更加复杂。你应该用0、一个特殊的值或者一个空串代替空值；

- 取值离散大的字段：（变量各个取值之间的差异程度）的列放到联合索引的前面，可以通过count()函数查看字段的差异值，返回值越大说明字段的唯一值越多字段的离散程度高；

- 索引字段越小越好：数据库的数据存储以页为单位一页存储的数据越多一次IO操作获取的数据越大效率越高。

### 索引查询一定能提高查询的性能吗

通常，通过索引查询数据比全表扫描要快。但是我们也必须注意到它的代价。

- 索引需要空间来存储，也需要定期维护， 每当有记录在表中增减或索引列被修改时，索引本身也会被修改。 这意味着每条记录的INSERT，DELETE，UPDATE将为此多付出4，5 次的磁盘I/O。 因为索引需要额外的存储空间和处理，那些不必要的索引反而会使查询反应时间变慢。

- 使用索引查询不一定能提高查询性能，索引范围查询(INDEX RANGE SCAN)适用于两种情况:

- 基于一个范围的检索，一般查询返回结果集小于表中记录数的30%,对于符合条件比较少的用索引效率会高。

- 基于非唯一性索引的检索

### 最左前缀原则；最左匹配原则

- 顾名思义，就是最左优先，在创建多列索引时，要根据业务需求，where子句中使用最频繁的一列放在最左边。

- 最左前缀匹配原则，非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

- =和in可以乱序，比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式

### B树和B+树的区别

- 在B树中，你可以将键和值存放在内部节点和叶子节点；B树的叶子节点各自独立。

- 在B+树中，内部节点都是键，没有值，叶子节点同时存放键和值。叶子节点之间会有指针指向，大大提升范围查询的效率，方便遍历整个树，不再需要中序遍历

- 使用B树的好处：B树可以在内部节点同时存储键和值，因此，把频繁访问的数据放在靠近根节点的地方将会大大提高热点数据的查询效率。这种特性使得B树在特定数据重复多次查询的场景中更加高效。

- 使用B+树的好处：由于B+树的内部节点只存放键，不存放值，因此，一次读取，可以在内存页中获取更多的键，有利于更快地缩小查找范围。 B+树的叶节点由一条链相连，因此，当需要进行一次全数据遍历的时候，B+树只需要使用O(logN)时间找到最小的一个节点，然后通过链进行O(N)的顺序遍历即可。B树则需要对树的每一层进行遍历，这会需要更多的内存置换次数，因此也就需要花费更多的时间

- B树只适合随机检索，而B+树同时支持随机检索和顺序检索；

- B+树空间利用率更高，可减少I/O次数，磁盘读写代价更低。一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗。B+树的内部结点并没有指向关键字具体信息的指针，只是作为索引使用，其内部结点比B树小，盘块能容纳的结点中关键字数量更多，一次性读入内存中可以查找的关键字也就越多，相对的，IO读写次数也就降低了。而IO读写次数是影响索引检索效率的最大因素；

- B+树的查询效率更加稳定。B树搜索有可能会在非叶子结点结束，越靠近根节点的记录查找时间越短，只要找到关键字即可确定记录的存在，其性能等价于在关键字全集内做一次二分查找。而在B+树中，顺序检索比较明显，随机检索时，任何关键字的查找都必须走一条从根节点到叶节点的路，所有关键字的查找路径长度相同，导致每一个关键字的查询效率相当。

- B-树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。B+树的叶子节点使用指针顺序连接在一起，只要遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作。

- 增删文件（节点）时，效率更高。因为B+树的叶子节点包含所有关键字，并以有序的链表结构存储，这样可很好提高增删效率。

### Hash索引和B+树所有有什么区别或者说优劣呢?

hash索引进行等值查询更快(一般情况下)，但是却无法进行范围查询。

在大多数情况下，直接选择B+树索引可以获得稳定且较好的查询速度。而不需要使用hash索引。

### B+树在满足聚簇索引和覆盖索引的时候不需要回表查询数据，

在B+树的索引中，叶子节点可能存储了当前的key值，也可能存储了当前的key值以及整行的数据，这就是聚簇索引和非聚簇索引。 在InnoDB中，只有主键索引是聚簇索引，如果没有主键，则挑选一个唯一键建立聚簇索引。如果没有唯一键，则隐式的生成一个键来建立聚簇索引。

当查询使用聚簇索引时，在对应的叶子节点，可以获取到整行数据，因此不用再次进行回表查询。

### 什么是聚簇索引？何时使用聚簇索引与非聚簇索引

聚簇索引：将数据存储与索引放到了一块，找到索引也就找到了数据

非聚簇索引：将数据存储于索引分开结构，索引结构的叶子节点指向了数据的对应行，myisam通过key\_buffer把索引先缓存到内存中，当需要访问数据时（通过索引访问数据），在内存中直接搜索索引，然后通过索引找到磁盘相应数据，这也就是为什么索引不在key buffer命中时，速度慢的原因

推荐使用整型自增的主键

可以更好的维护我们的索引

## 事务

### 什么是数据库事务？

事务是一个不可分割的数据库操作序列，也是数据库并发控制的基本单位，其执行的结果必须使数据库从一种一致性状态变到另一种一致性状态。事务是逻辑上的一组操作，要么都执行，要么都不执行。

### 事物的四大特性(ACID)介绍一下?

. 原子性： 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；

. 一致性： 执行事务前后，数据保持一致，多个事务对同一个数据读取的结果是相同的；

. 隔离性： 并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的；

. 持久性： 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

### 什么是脏读？幻读？不可重复读？

- 脏读(Drity Read)：某个事务已更新一份数据，另一个事务在此时读取了同一份数据，由于某些原因，前一个RollBack了操作，则后一个事务所读取的数据就会是不正确的。

- 不可重复读(Non-repeatable read):在一个事务的两次查询之中数据不一致，这可能是两次查询过程中间插入了一个事务更新的原有的数据。

- 幻读(Phantom Read):在一个事务的两次查询中数据笔数不一致，例如有一个事务查询了几列(Row)数据，而另一个事务却在此时插入了新的几列数据，先前的事务在接下来的查询中，就会发现有几列数据是它先前所没有的。

### 什么是事务的隔离级别？MySQL的默认隔离级别是什么？

. READ-UNCOMMITTED(读取未提交)： 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读。

. READ-COMMITTED(读取已提交)： 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生。

. REPEATABLE-READ(可重复读)： 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。

. SERIALIZABLE(可串行化)： 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。

## 锁

### 对MySQL的锁了解吗

当数据库有并发事务的时候，可能会产生数据的不一致，这时候需要一些机制来保证访问的次序，锁机制就是这样的一个机制。

### 隔离级别与锁的关系

在Read Uncommitted级别下，读取数据不需要加共享锁，这样就不会跟被修改的数据上的排他锁冲突

在Read Committed级别下，读操作需要加共享锁，但是在语句执行完以后释放共享锁；

在Repeatable Read级别下，读操作需要加共享锁，但是在事务提交之前并不释放共享锁，也就是必须等待事务执行完毕以后才释放共享锁。

SERIALIZABLE 是限制性最强的隔离级别，因为该级别锁定整个范围的键，并一直持有锁，直到事务完成。

### 按照锁的粒度分数据库锁有哪些？锁机制与InnoDB锁算法

行级锁 行级锁是Mysql中锁定粒度最细的一种锁，表示只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，但加锁的开销也最大。行级锁分为共享锁 和 排他锁。

特点：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

表级锁 表级锁是MySQL中锁定粒度最大的一种锁，表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分MySQL引擎支持。最常使用的MYISAM与INNODB都支持表级锁定。表级锁定分为表共享读锁（共享锁）与表独占写锁（排他锁）。

特点：开销小，加锁快；不会出现死锁；锁定粒度大，发出锁冲突的概率最高，并发度最低。

页级锁 页级锁是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。

特点：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般

### 从锁的类别上分MySQL都有哪些锁呢

共享锁: 又叫做读锁。 当用户要进行数据的读取时，对数据加上共享锁。共享锁可以同时加上多个。

排他锁: 又叫做写锁。 当用户要进行数据的写入时，对数据加上排他锁。排他锁只可以加一个，他和其他的排他锁，共享锁都相斥。

### MySQL中InnoDB引擎的行锁是怎么实现的？

InnoDB是基于索引来完成行锁

### 什么是死锁？怎么解决？

死锁是指两个或多个事务在同一资源上相互占用，并请求锁定对方的资源，从而导致恶性循环的现象。

常见的解决死锁的方法

1、如果不同程序会并发存取多个表，尽量约定以相同的顺序访问表，可以大大降低死锁机会。

2、在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率；

3、对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率；

如果业务处理不好可以用分布式事务锁或者使用乐观锁

### 数据库的乐观锁和悲观锁是什么？怎么实现的？

悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。在查询完数据的时候就把事务锁起来，直到提交事务。实现方式：使用数据库中的锁机制

乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。在修改数据的时候把事务锁起来，通过version的方式来进行锁定。实现方式：乐一般会使用版本号机制或CAS算法实现。

两种锁的使用场景

从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。

但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行retry，这样反倒是降低了性能，所以一般多写的场景下用悲观锁就比较合适。

## 视图

### 为什么要使用视图？什么是视图？

为了提高复杂SQL语句的复用性和表操作的安全性，MySQL数据库管理系统提供了视图特性。所谓视图，本质上是一种虚拟表，在物理上是不存在的，其内容与真实的表相似，包含一系列带有名称的列和行数据。但是，视图并不在数据库中以储存的数据值形式存在。行和列数据来自定义视图的查询所引用基本表，并且在具体引用视图时动态生成。

视图使开发者只关心感兴趣的某些特定数据和所负责的特定任务，只能看到视图中所定义的数据，而不是视图所引用表中的数据，从而提高了数据库中数据的安全性。

### 视图有哪些特点？

视图的特点如下:

视图的列可以来自不同的表，是表的抽象和在逻辑意义上建立的新关系。

视图是由基本表(实表)产生的表(虚表)。

视图的建立和删除不影响基本表。

对视图内容的更新(添加，删除和修改)直接影响基本表。

当视图来自多个基本表时，不允许添加和删除数据。

视图的操作包括创建视图，查看视图，删除视图和修改视图。

### 视图的使用场景有哪些？

简化sql查询，提高开发效率。如果说还有另外一个用途那就是兼容老的表结构。

下面是视图的常见使用场景：

重用SQL语句；

简化复杂的SQL操作。在编写查询后，可以方便的重用它而不必知道它的基本查询细节；

使用表的组成部分而不是整个表；

保护数据。可以给用户授予表的特定部分的访问权限而不是整个表的访问权限；

更改数据格式和表示。视图可返回与底层表的表示和格式不同的数据。

### 视图的优点

- 查询简单化。视图能简化用户的操作

- 数据安全性。视图使用户能以多种角度看待同一数据，能够对机密数据提供安全保护

- 逻辑数据独立性。视图对重构数据库提供了一定程度的逻辑独立性

### 视图的缺点

性能。数据库必须把视图的查询转化成对基本表的查询，如果这个视图是由一个复杂的多表查询所定义，那么，即使是视图的一个简单查询，数据库也把它变成一个复杂的结合体，需要花费一定的时间。

修改限制。当用户试图修改视图的某些行时，数据库必须把它转化为对基本表的某些行的修改。事实上，当从视图中插入或者删除时，情况也是这样。对于简单视图来说，这是很方便的，但是，对于比较复杂的视图，可能是不可修改的

这些视图有如下特征：1.有UNIQUE等集合操作符的视图。2.有GROUP BY子句的视图。3.有诸如AVG\SUM\MAX等聚合函数的视图。 4.使用DISTINCT关键字的视图。5.连接表的视图（其中有些例外）

## 存储过程与函数

### 什么是存储过程？有哪些优缺点？

存储过程是一个预编译的SQL语句，优点是允许模块化的设计，就是说只需要创建一次，以后在该程序中就可以调用多次。如果某次操作需要执行多次SQL，使用存储过程比单纯SQL语句执行要快。

优点

1）存储过程是预编译过的，执行效率高。

2）存储过程的代码直接存放于数据库中，通过存储过程名直接调用，减少网络通讯。

3）安全性高，执行存储过程需要有一定权限的用户。

4）存储过程可以重复使用，减少数据库开发人员的工作量。

缺点

1）调试麻烦，但是用 PL/SQL Developer 调试很方便！弥补这个缺点。

2）移植问题，数据库端代码当然是与数据库相关的。但是如果是做工程型项目，基本不存在移植问题。

3）重新编译问题，因为后端代码是运行前编译的，如果带有引用关系的对象发生改变时，受影响的存储过程、包将需要重新编译（不过也可以设置成运行时刻自动编译）。

4）如果在一个程序系统中大量的使用存储过程，到程序交付使用的时候随着用户需求的增加会导致数据结构的变化，接着就是系统的相关问题了，最后如果用户想维护该系统可以说是很难很难、而且代价是空前的，维护起来更麻烦。

## 触发器

### 什么是触发器？触发器的使用场景有哪些？

触发器是用户定义在关系表上的一类由事件驱动的特殊的存储过程。触发器是指一段代码，当触发某个事件时，自动执行这些代码。

使用场景

可以通过数据库中的相关表实现级联更改。

实时监控某张表中的某个字段的更改而需要做出相应的处理。

例如可以生成某些业务的编号。

注意不要滥用，否则会造成数据库及应用程序的维护困难。

大家需要牢记以上基础知识点，重点是理解数据类型CHAR和VARCHAR的差异，表存储引擎InnoDB和MyISAM的区别。

### MySQL中都有哪些触发器？

在MySQL数据库中有如下六种触发器：

- Before Insert

- After Insert

- Before Update

- After Update

- Before Delete

- After Delete

## 常用SQL语句

### SQL语句主要分为哪几类

数据定义语言DDL（Data Ddefinition Language）CREATE，DROP，ALTER

主要为以上操作 即对逻辑结构等有操作的，其中包括表结构，视图和索引。

数据查询语言DQL（Data Query Language）SELECT

这个较为好理解 即查询操作，以select关键字。各种简单查询，连接查询等 都属于DQL。

数据操纵语言DML（Data Manipulation Language）INSERT，UPDATE，DELETE

主要为以上操作 即对数据进行操作的，对应上面所说的查询操作 DQL与DML共同构建了多数初级程序员常用的增删改查操作。而查询是较为特殊的一种 被划分到DQL中。

数据控制功能DCL（Data Control Language）GRANT，REVOKE，COMMIT，ROLLBACK

主要为以上操作 即对数据库安全性完整性等有操作的，可以简单的理解为权限控制等。

### 超键、候选键、主键、外键分别是什么？

- 超键：在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。

- 候选键：是最小超键，即没有冗余元素的超键。

- 主键：数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。

- 外键：在一个表中存在的另一个表的主键称此表的外键。

### SQL 约束有哪几种？

- NOT NULL: 用于控制字段的内容一定不能为空（NULL）。

- UNIQUE: 控件字段内容不能重复，一个表允许有多个 Unique 约束。

- PRIMARY KEY: 也是用于控件字段内容不能重复，但它在一个表只允许出现一个。

- FOREIGN KEY: 用于预防破坏表之间连接的动作，也能防止非法数据插入外键列，因为它必须是它指向的那个表中的值之一。

- CHECK: 用于控制字段的值范围。

### 六种关联查询

内连接分为三类

- 等值连接：ON A.id=B.id

- 不等值连接：ON A.id > B.id

- 自连接：SELECT \* FROM A T1 INNER JOIN A T2 ON T1.id=T2.pid

外连接（LEFT JOIN/RIGHT JOIN）

- 左外连接：LEFT OUTER JOIN, 以左表为主，先查询出左表，按照ON后的关联条件匹配右表，没有匹配到的用NULL填充，可以简写成LEFT JOIN

- 右外连接：RIGHT OUTER JOIN, 以右表为主，先查询出右表，按照ON后的关联条件匹配左表，没有匹配到的用NULL填充，可以简写成RIGHT JOIN

联合查询（UNION与UNION ALL）

SELECT \* FROM A UNION SELECT \* FROM B UNION ...

- 就是把多个结果集集中在一起，UNION前的结果为基准，需要注意的是联合查询的列数要相等，相同的记录行会合并

- 如果使用UNION ALL，不会合并重复的记录行

- 效率 UNION 高于 UNION ALL

### 什么是子查询

条件：一条SQL语句的查询结果做为另一条查询语句的条件或查询结果

嵌套：多条SQL语句嵌套使用，内部的SQL查询语句称为子查询。

### mysql中 in 和 exists 区别

mysql中的in语句是把外表和内表作hash 连接，而exists语句是对外表作loop循环，每次loop循环再对内表进行查询。一直大家都认为exists比in语句的效率要高，这种说法其实是不准确的。这个是要区分环境的。

如果查询的两个表大小相当，那么用in和exists差别不大。

如果两个表中一个较小，一个是大表，则子查询表大的用exists，子查询表小的用in。

not in 和not exists：如果查询语句使用了not in，那么内外表都进行全表扫描，没有用到索引；而not extsts的子查询依然能用到表上的索引。所以无论那个表大，用not exists都比not in要快。

## 区别和含义

### varchar与char的区别

char的特点

char表示定长字符串，长度是固定的；

如果插入数据的长度小于char的固定长度时，则用空格填充；

因为长度固定，所以存取速度要比varchar快很多，甚至能快50%，但正因为其长度固定，所以会占据多余的空间，是空间换时间的做法；

对于char来说，最多能存放的字符个数为255，和编码无关

varchar的特点

varchar表示可变长字符串，长度是可变的；

插入的数据是多长，就按照多长来存储；

varchar在存取方面与char相反，它存取慢，因为长度不固定，但正因如此，不占据多余的空间，是时间换空间的做法；

对于varchar来说，最多能存放的字符个数为65532

总之，结合性能角度（char更快）和节省磁盘空间角度（varchar更小），具体情况还需具体来设计数据库才是妥当的做法。

### varchar(50)中50的涵义

最多存放50个字符，varchar(50)和(200)存储hello所占空间一样，但后者在排序时会消耗更多内存，因为order by col采用fixed\_length计算col长度(memory引擎也一样)。在早期 MySQL 版本中， 50 代表字节数，现在代表字符数。

### int(20)中20的涵义

是指显示字符的长度。20表示最大显示宽度为20，但仍占4字节存储，存储范围不变；

不影响内部存储，只是影响带 zerofill 定义的 int 时，前面补多少个 0，易于报表展示

### mysql中int(10)和char(10)以及varchar(10)的区别

int(10)的10表示显示的数据的长度，不是存储数据的大小；chart(10)和varchar(10)的10表示存储数据的大小，即表示存储多少个字符。

int(10) 10位的数据长度 9999999999，占32个字节，int型4位

char(10) 10位固定字符串，不足补空格 最多10个字符

varchar(10) 10位可变字符串，不足补空格 最多10个字符

char(10)表示存储定长的10个字符，不足10个就用空格补齐，占用更多的存储空间

varchar(10)表示存储10个变长的字符，存储多少个就是多少个，空格也按一个字符存储，这一点是和char(10)的空格不同的，char(10)的空格表示占位不算一个字符

### UNION与UNION ALL的区别？

如果使用UNION ALL，不会合并重复的记录行

效率 UNION 高于 UNION ALL

## SQL优化

### 如何定位及优化SQL语句的性能问题？

对于低性能的SQL语句的定位，最重要也是最有效的方法就是使用执行计划，MySQL提供了explain命令来查看语句的执行计划。 我们知道，不管是哪种数据库，或者是哪种数据库引擎，在对一条SQL语句进行执行的过程中都会做很多相关的优化，对于查询语句，最重要的优化方式就是使用索引。 而执行计划，就是显示数据库引擎对于SQL语句的执行的详细情况，其中包含了是否使用索引，使用什么索引，使用的索引的相关信息等。

### SQL的生命周期？

应用服务器与数据库服务器建立一个连接

数据库进程拿到请求sql

解析并生成执行计划，执行

读取数据到内存并进行逻辑处理

通过步骤一的连接，发送结果到客户端

关掉连接，释放资源

### 大表数据查询，怎么优化

优化shema、sql语句+索引；

第二加缓存，memcached, redis；

主从复制，读写分离；

垂直拆分，根据你模块的耦合度，将一个大的系统分为多个小的系统，也就是分布式系统；

水平切分，针对数据量大的表，这一步最麻烦，最能考验技术水平，要选择一个合理的sharding key, 为了有好的查询效率，表结构也要改动，做一定的冗余，应用也要改，sql中尽量带sharding key，将数据定位到限定的表上去查，而不是扫描全部的表；

### 超大分页怎么处理？

先快速定位需要获取的id段，然后再关联：

SELECT a.\* FROM 表1 a, (select id from 表1 where 条件 LIMIT 100000,20 ) b where a.id=b.id

### mysql 分页

LIMIT n 等价于 LIMIT 0,n。

### 慢查询日志

用于记录执行时间超过某个临界值的SQL日志，用于快速定位慢查询，为我们的优化做参考。

### 为什么要尽量设定一个主键？

主键是数据库确保数据行在整张表唯一性的保障，即使业务上本张表没有主键，也建议添加一个自增长的ID列作为主键。设定了主键之后，在后续的删改查的时候可能更加快速以及确保操作数据范围安全。

### 主键使用自增ID还是UUID？

推荐使用自增ID，不要使用UUID。

因为在InnoDB存储引擎中，主键索引是作为聚簇索引存在的，也就是说，主键索引的B+树叶子节点上存储了主键索引以及全部的数据(按照顺序)，如果主键索引是自增ID，那么只需要不断向后排列即可，如果是UUID，由于到来的ID与原来的大小不确定，会造成非常多的数据插入，数据移动，然后导致产生很多的内存碎片，进而造成插入性能的下降。

总之，在数据量大一些的情况下，用自增主键性能会好一些。

关于主键是聚簇索引，如果没有主键，InnoDB会选择一个唯一键来作为聚簇索引，如果没有唯一键，会生成一个隐式的主键。

### 字段为什么要求定义为not null？

null值会占用更多的字节，且会在程序中造成很多与预期不符的情况。

### 如果要存储用户的密码散列，应该使用什么字段进行存储？

密码散列，盐，用户身份证号等固定长度的字符串应该使用char而不是varchar来存储，这样可以节省空间且提高检索效率。

### 优化查询过程中的数据访问

- 访问数据太多导致查询性能下降

- 确定应用程序是否在检索大量超过需要的数据，可能是太多行或列

- 确认MySQL服务器是否在分析大量不必要的数据行

- 避免犯如下SQL语句错误

- 查询不需要的数据。解决办法：使用limit解决

- 多表关联返回全部列。解决办法：指定列名

- 总是返回全部列。解决办法：避免使用SELECT \*

- 重复查询相同的数据。解决办法：可以缓存数据，下次直接读取缓存

- 是否在扫描额外的记录。解决办法：

- 使用explain进行分析，如果发现查询需要扫描大量的数据，但只返回少数的行，可以通过如下技巧去优化：

- 使用索引覆盖扫描，把所有的列都放到索引中，这样存储引擎不需要回表获取对应行就可以返回结果。

- 改变数据库和表的结构，修改数据表范式

- 重写SQL语句，让优化器可以以更优的方式执行查询。

### 优化长难的查询语句

- 一个复杂查询还是多个简单查询

- MySQL内部每秒能扫描内存中上百万行数据，相比之下，响应数据给客户端就要慢得多

- 使用尽可能小的查询是好的，但是有时将一个大的查询分解为多个小的查询是很有必要的。

- 切分查询

- 将一个大的查询分为多个小的相同的查询

- 一次性删除1000万的数据要比一次删除1万，暂停一会的方案更加损耗服务器开销。

- 分解关联查询，让缓存的效率更高。

- 执行单个查询可以减少锁的竞争。

- 在应用层做关联更容易对数据库进行拆分。

- 查询效率会有大幅提升。

- 较少冗余记录的查询。

### 优化特定类型的查询语句

- count(\*)会忽略所有的列，直接统计所有列数，不要使用count(列名)

- MyISAM中，没有任何where条件的count(\*)非常快。

- 当有where条件时，MyISAM的count统计不一定比其它引擎快。

- 可以使用explain查询近似值，用近似值替代count(\*)

- 增加汇总表

- 使用缓存

### 优化关联查询

确定ON或者USING子句中是否有索引。

确保GROUP BY和ORDER BY只有一个表中的列，这样MySQL才有可能使用索引。

### 优化子查询

- 用关联查询替代

- 优化GROUP BY和DISTINCT

- 这两种查询据可以使用索引来优化，是最有效的优化方法

- 关联查询中，使用标识列分组的效率更高

- 如果不需要ORDER BY，进行GROUP BY时加ORDER BY NULL，MySQL不会再进行文件排序。

- WITH ROLLUP超级聚合，可以挪到应用程序处理

### 优化LIMIT分页

LIMIT偏移量大的时候，查询效率较低

可以记录上次查询的最大ID，下次查询时直接根据该ID来查询

### 优化UNION查询

UNION ALL的效率高于UNION

### 优化WHERE子句

## 数据库优化

### 为什么要优化

- 系统的吞吐量瓶颈往往出现在数据库的访问速度上

- 随着应用程序的运行，数据库的中的数据会越来越多，处理时间会相应变慢

- 数据是存放在磁盘上的，读写速度无法和内存相比

- 优化原则：减少系统瓶颈，减少资源占用，增加系统的反应速度。

### 数据库结构优化

将字段很多的表分解成多个表

增加中间表

减少冗余字段

### 大表怎么优化？某个表有近千万数据，CRUD比较慢，如何优化？分库分表了是怎么做的？分表分库了有什么问题？有用到中间件么？他们的原理知道么？

- 限定数据的范围： 务必禁止不带任何限制数据范围条件的查询语句。比如：我们当用户在查询订单历史的时候，我们可以控制在一个月的范围内。；

- 读/写分离： 经典的数据库拆分方案，主库负责写，从库负责读；

- 缓存： 使用MySQL的缓存，另外对重量级、更新少的数据可以考虑使用应用级别的缓存；

### MySQL的复制原理以及流程

主从复制：将主数据库中的DDL和DML操作通过二进制日志（BINLOG）传输到从数据库上，然后将这些日志重新执行（重做）；从而使得从数据库的数据与主数据库保持一致。

主从复制的作用

主数据库出现问题，可以切换到从数据库。

可以进行数据库层面的读写分离。

可以在从数据库上进行日常备份。

MySQL主从复制解决的问题

数据分布：随意开始或停止复制，并在不同地理位置分布数据备份

负载均衡：降低单个服务器的压力

高可用和故障切换：帮助应用程序避免单点失败

升级测试：可以用更高版本的MySQL作为从库

MySQL主从复制工作原理

在主库上把数据更高记录到二进制日志

从库将主库的日志复制到自己的中继日志

从库读取中继日志的事件，将其重放到从库数据中

### 读写分离有哪些解决方案？

读写分离是依赖于主从复制，而主从复制又是为读写分离服务的。因为主从复制要求slave不能写只能读（如果对slave执行写操作，那么show slave status将会呈现Slave\_SQL\_Running=NO，此时你需要按照前面提到的手动同步一下slave）。

方案一

使用mysql-proxy代理

优点：直接实现读写分离和负载均衡，不用修改代码，master和slave用一样的帐号，mysql官方不建议实际生产中使用

缺点：降低性能， 不支持事务

方案二

使用AbstractRoutingDataSource+aop+annotation在dao层决定数据源。

如果采用了mybatis， 可以将读写分离放在ORM层，比如mybatis可以通过mybatis plugin拦截sql语句，所有的insert/update/delete都访问master库，所有的select 都访问salve库，这样对于dao层都是透明。 plugin实现时可以通过注解或者分析语句是读写方法来选定主从库。不过这样依然有一个问题， 也就是不支持事务， 所以我们还需要重写一下DataSourceTransactionManager， 将read-only的事务扔进读库， 其余的有读有写的扔进写库。

方案三

使用AbstractRoutingDataSource+aop+annotation在service层决定数据源，可以支持事务.

缺点：类内部方法通过this.xx()方式相互调用时，aop不会进行拦截，需进行特殊处理。

# Redis

Redis是一个key-value存储系统，大部分情况下是因为其高性能的特性，被当做缓存使用。已知性能最快的Key-Value DB；

## Redis 优势

- 性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。

- 丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。

- 原子 – Redis的所有操作都是原子性的，意思就是要么成功执行要么失败完全不执行。单个操作是原子性的。多个操作也支持事务，即原子性，通过MULTI和EXEC指令包起来。

- 丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

- Redis支持数据的备份，即master-slave模式的数据备份。

- Redis支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。

### 缺点：

数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。

## 为什么要用 Redis 而不用 map/guava 做缓存?

缓存分为本地缓存和分布式缓存。以 Java 为例，使用自带的 map 或者 guava 实现的是本地缓存，最主要的特点是轻量以及快速，生命周期随着 jvm 的销毁而结束，并且在多实例的情况下，每个实例都需要各自保存一份缓存，缓存不具有一致性。

使用 redis 或 memcached 之类的称为分布式缓存，在多实例的情况下，各实例共用一份缓存数据，缓存具有一致性。缺点是需要保持 redis 或 memcached服务的高可用，整个程序架构上较为复杂。

## Redis为什么这么快

1、完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。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、数据结构简单，对数据操作也简单，Redis 中的数据结构是专门进行设计的；

3、采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；

4、使用多路 I/O 复用模型，非阻塞 IO；

5、使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis 直接自己构建了 VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；

## 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)。

### Redis 字符串(String)

Redis 字符串数据类型的相关命令用于管理 redis 字符串值，基本语法如下：

### Redis 哈希(Hash)

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。

Redis 中每个 hash 可以存储 232 - 1 键值对（40多亿）。

### Redis 列表(List)

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）

一个列表最多可以包含 232 - 1 个元素 (4294967295, 每个列表超过40亿个元素)。

### Redis 集合(Set)

Redis 的 Set 是 String 类型的无序集合。集合成员是唯一的，这就意味着集合中不能出现重复的数据。

Redis 中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)。

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)。

### Redis 有序集合(sorted set)

Redis 有序集合和集合一样也是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。

有序集合的成员是唯一的,但分数(score)却可以重复。

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)。 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)。

## Redis的应用场景

\*\*总结一\*\*

- 计数器

可以对 String 进行自增自减运算，从而实现计数器功能。Redis 这种内存型数据库的读写性能非常高，很适合存储频繁读写的计数量。

- 缓存

将热点数据放到内存中，设置内存的最大使用量以及淘汰策略来保证缓存的命中率。

- 消息队列(发布/订阅功能)

List 是一个双向链表，可以通过 lpush 和 rpop 写入和读取消息。实现最新消息的排行。

- 存储对象

使用hash

## 持久化

持久化就是把内存的数据写到磁盘中去，防止服务宕机了内存数据丢失。

### Redis 的持久化机制是什么？各自的优缺点？

Redis 提供两种持久化机制 RDB 和 AOF 机制:

1. RDB：是Redis DataBase缩写快照（默认）

RDB是Redis默认的持久化方式。按照一定的时间将内存的数据以快照的形式保存到硬盘中，对应产生的数据文件为dump.rdb。通过配置文件中的save参数来定义快照的周期。

优点：

1、只有一个文件 dump.rdb，方便持久化。

2、容灾性好，一个文件可以保存到安全的磁盘。

3、性能最大化，fork 子进程来完成写操作，让主进程继续处理命令，所以是 IO 最大化。使用单独子进程来进行持久化，主进程不会进行任何 IO 操作，保证了 redis 的高性能

4.相对于数据集大时，比 AOF 的启动效率更高。

缺点：

- 1、数据安全性低。RDB 是间隔一段时间进行持久化，如果持久化之间 redis 发生故障，会发生数据丢失。适合数据要求不严谨的时候)

2. AOF：持久化

AOF持久化(即Append Only File持久化)，则是将Redis执行的每次写命令记录到单独的日志文件中，当重启Redis会重新将持久化的日志中文件恢复数据。

当两种方式同时开启时，数据恢复Redis会优先选择AOF恢复。

优点：

1、数据安全，aof 持久化可以配置 appendfsync 属性，有 always，每进行一次 命令操作就记录到 aof 文件中一次。

2、通过 append 模式写文件，即使中途服务器宕机，可以通过 redis-check-aof 工具解决数据一致性问题。

3、AOF 机制的 rewrite 模式。AOF 文件没被 rewrite 之前（文件过大时会对命令 进行合并重写），可以删除其中的某些命令（比如误操作的 flushall）)

缺点：

1、AOF 文件比 RDB 文件大，且恢复速度慢。

2、数据集大的时候，比 rdb 启动效率低。

优缺点是什么？

- AOF文件比RDB更新频率高，优先使用AOF还原数据。

- AOF比RDB更安全也更大

- RDB性能比AOF好

- 如果两个都配了优先加载AOF

## 过期键的删除策略

### Redis的过期键的删除策略

我们可以设置Redis中缓存的key的过期时间。过期策略就是指当Redis中缓存的key过期了如何处理。

过期策略通常有以下三种：

- 定时过期：每个设置过期时间的key都需要创建一个定时器，到过期时间就会立即清除。该策略可以立即清除过期的数据，对内存很友好；但是会占用大量的CPU资源去处理过期的数据，从而影响缓存的响应时间和吞吐量。

- 惰性过期：只有当访问一个key时，才会判断该key是否已过期，过期则清除。该策略可以最大化地节省CPU资源，却对内存非常不友好。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。

- 定期过期：每隔一定的时间，会扫描一定数量的数据库的expires字典中一定数量的key，并清除其中已过期的key。该策略是前两者的一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。

Redis中同时使用了惰性过期和定期过期两种过期策略。

### Redis的内存淘汰策略有哪些

Redis的内存淘汰策略是指在Redis的用于缓存的内存不足时，怎么处理需要新写入且需要申请额外空间的数据。

全局的键空间选择性移除

noeviction：当内存不足以容纳新写入数据时，新写入操作会报错。

allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key。（这个是最常用的）

allkeys-random：当内存不足以容纳新写入数据时，在键空间中，随机移除某个key。

设置过期时间的键空间选择性移除

volatile-lru：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的key。

volatile-random：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个key。

volatile-ttl：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的key优先移除。

总结

Redis的内存淘汰策略的选取并不会影响过期的key的处理。内存淘汰策略用于处理内存不足时的需要申请额外空间的数据；过期策略用于处理过期的缓存数据。

## Redis如何做内存优化？

可以好好利用Hash,list,sorted set,set等集合类型数据，因为通常情况下很多小的Key-Value可以用更紧凑的方式存放到一起。尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。

## 事务

### Redis事务的概念

Redis 事务的本质是通过MULTI、EXEC、WATCH等一组命令的集合。事务支持一次执行多个命令，一个事务中所有命令都会被序列化。在事务执行过程，会按照顺序串行化执行队列中的命令，其他客户端提交的命令请求不会插入到事务执行命令序列中。

总结说：redis事务就是一次性、顺序性、排他性的执行一个队列中的一系列命令。

### Redis事务相关命令

Redis事务功能是通过MULTI、EXEC、DISCARD和WATCH 四个原语实现的

Redis会将一个事务中的所有命令序列化，然后按顺序执行。

- redis 不支持回滚，“Redis 在事务失败时不进行回滚，而是继续执行余下的命令”， 所以 Redis 的内部可以保持简单且快速。

- 如果在一个事务中的命令出现错误，那么所有的命令都不会执行；

- 如果在一个事务中出现运行错误，那么正确的命令会被执行。

WATCH 命令是一个乐观锁，可以为 Redis 事务提供 check-and-set （CAS）行为。 可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行，监控一直持续到EXEC命令。

MULTI命令用于开启一个事务，它总是返回OK。 MULTI执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个队列中，当EXEC命令被调用时，所有队列中的命令才会被执行。

EXEC：执行所有事务块内的命令。返回事务块内所有命令的返回值，按命令执行的先后顺序排列。 当操作被打断时，返回空值 nil 。

通过调用DISCARD，客户端可以清空事务队列，并放弃执行事务， 并且客户端会从事务状态中退出。

UNWATCH命令可以取消watch对所有key的监控。

### 事务管理（ACID）概述

\*\*Redis的事务总是具有ACID中的一致性和隔离性\*\*，其他特性是不支持的。当服务器运行在\*AOF\*持久化模式下，并且appendfsync选项的值为always时，事务也具有耐久性。

Redis 是单进程程序，并且它保证在执行事务时，不会对事务进行中断，事务可以运行直到执行完所有事务队列中的命令为止。因此，\*\*Redis 的事务是总是带有隔离性的\*\*。

Redis中，单条命令是原子性执行的，但\*\*事务不保证原子性，且没有回滚\*\*。事务中任意命令执行失败，其余的命令仍会被执行。

## 集群方案

### 哨兵模式

sentinel，中文名是哨兵。哨兵是 redis 集群机构中非常重要的一个组件，主要有以下功能：

\* 集群监控：负责监控 redis master 和 slave 进程是否正常工作。

\* 消息通知：如果某个 redis 实例有故障，那么哨兵负责发送消息作为报警通知给管理员。

\* 故障转移：如果 master node 挂掉了，会自动转移到 slave node 上。

\* 配置中心：如果故障转移发生了，通知 client 客户端新的 master 地址。

哨兵用于实现 redis 集群的高可用，本身也是分布式的，作为一个哨兵集群去运行，互相协同工作。

故障转移时，判断一个 master node 是否宕机了，需要大部分的哨兵都同意才行，涉及到了分布式选举的问题。

\*\*哨兵的核心知识\*\*

- 哨兵至少需要 3 个实例，来保证自己的健壮性。

- 哨兵 + redis 主从的部署架构，是\*\*不保证数据零丢失\*\*的，只能保证 redis 集群的高可用性。

- 对于哨兵 + redis 主从这种复杂的部署架构，尽量在测试环境和生产环境，都进行充足的测试和演练。

### 官方Redis Cluster 方案(服务端路由查询)

采用slot(槽)的概念，一共分成16384个槽。将请求发送到任意节点，接收到请求的节点会将查询请求发送到正确的节点上执行

方案说明

1. 通过哈希的方式，将数据分片，每个节点均分存储一定哈希槽(哈希值)区间的数据，默认分配了16384 个槽位

2. 每份数据分片会存储在多个互为主从的多节点上

3. 数据写入先写主节点，再同步到从节点(支持配置为阻塞同步)

4. 同一分片多个节点间的数据不保持一致性

5. 读取数据时，当客户端操作的key没有分配在该节点上时，redis会返回转向指令，指向正确的节点

6. 扩容时时需要需要把旧节点的数据迁移一部分到新节点

### Redis 主从架构

单机的 redis，能够承载的 QPS 大概就在上万到几万不等。对于缓存来说，一般都是用来支撑读高并发的。因此架构做成主从(master-slave)架构，一主多从，主负责写，并且将数据复制到其它的 slave 节点，从节点负责读。所有的读请求全部走从节点。这样也可以很轻松实现水平扩容，支撑读高并发。

redis replication -> 主从架构 -> 读写分离 -> 水平扩容支撑读高并发

redis replication 的核心机制

- redis 采用异步方式复制数据到 slave 节点

- 一个 master node 是可以配置多个 slave node 的；

- slave node 也可以连接其他的 slave node；

- slave node 做复制的时候，不会 block master node 的正常工作；

- slave node 在做复制的时候，也不会 block 对自己的查询操作，它会用旧的数据集来提供服务；但是复制完成的时候，需要删除旧数据集，加载新数据集，这个时候就会暂停对外服务了；

- slave node 主要用来进行横向扩容，做读写分离，扩容的 slave node 可以提高读的吞吐量。

注意，如果采用了主从架构，那么建议必须开启 master node 的持久化，不建议用 slave node 作为 master node 的数据热备，因为那样的话，如果你关掉 master 的持久化，可能在 master 宕机重启的时候数据是空的，然后可能一经过复制， slave node 的数据也丢了。

### redis 主从复制的核心原理

当启动一个 slave node 的时候，它会发送一个 PSYNC 命令给 master node。

如果这是 slave node 初次连接到 master node，那么会触发一次 full resynchronization 全量复制。此时 master 会启动一个后台线程，开始生成一份 RDB 快照文件，

同时还会将从客户端 client 新收到的所有写命令缓存在内存中。RDB 文件生成完毕后， master 会将这个 RDB 发送给 slave，slave 会先写入本地磁盘，然后再从本地磁盘加载到内存中，

接着 master 会将内存中缓存的写命令发送到 slave，slave 也会同步这些数据。

slave node 如果跟 master node 有网络故障，断开了连接，会自动重连，连接之后 master node 仅会复制给 slave 部分缺少的数据。

所有的slave节点数据的复制和同步都由master节点来处理，会照成master节点压力太大，使用主从从结构来解决

## 分区

分区可以让Redis管理更大的内存，Redis将可以使用所有机器的内存。

### Redis分区实现方案

客户端分区就是在客户端就已经决定数据会被存储到哪个redis节点或者从哪个redis节点读取。

代理分区 意味着客户端将请求发送给代理，然后代理决定去哪个节点写数据或者读数据。

查询路由(Query routing) 的意思是客户端随机地请求任意一个redis实例，然后由Redis将请求转发给正确的Redis节点。

### Redis分区有什么缺点？

涉及多个key的操作通常不会被支持。

同时操作多个key,则不能使用Redis事务.

分区使用的粒度是key，不能使用一个非常长的排序key存储一个数据集（The partitioning granularity is the key, so it is not possible to shard a dataset with a single huge key like a very big sorted set）

当使用分区的时候，数据处理会非常复杂，例如为了备份你必须从不同的Redis实例和主机同时收集RDB / AOF文件。

分区时动态扩容或缩容可能非常复杂。

## 分布式问题

### Redis实现分布式锁

Redis为单进程单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对Redis的连接并不存在竞争关系Redis中可以使用SETNX命令实现分布式锁。

### 如何解决 Redis 的并发竞争 Key 问题

所谓 Redis 的并发竞争 Key 的问题也就是多个系统同时对一个 key 进行操作，但是最后执行的顺序和我们期望的顺序不同，这样也就导致了结果的不同！

推荐一种方案：分布式锁（zookeeper 和 redis 都可以实现分布式锁）。（如果不存在 Redis 的并发竞争 Key 问题，不要使用分布式锁，这样会影响性能）

基于zookeeper临时有序节点可以实现的分布式锁。大致思想为：每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。 判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。 当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。完成业务流程后，删除对应的子节点释放锁。

在实践中，当然是从以可靠性为主。所以首推Zookeeper。

### 什么是 RedLock

Redis 官方站提出了一种权威的基于 Redis 实现分布式锁的方式名叫 Redlock，此种方式比原先的单节点的方法更安全。它可以保证以下特性：

安全特性：互斥访问，即永远只有一个 client 能拿到锁

避免死锁：最终 client 都可能拿到锁，不会出现死锁的情况，即使原本锁住某资源的 client crash 了或者出现了网络分区

容错性：只要大部分 Redis 节点存活就可以正常提供服务

## 缓存异常

### 缓存雪崩

缓存雪崩是指缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决方案

1. 缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。

2. 一般并发量不是特别多的时候，使用最多的解决方案是加锁排队。

3. 给每一个缓存数据增加相应的缓存标记，记录缓存的是否失效，如果缓存标记失效，则更新数据缓存。

### 缓存穿透

缓存穿透

缓存穿透是指缓存和数据库中都没有的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决方案

1. 接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截；

2. 从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击

3. 采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力

### 缓存击穿

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力。和缓存雪崩不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

解决方案

设置热点数据永远不过期。

加互斥锁，互斥锁

### 缓存预热

缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

解决方案

直接写个缓存刷新页面，上线时手工操作一下；

数据量不大，可以在项目启动的时候自动进行加载；

定时刷新缓存；

### 缓存降级

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

\*\*缓存降级\*\*的最终目的是保证核心服务可用，即使是有损的。

## 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，包括内存快照和AOF日志文件，特别是不要启用内存快照做持久化。

2. 如果数据比较关键，某个Slave开启AOF备份数据，策略为每秒同步一次。

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

4. 尽量避免在压力较大的主库上增加从库

5. Master调用BGREWRITEAOF重写AOF文件，AOF在重写的时候会占大量的CPU和内存资源，导致服务load过高，出现短暂服务暂停现象。

6. 为了Master的稳定性，主从复制不要用图状结构，用单向链表结构更稳定，即主从关系为：Master<–Slave1<–Slave2<–Slave3…，这样的结构也方便解决单点故障问题，实现Slave对Master的替换，也即，如果Master挂了，可以立马启用Slave1做Master，其他不变。

\*\*缓存降级\*\*的最终目的是保证核心服务可用，即使是有损的。

# Linux

inux拥有所有这些组件：内核，shell和GUI，系统实用程序和应用程序。

## 常见目录说明：

/bin： 存放二进制可执行文件(ls,cat,mkdir等)，常用命令一般都在这里；

/etc： 存放系统管理和配置文件；

/home： 存放所有用户文件的根目录，是用户主目录的基点，比如用户user的主目录就是/home/user，可以用~user表示；

\*\*/usr \*\*： 用于存放系统应用程序；

/opt： 额外安装的可选应用程序包所放置的位置。一般情况下，我们可以把tomcat等都安装到这里；

/proc： 虚拟文件系统目录，是系统内存的映射。可直接访问这个目录来获取系统信息；

/root： 超级用户（系统管理员）的主目录（特权阶级o）；

/sbin: 存放二进制可执行文件，只有root才能访问。这里存放的是系统管理员使用的系统级别的管理命令和程序。如ifconfig等；

/dev： 用于存放设备文件；

/mnt： 系统管理员安装临时文件系统的安装点，系统提供这个目录是让用户临时挂载其他的文件系统；

/boot： 存放用于系统引导时使用的各种文件；

\*\*/lib \*\*： 存放着和系统运行相关的库文件 ；

/tmp： 用于存放各种临时文件，是公用的临时文件存储点；

/var： 用于存放运行时需要改变数据的文件，也是某些大文件的溢出区，比方说各种服务的日志文件（系统启动日志等。）等；

/lost+found： 这个目录平时是空的，系统非正常关机而留下“无家可归”的文件（windows下叫什么.chk）就在这里。

## 生产环境变慢的诊断思路

1. top命令查看整机性能

2. vmstat查看cpu性能

3. free命令查看内存

4. df查看硬盘使用情况

5. iostat命令查看磁盘的IO情况

6. ifstat命令查看网络的IO

## 生产环境出现CPU占用过高

1. 先用top命令找出CPU占比最高的

2. ps -ef或者jps命令定位是怎么样的一个后台程序

3. 定位到具体线程或代码行：：ps -mp 进程 -o THREAD,tid,time

1. -m 显示所有线程

2. -p pid进程使用cpu的时间

3. -o 该参数后是用户自定义格式

4. 将需要的线程ID转换为16进制英文要小写

5. jstack 进程ID|grep tid -A60

## 文件管理命令

### cat 命令

cat 命令用于连接文件并打印到标准输出设备上。

1. 一次显示整个文件:

2. 从键盘创建一个文件:

3. 将几个文件合并为一个文件:

### chmod 命令

Linux/Unix 的文件调用权限分为三级 : 文件拥有者、群组、其他。利用 chmod 可以控制文件如何被他人所调用。

用于改变 linux 系统文件或目录的访问权限。用它控制文件或目录的访问权限。该命令有两种用法。一种是包含字母和操作符表达式的文字设定法；另一种是包含数字的数字设定法。

权限范围：

u ：目录或者文件的当前的用户

g ：目录或者文件的当前的群组

o ：除了目录或者文件的当前用户或群组之外的用户或者群组

a ：所有的用户及群组

权限代号：

r ：读权限，用数字4表示

w ：写权限，用数字2表示

x ：执行权限，用数字1表示

### chown 命令

chown 将指定文件的拥有者改为指定的用户或组，用户可以是用户名或者用户 ID；组可以是组名或者组 ID；文件是以空格分开的要改变权限的文件列表，支持通配符

### cp 命令

将源文件复制至目标文件，或将多个源文件复制至目标目录。

-i 提示

-r 复制目录及目录内所有项目

-a 复制的文件与原文件时间一样

cp -ai a.txt test

### find 命令

用于在文件树中查找文件，并作出相应的处理。

（1）查找 48 小时内修改过的文件

find -atime -2

（2）在当前目录查找 以 .log 结尾的文件。 . 代表当前目录

find ./ -name '\*.log'

（3）查找 /opt 目录下 权限为 777 的文件

find /opt -perm 777

（4）查找大于 1K 的文件

find -size +1000c

### head 命令

head 用来显示档案的开头至标准输出中，默认 head 命令打印其相应文件的开头 10 行。

\*\*实例\*\*：

（1）显示 1.log 文件中前 20 行

head 1.log -n 20

（2）显示 1.log 文件前 20 字节

head -c 20 log2014.log

（3）显示 t.log最后 10 行

head -n -10 t.log

### less 命令

less 与 more 类似，但使用 less 可以随意浏览文件，而 more 仅能向前移动，却不能向后移动，而且 less 在查看之前不会加载整个文件。

### ln 命令

功能是为文件在另外一个位置建立一个同步的链接，当在不同目录需要该问题时，就不需要为每一个目录创建同样的文件，通过 ln 创建的链接（link）减少磁盘占用量。

软链接：

1.软链接，以路径的形式存在。类似于Windows操作系统中的快捷方式

2.软链接可以 跨文件系统 ，硬链接不可以

3.软链接可以对一个不存在的文件名进行链接

4.软链接可以对目录进行链接

硬链接:

1.硬链接，以文件副本的形式存在。但不占用实际空间。

2.不允许给目录创建硬链接

3.硬链接只有在同一个文件系统中才能创建

（1）给文件创建软链接，并显示操作信息 ln -sv source.log link.log

（2）给文件创建硬链接，并显示操作信息 ln -v source.log link1.log

（3）给目录创建软链接 ln -sv /opt/soft/test/test3 /opt/soft/test/test5

### more 命令

功能类似于 cat, more 会以一页一页的显示方便使用者逐页阅读，而最基本的指令就是按空白键（space）就往下一页显示，按 b 键就会往回（back）一页显示。

命令参数：

+n 从笫 n 行开始显示

-n 定义屏幕大小为n行

+/pattern 在每个档案显示前搜寻该字串（pattern），然后从该字串前两行之后开始显示

-c 从顶部清屏，然后显示

-d 提示“Press space to continue，’q’ to quit（按空格键继续，按q键退出）”，禁用响铃功能

-l 忽略Ctrl+l（换页）字符

-p 通过清除窗口而不是滚屏来对文件进行换页，与-c选项相似

-s 把连续的多个空行显示为一行

-u 把文件内容中的下画线去掉

常用操作命令：

Enter 向下 n 行，需要定义。默认为 1 行

Ctrl+F 向下滚动一屏

空格键 向下滚动一屏

Ctrl+B 返回上一屏

= 输出当前行的行号

:f 输出文件名和当前行的行号

V 调用vi编辑器

!命令 调用Shell，并执行命令

q 退出more

### mv 命令

移动文件或修改文件名，根据第二参数类型（如目录，则移动文件；如为文件则重命令该文件）。

（1）将文件 test.log 重命名为 test1.txt

mv test.log test1.txt

（2）将文件 log1.txt,log2.txt,log3.txt 移动到根的 test3 目录中

mv llog1.txt log2.txt log3.txt /test3

（3）将文件 file1 改名为 file2，如果 file2 已经存在，则询问是否覆盖

mv -i log1.txt log2.txt

（4）移动当前文件夹下的所有文件到上一级目录

mv \* ../

### rm 命令

删除一个目录中的一个或多个文件或目录，如果没有使用 -r 选项，则 rm 不会删除目录。如果使用 rm 来删除文件，通常仍可以将该文件恢复原状。

\*\*实例\*\*：

（1）删除任何 .log 文件，删除前逐一询问确认：

rm -i \*.log

（2）删除 test 子目录及子目录中所有档案删除，并且不用一一确认：

rm -rf test

（3）删除以 -f 开头的文件

rm -- -f\*

### tail 命令

用于显示指定文件末尾内容，不指定文件时，作为输入信息进行处理。常用查看日志文件。

\*\*常用参数\*\*：

-f 循环读取（常用于查看递增的日志文件）

-n<行数> 显示行数（从后向前）

### touch 命令

Linux touch命令用于修改文件或者目录的时间属性，包括存取时间和更改时间。

实例

使用指令"touch"修改文件"testfile"的时间属性为当前系统时间，输入如下命令：

$ touch testfile #修改文件的时间属性

首先，使用ls命令查看testfile文件的属性，如下所示：

$ ls -l testfile #查看文件的时间属性

执行指令"touch"修改文件属性以后，并再次查看该文件的时间属性，如下所示：

touch testfile #修改文件时间属性为当前系统时间

ls -l testfile #查看文件的时间属性

修改后文件的时间属性为当前系统时间

使用指令"touch"时，如果指定的文件不存在，则将创建一个新的空白文件。例如，在当前目录下，使用该指令创建一个空白文件"file"，输入如下命令：

$ touch file

### vim 命令

Vim是从 vi 发展出来的一个文本编辑器。代码补完、编译及错误跳转等方便编程的功能特别丰富，在程序员中被广泛使用。

基本上 vi/vim 共分为三种模式，分别是\*\*命令模式（Command mode）\*\*，\*\*输入模式（Insert mode）\\*\*和\\*\*底线命令模式（Last line mode）\*\*。

### whereis 命令

whereis 命令只能用于程序名的搜索，而且只搜索二进制文件（参数-b）、man说明文件（参数-m）和源代码文件（参数-s）。如果省略参数，则返回所有信息。whereis 及 locate 都是基于系统内建的数据库进行搜索，因此效率很高，而find则是遍历硬盘查找文件。

### which 命令

在 linux 要查找某个文件，但不知道放在哪里了，可以使用下面的一些命令来搜索：

which 查看可执行文件的位置。

whereis 查看文件的位置。

locate 配合数据库查看文件位置。

find 实际搜寻硬盘查询文件名称。

which 是在 PATH 就是指定的路径中，搜索某个系统命令的位置，并返回第一个搜索结果。使用 which 命令，就可以看到某个系统命令是否存在，以及执行的到底是哪一个位置的命令。

## 文档编辑命令

### grep 命令

强大的文本搜索命令，grep(Global Regular Expression Print) 全局正则表达式搜索。

（1）查找指定进程

ps -ef | grep svn

（2）查找指定进程个数

ps -ef | grep svn -c

（3）从文件中读取关键词

cat test1.txt | grep -f key.log

（4）从文件夹中递归查找以grep开头的行，并只列出文件

grep -lR '^grep' /tmp

（5）查找非x开关的行内容

grep '^[^x]' test.txt

（6）显示包含 ed 或者 at 字符的内容行

grep -E 'ed|at' test.txt

### wc 命令

wc(word count)功能为统计指定的文件中字节数、字数、行数，并将统计结果输出

\*\*命令参数\*\*：

-c 统计字节数

-l 统计行数

-m 统计字符数

-w 统计词数，一个字被定义为由空白、跳格或换行字符分隔的字符串

## 磁盘管理命令

### cd 命令

cd(changeDirectory) 命令 说明：切换当前目录至 dirName。

### df 命令（常用）

显示磁盘空间使用情况。获取硬盘被占用了多少空间，目前还剩下多少空间等信息，如果没有文件名被指定，则所有当前被挂载的文件系统的可用空间将被显示。默认情况下，磁盘空间将以 1KB 为单位进行显示，除非环境变量 POSIXLY\_CORRECT 被指定，那样将以512字节为单位进行显示：

-a 全部文件系统列表

-h 以方便阅读的方式显示信息（常用参数）

-i 显示inode信息

-k 区块为1024字节

-l 只显示本地磁盘

-T 列出文件系统类型

### du 命令

du 命令也是查看使用空间的，但是与 df 命令不同的是 Linux du 命令是对文件和目录磁盘使用的空间的查看：

-a 显示目录中所有文件大小

-k 以KB为单位显示文件大小

-m 以MB为单位显示文件大小

-g 以GB为单位显示文件大小

-h 以易读方式显示文件大小

-s 仅显示总计

-c或--total 除了显示个别目录或文件的大小外，同时也显示所有目录或文件的总和

### ls命令

就是 list 的缩写，通过 ls 命令不仅可以查看 linux 文件夹包含的文件，而且可以查看文件权限(包括目录、文件夹、文件权限)查看目录信息等等。

常用参数搭配：

ls -a 列出目录所有文件，包含以.开始的隐藏文件

ls -A 列出除.及..的其它文件

ls -r 反序排列

ls -t 以文件修改时间排序

ls -S 以文件大小排序

ls -h 以易读大小显示

ls -l 除了文件名之外，还将文件的权限、所有者、文件大小等信息详细列出来

### mkdir 命令

mkdir 命令用于创建文件夹。

可用选项：

- \*\*-m\*\*: 对新建目录设置存取权限，也可以用 chmod 命令设置;

- \*\*-p\*\*: 可以是一个路径名称。此时若路径中的某些目录尚不存在,加上此选项后，系统将自动建立好那些尚不在的目录，即一次可以建立多个目录。

### pwd 命令

pwd 命令用于查看当前工作目录路径。

### rmdir 命令

从一个目录中删除一个或多个子目录项，删除某目录时也必须具有对其父目录的写权限。

## 网络通讯命令

### ifconfig 命令

- ifconfig 用于查看和配置 Linux 系统的网络接口。

- 查看所有网络接口及其状态：`ifconfig -a` 。

- 使用 up 和 down 命令启动或停止某个接口：`ifconfig eth0 up` 和 `ifconfig eth0 down` 。

### iptables 命令

iptables ，是一个配置 Linux 内核防火墙的命令行工具。功能非常强大，对于我们开发来说，主要掌握如何开放端口即可。

### netstat 命令

Linux netstat命令用于显示网络状态。

利用netstat指令可让你得知整个Linux系统的网络情况

### ping 命令

Linux ping命令用于检测主机。

执行ping指令会使用ICMP传输协议，发出要求回应的信息，若远端主机的网络功能没有问题，就会回应该信息，因而得知该主机运作正常。

### telnet 命令

Linux telnet命令用于远端登入。

执行telnet指令开启终端机阶段作业，并登入远端主机。

## 系统管理命令

### date 命令

显示或设定系统的日期与时间。

（1）显示下一天

date +%Y%m%d --date="+1 day" //显示下一天的日期

（2）-d参数使用

date -d "nov 22" 今年的 11 月 22 日是星期三

date -d '2 weeks' 2周后的日期

date -d 'next monday' (下周一的日期)

date -d next-day +%Y%m%d（明天的日期）或者：date -d tomorrow +%Y%m%d

date -d last-day +%Y%m%d(昨天的日期) 或者：date -d yesterday +%Y%m%d

date -d last-month +%Y%m(上个月是几月)

date -d next-month +%Y%m(下个月是几月)

### free 命令（常用）

显示系统内存使用情况，包括物理内存、交互区内存(swap)和内核缓冲区内存。

常用-m参数

命令参数：

-b 以Byte显示内存使用情况

-k 以kb为单位显示内存使用情况

-m 以mb为单位显示内存使用情况

-g 以gb为单位显示内存使用情况

-s<间隔秒数> 持续显示内存

-t 显示内存使用总合

### kill 命令

发送指定的信号到相应进程。不指定型号将发送SIGTERM（15）终止指定进程。如果任无法终止该程序可用"-KILL" 参数，其发送的信号为SIGKILL(9) ，将强制结束进程，使用ps命令或者jobs 命令可以查看进程号。root用户将影响用户的进程，非root用户只能影响自己的进程。

### ps 命令

ps(process status)，用来查看当前运行的进程状态，一次性查看，如果需要动态连续结果使用 top

linux上进程有5种状态:

1. 运行(正在运行或在运行队列中等待)

2. 中断(休眠中, 受阻, 在等待某个条件的形成或接受到信号)

3. 不可中断(收到信号不唤醒和不可运行, 进程必须等待直到有中断发生)

4. 僵死(进程已终止, 但进程描述符存在, 直到父进程调用wait4()系统调用后释放)

5. 停止(进程收到SIGSTOP, SIGSTP, SIGTIN, SIGTOU信号后停止运行运行)

\*\*命令参数\*\*：

-A 显示所有进程

a 显示所有进程

-a 显示同一终端下所有进程

c 显示进程真实名称

e 显示环境变量

f 显示进程间的关系

r 显示当前终端运行的进程

-aux 显示所有包含其它使用的进程

### rpm 命令

Linux rpm 命令用于管理套件。

### top 命令（常用）

显示当前系统正在执行的进程的相关信息，包括进程 ID、内存占用率、CPU 占用率等，

99.9%id：空闲cpu百分比

load average: 0.00, 0.02, 0.05系统负载，即任务队列的平均长度。

三个数值分别为 1分钟、5分钟、15分钟前到现在的平均值。

\*\*常用参数\*\*：

-c 显示完整的进程命令

-s 保密模式

-p <进程号> 指定进程显示

-n <次数>循环显示次数

### yum 命令

yum（ Yellow dog Updater, Modified）是一个在Fedora和RedHat以及SUSE中的Shell前端软件包管理器。

### vmstat 命令

vmstat 5 5 【在5秒时间内进行5次采样】

\*\*字段说明：\*\*

Procs（进程）：

r: 运行队列中进程数量

b： 等待IO的进程数量

CPU（以百分比表示）：

us: 用户进程执行时间(user time)

sy: 系统进程执行时间(system time)

id: 空闲时间(包括IO等待时间),中央处理器的空闲时间 。以百分比表示。

wa: 等待IO时间

### iostat命令

iostat -xdk 2 3 查看设备使用率（%util）、响应时间（await

rrqm/s： 每秒进行 merge 的读操作数目.即 delta(rmerge)/s

wrqm/s： 每秒进行 merge 的写操作数目.即 delta(wmerge)/s

%util： 一秒中有百分之多少的时间用于 I/O

如果%util接近100%，说明产生的I/O请求太多，I/O系统已经满负荷

idle小于70% IO压力就较大了，一般读取速度有较多的wait。

## 备份压缩命令

### bzip2 命令

- 创建 `\*.bz2` 压缩文件：`bzip2 test.txt` 。

- 解压 `\*.bz2` 文件：`bzip2 -d test.txt.bz2` 。

### gzip 命令

- 创建一个 `\*.gz` 的压缩文件：`gzip test.txt` 。

- 解压 `\*.gz` 文件：`gzip -d test.txt.gz` 。

- 显示压缩的比率：`gzip -l \*.gz` 。

### tar 命令

用来压缩和解压文件。tar 本身不具有压缩功能，只具有打包功能，有关压缩及解压是调用其它的功能来完成。

弄清两个概念：打包和压缩。打包是指将一大堆文件或目录变成一个总的文件；压缩则是将一个大的文件通过一些压缩算法变成一个小文件

常用参数：

-c 建立新的压缩文件

-f 指定压缩文件

-r 添加文件到已经压缩文件包中

-u 添加改了和现有的文件到压缩包中

-x 从压缩包中抽取文件

-t 显示压缩文件中的内容

-z 支持gzip压缩

-j 支持bzip2压缩

-Z 支持compress解压文件

-v 显示操作过程

### unzip 命令

- 解压 `\*.zip` 文件：`unzip test.zip` 。

- 查看 `\*.zip` 文件的内容：`unzip -l jasper.zip` 。

## 服务类命令

### service(centos6)

注册在系统中的标准化层序

有方便统一的管理方式（常用的方法）

service 服务名 start

service 服务名 stop

service 服务名 restart

service 服务名 reload

service 服务名 status

查看服务的方法 /etc/init.d/服务名

通过chkconfig 命令设置自启动

查看服务 chkconfig --list|grep xxx

chkconfig --level 5 服务名 on

(on 是启动，off是关闭)

### systemctl(centos7)

注册在系统中的标准化程序

有方便统一的管理方式（常用的方法）

systemctl start 服务名(xxxx.service)

systemctl restart 服务名(xxxx.service)

systemctl stop 服务名(xxxx.service)

systemctl reload 服务名(xxxx.service)

systemctl status 服务名(xxxx.service)

查看服务的方法 /usr/lib/systemd/system

查看服务的命令

systemctl list-unit-files

systemctl --type service

通过systemctl命令设置自启动

自启动systemctl enable service\_name

不自启动systemctl disable serivice\_name

# Zookeeper

zookeeper，它是一个分布式服务框架，是Apache Hadoop 的一个子项目，它主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。

上面的解释有点抽象，简单来说zookeeper=文件系统+监听通知机制。

## znode：

- \*\*PERSISTENT-持久化目录节点\*\*

客户端与zookeeper断开连接后，该节点依旧存在

- \*\*PERSISTENT\_SEQUENTIAL-持久化顺序编号目录节点\*\*

客户端与zookeeper断开连接后，该节点依旧存在，只是Zookeeper给该节点名称进行顺序编号

- \*\*EPHEMERAL-临时目录节点\*\*

客户端与zookeeper断开连接后，该节点被删除

- \*\*EPHEMERAL\_SEQUENTIAL-临时顺序编号目录节点\*\*

客户端与zookeeper断开连接后，该节点被删除，只是Zookeeper给该节点名称进行顺序编号

监听通知机制

客户端注册监听它关心的目录节点，当目录节点发生变化（数据改变、被删除、子目录节点增加删除）时，zookeeper会通知客户端。

就这么简单，下面我们看看Zookeeper能做点什么呢？

## Zookeeper能做什么

zookeeper功能非常强大，可以实现诸如分布式应用配置管理、统一命名服务、状态同步服务、集群管理等功能，我们这里拿比较简单的分布式应用配置管理为例来说明。

假设我们的程序是分布式部署在多台机器上，如果我们要改变程序的配置文件，需要逐台机器去修改，非常麻烦，现在把这些配置全部放到zookeeper上去，保存在 zookeeper 的某个目录节点中，然后所有相关应用程序对这个目录节点进行监听，一旦配置信息发生变化，每个应用程序就会收到 zookeeper 的通知，然后从 zookeeper 获取新的配置信息应用到系统中。

## 特点

1）Zookeeper：一个领导者（leader），多个跟随者（follower）组成的集群。

2）Leader负责进行投票的发起和决议，更新系统状态

3）Follower用于接收客户请求并向客户端返回结果，在选举Leader过程中参与投票

4）集群中只要有半数以上节点存活，Zookeeper集群就能正常服务。

5）全局数据一致：每个server保存一份相同的数据副本，client无论连接到哪个server，数据都是一致的。

6）更新请求顺序进行，来自同一个client的更新请求按其发送顺序依次执行。

7）数据更新原子性，一次数据更新要么成功，要么失败。

8）实时性，在一定时间范围内，client能读到最新数据。

选举机制

1）半数机制：集群中半数以上机器存活，集群可用。所以zookeeper适合装在奇数台机器上。

2）Zookeeper虽然在配置文件中并没有指定master和slave。但是，zookeeper工作时，是有一个节点为leader，其他则为follower，Leader是通过内部的选举机制临时产生的

假设有五台服务器组成的zookeeper集群，它们的id从1-5，同时它们都是最新启动的，也就是没有历史数据，在存放数据量这一点上，都是一样的。假设这些服务器依序启动，来看看会发生什么。

（1）服务器1启动，此时只有它一台服务器启动了，它发出去的报没有任何响应，所以它的选举状态一直是LOOKING状态。

（2）服务器2启动，它与最开始启动的服务器1进行通信，互相交换自己的选举结果，由于两者都没有历史数据，所以id值较大的服务器2胜出，但是由于没有达到超过半数以上的服务器都同意选举它(这个例子中的半数以上是3)，所以服务器1、2还是继续保持LOOKING状态。

（3）服务器3启动，根据前面的理论分析，服务器3成为服务器1、2、3中的老大，而与上面不同的是，此时有三台服务器选举了它，所以它成为了这次选举的leader。

（4）服务器4启动，根据前面的分析，理论上服务器4应该是服务器1、2、3、4中最大的，但是由于前面已经有半数以上的服务器选举了服务器3，所以它只能接收当小弟的命了。

（5）服务器5启动，同4一样当小弟。

## 服务器状态

服务器具有四种状态，分别是LOOKING、FOLLOWING、LEADING、OBSERVING。

- \*\*LOOKING\*\*：寻找Leader状态。当服务器处于该状态时，它会认为当前集群中没有Leader，因此需要进入Leader选举状态。

- \*\*FOLLOWING\*\*：跟随者状态。表明当前服务器角色是Follower。

- \*\*LEADING\*\*：领导者状态。表明当前服务器角色是Leader。

- \*\*OBSERVING\*\*：观察者状态。表明当前服务器角色是Observer。

## 常用命令

- \1. 启动ZK服务: sh bin/zkServer.sh start

- \2. 查看ZK服务状态: sh bin/zkServer.sh status

- \3. 停止ZK服务: sh bin/zkServer.sh stop

- \4. 重启ZK服务: sh bin/zkServer.sh restart

客户端命令：zkCli.sh -server 127.0.0.1:2181

- \1. 显示根目录下、文件： ls / 使用 ls 命令来查看当前 ZooKeeper 中所包含的内容

- \2. 显示根目录下、文件： ls2 / 查看当前节点数据并能看到更新次数等数据

- \3. 创建文件，并设置初始内容： create /zk "test" 创建一个新的 znode节点“ zk ”以及与它关联的字符串

- \4. 获取文件内容： get /zk 确认 znode 是否包含我们所创建的字符串

- \5. 修改文件内容： set /zk "zkbak" 对 zk 所关联的字符串进行设置

- \6. 删除文件： delete /zk 将刚才创建的 znode 删除

- \7. 退出客户端： quit

- \8. 帮助命令： help

# Negix

# Tomcat

# Netty

# RabbitMQ

# 常用框架源码

## Mybatis

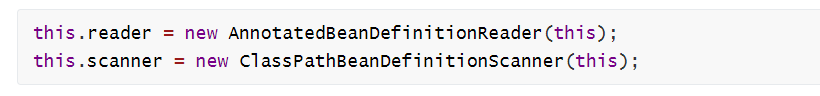
## Spring

### Spring IOC容器初始化：

以下源码分析是根据spring-context：5.1.8：RELEASE版本进行分析的，欢迎参考学习

#### this（）：

调用父类GenericApplicationContext构造函数，构建beanFactory （newDefaultListableBeanFactory（））



初始化：注解bean的定义读取器

获取bean工厂

DefaultListableBeanFactory beanFactory = unwrapDefaultListableBeanFactory(registry);

bean的定义Set集合

Set<BeanDefinitionHolder> beanDefs = new LinkedHashSet<>(8);

检查各个组件是否已经注册过了，没有注册就将其放入beandefs中去。

ConfigurationClassPostProcessor//解析@Configuration配置bean定义后置处理器，在bean定义完成之后执行，因为实现的是BeanDefinitionRegistryPostProcessor

DefaultEventListenerFactory//注册事件监听器工厂

EventListenerMethodProcessor//处理监听方法的注解解析器

AutowiredAnnotationBeanPostProcessor//解析Autowired

CommonAnnotationBeanPostProcessor//解析JSR规范的

初始化：classPath的定义扫描器 用于扫描注解的一些bean

#### register(annotatedClasses);

注册配置类：将配置类注册到容器中去，可以传入多个配置类依次进行遍历注册。

AnnotatedGenericBeanDefinition abd = new AnnotatedGenericBeanDefinition(annotatedClass);

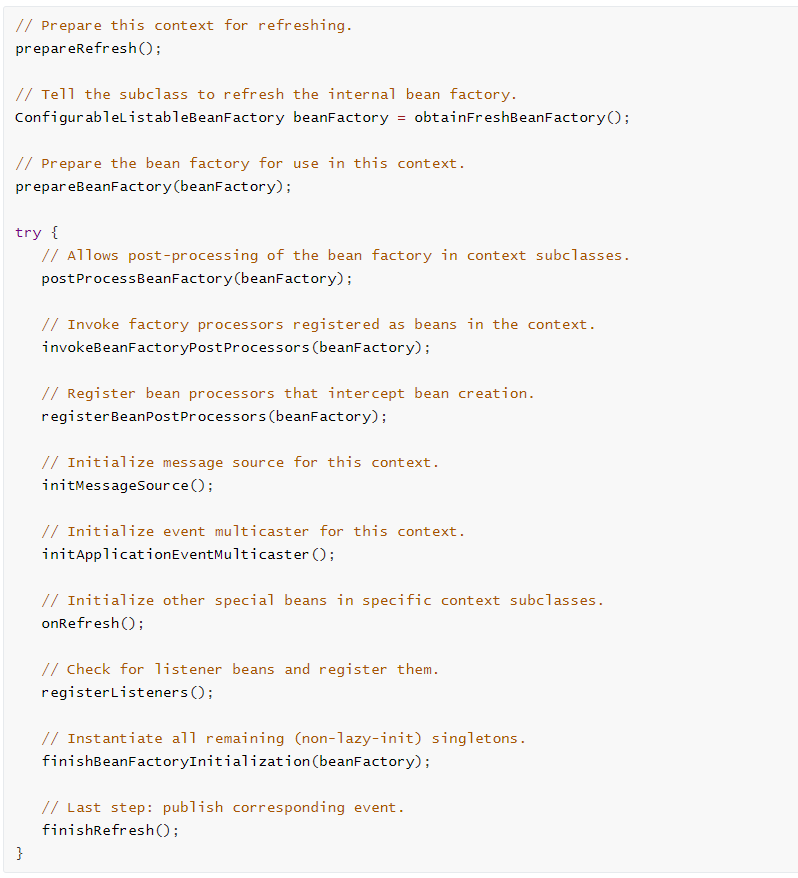
设置bean定义的属性：是否懒加载，Bean初始化销毁的顺序、是否首选、设置依赖、描述等等

BeanDefinitionHolder definitionHolder = new BeanDefinitionHolder(abd, beanName);

最后将definitionHolder注册到容器中去。

#### refresh()【创建刷新】（重点工作）;

源码如下：



#### 1、prepareRefresh()

刷新前的预处理;

1）、initPropertySources()

​ 初始化一些属性设置;子类自定义个性化的属性设置方法；空方法需要子类继承实现。

2）、getEnvironment().validateRequiredProperties();

​ 检验属性的合法等

3）、earlyApplicationEvents= new LinkedHashSet<ApplicationEvent>();

​ 保存容器中的一些早期的事件；

#### 2、obtainFreshBeanFactory();

获取BeanFactory；

1）、refreshBeanFactory();刷新 只允许bean工厂刷新一次，设置id；

2）、getBeanFactory();返回刚才GenericApplicationContext创建的BeanFactory对象；

3）、将创建的BeanFactory【DefaultListableBeanFactory】返回；

#### 3、prepareBeanFactory(beanFactory);

BeanFactory的预准备工作（BeanFactory进行一些设置）；

1）、设置BeanFactory的类加载器、支持表达式解析器...

2）、添加感知回调的后置处理器，BeanPostProcessor【ApplicationContextAwareProcessor】

3）、设置忽略的感知自动装配的接口

EnvironmentAware、EmbeddedValueResolverAware、ResourceLoaderAware、ApplicationEventPublisherAware、MessageSourceAware、ApplicationContextAware；

4）、注册可以解析的自动装配；我们能直接在任何组件中自动注入：

BeanFactory、ResourceLoader、ApplicationEventPublisher、ApplicationContext

5）、添加早期BeanPostProcessor，用于发现内部bean

ApplicationListenerDetector

6）、添加编译时的AspectJ切面代码织入；后置处理器

LoadTimeWeaverAwareProcessor

7）、给BeanFactory中注册一些环境的组件；

environment【ConfigurableEnvironment】、

systemProperties【Map<String, Object>】、

systemEnvironment【Map<String, Object>】

#### 4、postProcessBeanFactory(beanFactory);

BeanFactory准备工作完成后进行的后置处理工作；

1）、子类通过重写这个方法来在BeanFactory创建并预准备完成以后做进一步的设置，可以给beanfactory添加一些自己想要实现的bean工厂后置处理器。

#### 5、invokeBeanFactoryPostProcessors(beanFactory);

执行BeanFactoryPostProcessor的方法；执行bean工厂的后置处理器

BeanFactoryPostProcessor：BeanFactory的后置处理器。在BeanFactory标准初始化之后执行的；

两个接口：BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor

5.1先执行BeanDefinitionRegistryPostProcessor

1）、获取所有的BeanDefinitionRegistryPostProcessor；

String[] postProcessorNames =

beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);

2）、看先执行实现了PriorityOrdered优先级接口的BeanDefinitionRegistryPostProcessor、

3）、在执行实现了Ordered顺序接口的BeanDefinitionRegistryPostProcessor；

4）、最后执行没有实现任何优先级或者是顺序接口的BeanDefinitionRegistryPostProcessors；

5.2再执行BeanFactoryPostProcessor的方法

1）、获取所有的BeanFactoryPostProcessor

2）、看先执行实现了PriorityOrdered优先级接口的BeanFactoryPostProcessor、

3）、在执行实现了Ordered顺序接口的BeanFactoryPostProcessor；

4）、最后执行没有实现任何优先级或者是顺序接口的BeanFactoryPostProcessor；

检测LoadTimeWeaver，并准备代码织入，注册一个LoadTimeWeaver的bean后置处理器

#### 6、registerBeanPostProcessors(beanFactory);

注册BeanPostProcessor（Bean的后置处理器）【 intercept bean creation】

​ 不同接口类型的BeanPostProcessor；在Bean创建前后的执行时机是不一样的

​ BeanPostProcessor、

​ DestructionAwareBeanPostProcessor、

​ InstantiationAwareBeanPostProcessor、

​ SmartInstantiationAwareBeanPostProcessor、

​ MergedBeanDefinitionPostProcessor【internalPostProcessors】、

1）、获取所有的 BeanPostProcessor;后置处理器都默认可以通过PriorityOrdered、Ordered接口来执行优先级

String[] postProcessorNames = beanFactory.getBeanNamesForType(BeanPostProcessor.class, true, false);

2）、注册一个BeanPostProcessorChecker

BeanPostProcessorChecker在一个Bean不能被所有BeanPostProcessor处理时，会打印信息。

3）、对所有的bean后置处理器进行分类排序

2）、先注册PriorityOrdered优先级接口的BeanPostProcessor；

​ 把每一个BeanPostProcessor；添加到BeanFactory中

​ beanFactory.addBeanPostProcessor(postProcessor);

3）、再注册Ordered接口的

4）、最后注册没有实现任何优先级接口的

5）、最终注册重新注册所有的内部bean后置处理器；

6）、重新注册一个ApplicationListenerDetector；

用于发现内部bean是否是一个ApplicationListener

#### 7、initMessageSource();

初始化MessageSource组件（做国际化功能；消息绑定，消息解析）；MessageSource：取出国际化配置文件中的某个key的值；能按照区域信息获取；

1）、获取BeanFactory

2）、看容器中是否有id为messageSource的，类型是MessageSource的组件

如果有赋值给messageSource，如果没有自己创建一个DelegatingMessageSource；

3）、把创建好的MessageSource注册在容器中，以后获取国际化配置文件的值的时候，可以自动注入MessageSource；

#### 8、initApplicationEventMulticaster();

初始化应用事件派发器（多播器）；

1）、获取BeanFactory

2）、从BeanFactory中获取applicationEventMulticaster的ApplicationEventMulticaster；

3）、如果上一步没有配置；创建一个SimpleApplicationEventMulticaster

this.applicationEventMulticaster = new SimpleApplicationEventMulticaster(beanFactory);

beanFactory.registerSingleton(APPLICATION\_EVENT\_MULTICASTER\_BEAN\_NAME, this.applicationEventMulticaster);

4）、将创建的ApplicationEventMulticaster添加到BeanFactory中，以后其他组件直接自动注入

#### 9、onRefresh();

留给子类扩展，子类重写这个方法，在单例bean被初始化之前调用；

#### 10、registerListeners();

检查侦听器bean并注册它们

1、首先注册静态指定的侦听器。

2、从容器中拿到所有的ApplicationListener，不要在这里初始化factorybean:我们需要保留所有常规bean未初始化，让后处理器应用到它们!将每个监听器添加到事件派发器中；

getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);

3、派发之前步骤产生的事件；

#### 11、finishBeanFactoryInitialization(beanFactory);【重点步骤】

初始化所有剩下的（非懒加载）单实例bean；

1、bean设置类型转换器（conversion service）、添加默认的值解析器（EmbeddedValueResolver）冻结所有的bean定义，不让修改了，等前置操作不进行细致分析了

2、beanFactory.preInstantiateSingletons();初始化后剩下的单实例bean

1）、获取容器中的所有beanNames，依次进行初始化和创建对象

List<String> beanNames = new ArrayList<>(this.beanDefinitionNames);

​ 2）、遍历获取Bean的定义信息；RootBeanDefinition

RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);

​ 3）、不是抽象的，是单实例的，不是懒加载的bean进入下面步骤；

​ 判断是否是工厂Bean；是否是实现FactoryBean接口的Bean；

利用getBean(beanName);创建对象，以下是调用doGetBean(name, null, null, false);方法

3.1、先获取缓存中保存的单实例Bean。如果能获取到说明这个Bean之前被创建过（所有创建过的单实例Bean都会被缓存起来）

Object sharedInstance = getSingleton(beanName);

如果获取到的实例不为空但是args为空，说明正在该bean还没有初始化完，存在循环引用，调用getObjectForBeanInstance方法获取bean；从factoryBeanObjectCache中返回一个bean出去

private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);//一级缓存，存放已经创建好的bean，先从这里获取

private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);//早期的单实例对象，提前曝光的单例对象的cache，存放原始的 bean 对象（尚未填充属性），用于解决循环依赖，正在创建中的，再从这里获取

private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>(16);//单实例工厂的缓存，最后从这里获取，单例对象工厂的cache，存放 bean 工厂对象，用于解决循环依赖，如果获取到了就把对象移动到二级缓存中去。

//以上三个缓存中都获取不到说明bean没有被创建

private final Set<String> singletonsCurrentlyInCreation =

Collections.newSetFromMap(new ConcurrentHashMap<>(16));//正在创建中的bean都会存在这个map里面

private final Set<String> registeredSingletons = new LinkedHashSet<>(256);//已经注册好的单实例会放到其中去

//FactoryBeanRegistrySupport类中的缓存

private final Map<String, Object> factoryBeanObjectCache = new ConcurrentHashMap<>(16);//被工厂bean创建的单实例对象，Spring注册的 bean是 FactoryBean的话、这个 FactoryBean存放在三级缓存中、FactoryBean产生的 bean缓存在这里

​3.2、缓存中获取不到，开始Bean的创建对象流程；判断当前bean是否正在创建中，

if (isPrototypeCurrentlyInCreation(beanName)) {

throw new BeanCurrentlyInCreationException(beanName);//重复创建会抛出异常，

}

if (!typeCheckOnly) {

markBeanAsCreated(beanName);//标记这个bean正在创建中

}

​3.3、获取Bean的定义信息；

final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);

3.4、获取当前Bean依赖的其他Bean;如果有按照getBean()把依赖的Bean先创建出来；

String[] dependsOn = mbd.getDependsOn();

3.5、判断是否单实例bean，启动单实例Bean的创建流程；（我这里只分析一下单实例bean，多实例的bean是不被ioc容器管理的）

​执行getSingleton（），里面执行createBean(beanName, mbd, args);方法，

3.5.1、复制bean的定义信息和准备方法覆盖

​ 3.5.2、给BeanPostProcessors一个返回目标实例代理对象的机会

// Give BeanPostProcessors a chance to return a proxy instead of the target bean instance.

Object bean = resolveBeforeInstantiation(beanName, mbdToUse);

if (bean != null) {

return bean;

}

​ 3.5.2.1、让BeanPostProcessor先拦截返回代理对象；InstantiationAwareBeanPostProcessor：提前执行；

​ 先触发：postProcessBeforeInstantiation()；

​ 如果有返回值：触发postProcessAfterInitialization()；

bean = applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);

if (bean != null) {

bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);

}

3.5.3、如果获取到的代理对象为null；以下是执行doCreateBean（）方法，

Object beanInstance = doCreateBean(beanName, mbdToUse, args);//创建Bean

=========================================================================================================================

​ 3.5.3.1、获取实例包装对象，先从factoryBeanInstanceCache中删除该bean，未完成的FactoryBean实例的缓存。获取instanceWrapper

if (mbd.isSingleton()) {

instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);

}

if (instanceWrapper == null) {

instanceWrapper = createBeanInstance(beanName, mbd, args);

}

​ 3.5.3.2、applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);

​ 后置处理器处理bean定义信息 调用MergedBeanDefinitionPostProcessor的postProcessMergedBeanDefinition(mbd, beanType, beanName);

​ 3.5.3.3、快速缓存单例，以便能够解决循环引用

if (!this.singletonObjects.containsKey(beanName)) {

this.singletonFactories.put(beanName, singletonFactory);

this.earlySingletonObjects.remove(beanName);

this.registeredSingletons.add(beanName);

}

​ 3.5.3.4、【Bean属性赋值】

populateBean(beanName, mbd, instanceWrapper);

1）、拿到InstantiationAwareBeanPostProcessor后置处理器；

​ postProcessAfterInstantiation()；

​ 2）、拿到InstantiationAwareBeanPostProcessor后置处理器；

​ postProcessPropertyValues()；

​ 3）、应用Bean属性的值；为属性利用setter方法等进行赋值；

​ applyPropertyValues(beanName, mbd, bw, pvs);

​ 3.5.3.5、【Bean初始化】initializeBean(beanName, exposedObject, mbd);

​ 1）、【执行Aware接口方法】invokeAwareMethods(beanName, bean);执行xxxAware接口的方法

​ BeanNameAware\BeanClassLoaderAware\BeanFactoryAware

​ 2）、【执行后置处理器初始化之前】applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);

​ BeanPostProcessor.postProcessBeforeInitialization（）;

​ 3）、【执行初始化方法】invokeInitMethods(beanName, wrappedBean, mbd);

​ 1）、是否是InitializingBean接口的实现；执行接口规定的初始化；

​ 2）、是否自定义初始化方法；

​ 4）、【执行后置处理器初始化之后】applyBeanPostProcessorsAfterInitialization

​ BeanPostProcessor.postProcessAfterInitialization()；

​ 5）、返回包装weappedBean

​ 3.5.3.6、注册Bean的销毁方法；

registerDisposableBeanIfNecessary(beanName, bean, mbd);

======================================================================

3.6、将创建的Bean添加到缓存中singletonObjects；

synchronized (this.singletonObjects) {

this.singletonObjects.put(beanName, singletonObject);//单例对象的缓存 Map<String, Object>

this.singletonFactories.remove(beanName);//单例工厂的缓存，从三级缓存中删除。

this.earlySingletonObjects.remove(beanName);//早期单例对象的缓存，二级缓存中也删除。

this.registeredSingletons.add(beanName);//一组已注册的单例对象，按注册顺序包含bean名称。Set<String>

}

4）、所有Bean都利用getBean创建完成以后；触发初始化后回调

​ 检查所有的Bean是否是SmartInitializingSingleton接口的；如果是；就执行afterSingletonsInstantiated()；

12、finishRefresh();

完成BeanFactory的初始化创建工作；IOC容器就创建完成；

1）、clearResourceCaches();清除资源缓存

2）、initLifecycleProcessor();初始化和生命周期有关的后置处理器；LifecycleProcessor

默认从容器中找是否有lifecycleProcessor的组件【LifecycleProcessor】；

如果没有new DefaultLifecycleProcessor();加入到容器；

DefaultLifecycleProcessor defaultProcessor = new DefaultLifecycleProcessor();

defaultProcessor.setBeanFactory(beanFactory);

this.lifecycleProcessor = defaultProcessor;

3）、 getLifecycleProcessor().onRefresh();

拿到前面定义的生命周期处理器（BeanFactory）；回调onRefresh()；

4）、publishEvent(new ContextRefreshedEvent(this));发布容器刷新完成事件；

5）、liveBeansView.registerApplicationContext(this);java管理拓展；

### 总结

1）、Spring容器在启动的时候，先会保存所有注册进来的Bean的定义信息；

1）、xml注册bean；<bean>

2）、注解注册Bean；@Service、@Component、@Bean、xxx

2）、Spring容器会合适的时机创建这些Bean

1）、用到这个bean的时候；利用getBean创建bean；创建好以后保存在容器中；

2）、统一创建剩下所有的bean的时候；finishBeanFactoryInitialization()；

3）、后置处理器；BeanPostProcessor

1）、每一个bean创建完成，都会使用各种后置处理器进行处理；来增强bean的功能；

AutowiredAnnotationBeanPostProcessor:处理自动注入

AnnotationAwareAspectJAutoProxyCreator:来做AOP功能；

xxx....

增强的功能注解：

AsyncAnnotationBeanPostProcessor

....

4）、事件驱动模型；

ApplicationListener；事件监听；

ApplicationEventMulticaster；事件派发：

### 扩展原理：

#### BeanPostProcessor：

接口：bean后置处理器，bean创建对象初始化前后进行拦截工作的

1、BeanFactoryPostProcessor：beanFactory的后置处理器；

在BeanFactory标准初始化之后调用，来定制和修改BeanFactory的内容；

所有的bean定义已经保存加载到beanFactory，但是bean的实例还未创建

#### BeanFactoryPostProcessor

接口bean工厂的后置处理器

1)、ioc容器创建对象

2)、invokeBeanFactoryPostProcessors(beanFactory);

如何找到所有的BeanFactoryPostProcessor并执行他们的方法；

1）、直接在BeanFactory中找到所有类型是BeanFactoryPostProcessor的组件，并执行他们的方法

2）、在初始化创建其他组件前面执行

#### BeanDefinitionRegistryPostrocessor

继承于 BeanFactoryPostProcessor

postProcessBeanDefinitionRegistry();

在所有bean定义信息将要被加载，bean实例还未创建的；

优先于BeanFactoryPostProcessor执行；

利用BeanDefinitionRegistryPostProcessor给容器中再额外添加一些组件；

原理：

1）、ioc创建对象

2）、refresh()-》invokeBeanFactoryPostProcessors(beanFactory);

3）、从容器中获取到所有的BeanDefinitionRegistryPostProcessor组件。

1、依次触发所有的postProcessBeanDefinitionRegistry()方法

2、再来触发postProcessBeanFactory()方法BeanFactoryPostProcessor；

4）、再来从容器中找到BeanFactoryPostProcessor组件；然后依次触发postProcessBeanFactory()方法

#### ApplicationListener：

监听容器中发布的事件。事件驱动模型开发；

public interface ApplicationListener<E extends ApplicationEvent>

监听 ApplicationEvent 及其下面的子事件；

步骤：

1）、写一个监听器（ApplicationListener实现类）来监听某个事件（ApplicationEvent及其子类）

@EventListener;

原理：使用EventListenerMethodProcessor处理器来解析方法上的@EventListener；

2）、把监听器加入到容器；

3）、只要容器中有相关事件的发布，我们就能监听到这个事件；

ContextRefreshedEvent：容器刷新完成（所有bean都完全创建）会发布这个事件；

ContextClosedEvent：关闭容器会发布这个事件；

4）、发布一个事件：

applicationContext.publishEvent()；

### 原理：

ContextRefreshedEvent、IOCTest\_Ext$1[source=我发布的时间]、ContextClosedEvent；

1）、ContextRefreshedEvent事件：

1）、容器创建对象：refresh()；

2）、finishRefresh();容器刷新完成会发布ContextRefreshedEvent事件

2）、自己发布事件；

3）、容器关闭会发布ContextClosedEvent；

【事件发布流程】：

3）、publishEvent(new ContextRefreshedEvent(this));

1）、获取事件的多播器（派发器）：getApplicationEventMulticaster()

2）、multicastEvent派发事件：

3）、获取到所有的ApplicationListener；

for (final ApplicationListener<?> listener : getApplicationListeners(event, type)) {

1）、如果有Executor，可以支持使用Executor进行异步派发；

Executor executor = getTaskExecutor();

2）、否则，同步的方式直接执行listener方法；invokeListener(listener, event);

拿到listener回调onApplicationEvent方法；

【事件多播器（派发器）】

1）、容器创建对象：refresh();

2）、initApplicationEventMulticaster();初始化ApplicationEventMulticaster；

1）、先去容器中找有没有id=“applicationEventMulticaster”的组件；

2）、如果没有this.applicationEventMulticaster = new SimpleApplicationEventMulticaster(beanFactory);

并且加入到容器中，我们就可以在其他组件要派发事件，自动注入这个applicationEventMulticaster；

【容器中有哪些监听器】

1）、容器创建对象：refresh();

2）、registerListeners();

从容器中拿到所有的监听器，把他们注册到applicationEventMulticaster中；

String[] listenerBeanNames = getBeanNamesForType(ApplicationListener.class, true, false);

//将listener注册到ApplicationEventMulticaster中

getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);

#### SmartInitializingSingleton 原理：

在spring容器管理的所有单例对象（非懒加载对象）初始化完成之后调用的回调接口。智能初始化单例

afterSingletonsInstantiated();

1）、ioc容器创建对象并refresh()；

2）、finishBeanFactoryInitialization(beanFactory);初始化剩下的单实例bean；

1）、先创建所有的单实例bean；getBean();

2）、获取所有创建好的单实例bean，判断是否是SmartInitializingSingleton类型的；

如果是就调用afterSingletonsInstantiated();

## springMVC

## springboot

# 其他拓展思维

## 使用有限内存对大型数据文件排序

切分：从目标数据文件中读取数据，读取一定数量后对读取到的数据进行排序，并生成临时排序文件，重复此过程，将原始数据文件分割为若干个已排序的数据文件

合并：根据上一阶段得到的分组文件数量，如果内存不足以一次创建所有文件的指针，则需要进行多次合并。合并时，将若干个临时数据文件合并为更大的数据文件，使用归并排序思想，使用优先级队列辅助。重复此过程直到生成的临时数据文件个数足够少。

## 不使用其他变量交换两个数

1. 加减法

x = x + y;

y = x - y;

x = x - y;//已改正

1. 异或

x = x ^ y;

y = x ^ y;

x = x ^ y;

# 自我介绍

## 导出服务架构分析

ExportDataServiceController export方法用于导出生成文件；

### 第一步

对参数进行校验，不符合条件返回错误信息

### 第二步

通过前台参数构建导出数据（ExportData -> List<IDataAccessor>/List<ExportTree>）

通过前台项目的id来获取到对应的excel模板和每个excel需要获取的数据信息，有多少的sheet需要进行填充数据等等

ExportData主要包括两部分：

第一部分是：List<IDataAccessor> AccessorList

AbstractAccessor-》》ExportTree / ExportItem

是一个继承callable的类，进行工作获取远程接口数据

ExportItem-》》IDataBinder / ISheetData

导出条目配置对象，一个sheet对应一个Item

​Accessor里面存放的有导出树对象（模板名称用于获取模板，导出参数，工件类型等，该往哪个sheet写数据，sheet的数据，获取数据格式化后存入，从哪个接口获取url）是否导出多个excel，和条目配置对象（sheet页的中英文名称，绑定数据的执行器，获取数据的urlkey）将每个树的item都放到其中，启动线程的时候就是获取这个列表来决定开启的线程

​ 第二部分是：List<ExportTree> exportTreeList

ExportTree ->> List\*\*<\*\*ExportCategory> / ExcelTemplate

一个excel对应一个导出tree

ExportCategory ->> List\*\*<\*\*ExportItem\*\*>\*\*

一组sheet页对应一个category

ExportItem ->> IDataBinder/ISheetData

一个sheet对应一个Item，里面有绑定数据的binder和通过accessor获取到的IsheetData

每一个导出的excel都有对应的一个导出树对象（模板名称用于获取模板，导出参数，工件类型等）是否导出多个excel

### 第三步

构件好导出树之后就可以根据导出树的数据调用多线程任务去启动获取数据的服务

初始化好线程池：50个线程数，LinkedBlockingQueue阻塞队列

executorPool.submit(accessor)提交每个accessor

IDataAccessor继承Callable，可被调用获取数据，基本上每个sheet页都需要一个Accessor去对应的模块获取数据。当然也封装了一些通用的模板去获取数据，例如有些sheet页面只有一个表格，那么这种sheet也就可以封装一个统一的accessor去获取表格数据，表格的样式都统一，动态根据配置获取起始行列来生成sheet页。

定义抽象类AbstractAccessor实现IDataAccessor公共方法实现在抽象类，项目中所有的Accessor都继承抽象类，实现特有的从不同的url获取不同格式的数据功能，这就是使用了java设计模式中的模板模式

accessor的call方法获取到的的是一个ISHeetData接口，IsheetData有很多实现类（里面的信息包括起始行，起始列，sheet表的样式信息，表头信息表格标题等等，还有一些仅仅是一个起始行起始列和一个Workbook的excel文件，存放图片列表的sheet）是每个sheet页的数据ISheetData放到Accessor

### 第四步

获取完数据所有的数据我们的导出树就已经完整了，里面已经有了数据

遍历里面的ExportItem把相关的数据样式还有位置信息等等都拿出来，插入到我们的模板中

AbstractBinder用于插数据到excel中，bind（）方法传入的参数是ISheetData。单个表格插入，图片列表插入，自定义数据插入的数据绑定器。

绑定完数据之后就把生成好的LLD文档和ZTP文档放到服务器的临时目录下，等待用户调用下载

使用aspose.cells操作