# Java基础相关

## 基本数据类型



### 8种基本类型的包装类和常量池

Java 基本类型的包装类的大部分都实现了常量池技术，即 Byte,Short,Integer,Long,Character,Boolean；前面 4 种包装类默认创建了数值[-128，127] 的相应类型的缓存数据，Character创建了数值在[0,127]范围的缓存数据，Boolean 直接返回True Or False。如果超出对应范围仍然会去创建新的对象。

### BigDecimal

浮点数之间的等值判断，基本数据类型不能用==来比较，包装数据类型不能用 equals 来判断。

BigDecimal 的大小比较

a. compareTo(b) : 返回 -1 表示小于，0 表示 等于， 1表示 大于。

BigDecimal 保留几位小数

通过 setScale方法设置保留几位小数以及保留规则。

我们在使用BigDecimal时，为了防止精度丢失，推荐使用它的 BigDecimal(String) 构造方法来创建对象。

BigDecimal 主要用来操作（大）浮点数，BigInteger 主要用来操作大整数（超过 long 类型）。

BigDecimal 的实现利用到了 BigInteger, 所不同的是 BigDecimal 加入了小数位的概念

### 基本数据类型与包装数据类型的使用标准

Reference:《阿里巴巴Java开发手册》

【强制】所有的 POJO 类属性必须使用包装数据类型。

【强制】RPC 方法的返回值和参数必须使用包装数据类型。

【推荐】所有的局部变量使用基本数据类型。

比如我们如果自定义了一个Student类,其中有一个属性是成绩score,如果用Integer而不用int定义,一次考试,学生可能没考,值是null,也可能考了,但考了0分,值是0,这两个表达的状态明显不一样.

说明 :POJO 类属性没有初值是提醒使用者在需要使用时，必须自己显式地进行赋值，任何 NPE 问题，或者入库检查，都由使用者来保证。

正例 : 数据库的查询结果可能是 null，因为自动拆箱，用基本数据类型接收有 NPE 风险。

反例 : 比如显示成交总额涨跌情况，即正负 x%，x 为基本数据类型，调用的 RPC 服务，调用不成功时，返回的是默认值，页面显示为 0%，这是不合理的，应该显示成中划线。所以包装数据类型的 null 值，能够表示额外的信息，如:远程调用失败，异常退出。

## 注释

Java 中的注释有三种：

单行注释

多行注释

文档注释。

在我们编写代码的时候，如果代码量比较少，我们自己或者团队其他成员还可以很轻易地看懂代码，但是当项目结构一旦复杂起来，我们就需要用到注释了。注释并不会执行，是我们程序员写给自己看的，注释是你的代码说明书，能够帮助看代码的人快速地理清代码之间的逻辑关系。因此，在写程序的时候随手加上注释是一个非常好的习惯。

《Clean Code》这本书明确指出：

代码的注释不是越详细越好。实际上好的代码本身就是注释，我们要尽量规范和美化自己的代码来减少不必要的注释。

若编程语言足够有表达力，就不需要注释，尽量通过代码来阐述。

## 值传递

按值调用(call by value)表示方法接收的是调用者提供的值，

而按引用调用（call by reference)表示方法接收的是调用者提供的变量地址。一个方法可以修改传递引用所对应的变量值，而不能修改传递值调用所对应的变量值。 它用来描述各种程序设计语言（不只是 Java)中方法参数传递方式。

Java 程序设计语言总是采用按值调用。也就是说，方法得到的是所有参数值的一个拷贝，也就是说，方法不能修改传递给它的任何参数变量的内容。

一个方法不能修改一个基本数据类型的参数，而对象引用作为参数就不一样

一个方法可以改变一个对象参数的状态。

一个方法不能让对象参数引用一个新的对象。

## 重载和重写的区别

重载就是同样的一个方法能够根据输入数据的不同，做出不同的处理

重载就是同一个类中多个同名方法根据不同的传参来执行不同的逻辑处理。

重写就是当子类继承自父类的相同方法，输入数据一样，但要做出有别于父类的响应时，你就要覆盖父类方法

重写发生在运行期，是子类对父类的允许访问的方法的实现过程进行重新编写。

返回值类型、方法名、参数列表必须相同，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类。

如果父类方法访问修饰符为 private/final/static 则子类就不能重写该方法，但是被 static 修饰的方法能够被再次声明。

构造方法无法被重写

## 面向对象和面向过程的区别

面向过程 ：面向过程性能比面向对象高。 因为类调用时需要实例化，开销比较大，比较消耗资源，所以当性能是最重要的考量因素的时候，比如单片机、嵌入式开发、Linux/Unix 等一般采用面向过程开发。但是，面向过程没有面向对象易维护、易复用、易扩展。

面向对象 ：面向对象易维护、易复用、易扩展。 因为面向对象有封装、继承、多态性的特性，所以可以设计出低耦合的系统，使系统更加灵活、更加易于维护。但是，面向对象性能比面向过程低。

## 多态

多态，顾名思义，表示一个对象具有多种的状态。具体表现为父类的引用指向子类的实例。

多态的特点:

对象类型和引用类型之间具有继承（类）/实现（接口）的关系；

对象类型不可变，引用类型可变；

方法具有多态性，属性不具有多态性；

引用类型变量发出的方法调用的到底是哪个类中的方法，必须在程序运行期间才能确定；

多态不能调用“只在子类存在但在父类不存在”的方法；

如果子类重写了父类的方法，真正执行的是子类覆盖的方法，如果子类没有覆盖父类的方法，执行的是父类的方法。

## 成员变量与局部变量的区别

从语法形式上看:成员变量是属于类的，而局部变量是在方法中定义的变量或是方法的参数；成员变量可以被 public,private,static 等修饰符所修饰，而局部变量不能被访问控制修饰符及 static 所修饰；但是，成员变量和局部变量都能被 final 所修饰。

从变量在内存中的存储方式来看:如果成员变量是使用static修饰的，那么这个成员变量是属于类的，如果没有使用static修饰，这个成员变量是属于实例的。而对象存在于堆内存，局部变量则存在于栈内存。

从变量在内存中的生存时间上看:成员变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而自动消失。

成员变量如果没有被赋初值:则会自动以类型的默认值而赋值（一种情况例外:被 final 修饰的成员变量也必须显式地赋值），而局部变量则不会自动赋值。

## 接口和抽象类

### 区别

接口的方法默认是 public，所有方法在接口中不能有实现(Java 8 开始接口方法可以有默认实现），而抽象类可以有非抽象的方法。

接口中除了 static、final 变量，不能有其他变量，而抽象类中则不一定。

一个类可以实现多个接口，但只能实现一个抽象类。接口自己本身可以通过 extends 关键字扩展多个接口。

接口方法默认修饰符是 public，抽象方法可以有 public、protected 和 default 这些修饰符（抽象方法就是为了被重写所以不能使用 private 关键字修饰！）。

从设计层面来说，抽象是对类的抽象，是一种模板设计，而接口是对行为的抽象，是一种行为的规范。

备注：

在 JDK8 中，接口也可以定义静态方法，可以直接用接口名调用。实现类和实现是不可以调用的。如果同时实现两个接口，接口中定义了一样的默认方法，则必须重写，不然会报错。

jdk9 的接口被允许定义私有方法 。

总结一下 jdk7~jdk9 Java 中接口概念的变化：

在 jdk 7 或更早版本中，接口里面只能有常量变量和抽象方法。这些接口方法必须由选择实现接口的类实现。

jdk8 的时候接口可以有默认方法和静态方法功能。

Jdk 9 在接口中引入了私有方法和私有静态方法。

## **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)。)

String 类中使用 final 关键字修饰字符数组来保存字符串，private final char value[]，所以 String 对象是不可变的。

而 StringBuilder 与 StringBuffer 都继承自 AbstractStringBuilder 类，在 AbstractStringBuilder 中也是使用字符数组保存字符串char[]value 但是没有用 final 关键字修饰，所以这两种对象都是可变的。

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。

### 性能

每次对 String 类型进行改变的时候，都会生成一个新的 String 对象，然后将指针指向新的 String 对象。StringBuffer 每次都会对 StringBuffer 对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用 StringBuilder 相比使用 StringBuffer 仅能获得 10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

### 总结：

操作少量的数据: 适用 String

单线程操作字符串缓冲区下操作大量数据: 适用 StringBuilder

多线程操作字符串缓冲区下操作大量数据: 适用 StringBuffer

## Object 类的常见方法

public final native Class<?> getClass()//native方法，用于返回当前运行时对象的Class对象，使用了final关键字修饰，故不允许子类重写。

public native int hashCode() //native方法，用于返回对象的哈希码，主要使用在哈希表中，比如JDK中的HashMap。

public boolean equals(Object obj)//用于比较2个对象的内存地址是否相等，String类对该方法进行了重写用户比较字符串的值是否相等。

protected native Object clone() throws CloneNotSupportedException//naitive方法，用于创建并返回当前对象的一份拷贝。一般情况下，对于任何对象 x，表达式 x.clone() != x 为true，x.clone().getClass() == x.getClass() 为true。Object本身没有实现Cloneable接口，所以不重写clone方法并且进行调用的话会发生CloneNotSupportedException异常。

public String toString()//返回类的名字@实例的哈希码的16进制的字符串。建议Object所有的子类都重写这个方法。

public final native void notify()//native方法，并且不能重写。唤醒一个在此对象监视器上等待的线程(监视器相当于就是锁的概念)。如果有多个线程在等待只会任意唤醒一个。

public final native void notifyAll()//native方法，并且不能重写。跟notify一样，唯一的区别就是会唤醒在此对象监视器上等待的所有线程，而不是一个线程。

public final native void wait(long timeout) throws InterruptedException//native方法，并且不能重写。暂停线程的执行。注意：sleep方法没有释放锁，而wait方法释放了锁 。timeout是等待时间。

public final void wait(long timeout, int nanos) throws InterruptedException//多了nanos参数，这个参数表示额外时间（以毫微秒为单位，范围是 0-999999）。 所以超时的时间还需要加上nanos毫秒。

public final void wait() throws InterruptedException//跟之前的2个wait方法一样，只不过该方法一直等待，没有超时时间这个概念

protected void finalize() throws Throwable { }//实例被垃圾回收器回收的时候触发的操作

## 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；

## 文件与 I\O 流

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

### 为什么还要有字符流?

回答：字符流是由 Java 虚拟机将字节转换得到的，问题就出在这个过程还算是非常耗时，并且，如果我们不知道编码类型就很容易出现乱码问题。所以， I/O 流就干脆提供了一个直接操作字符的接口，方便我们平时对字符进行流操作。如果音频文件、图片等媒体文件用字节流比较好，如果涉及到字符的话使用字符流比较好。

### 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

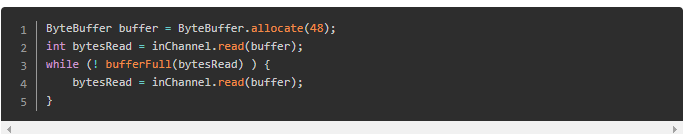
使用NIO的API调用时看起来与使用IO时有所不同，但这并不意外，因为并不是仅从一个InputStream逐字节读取，而是数据必须先读入缓冲区再处理。

（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

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

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。

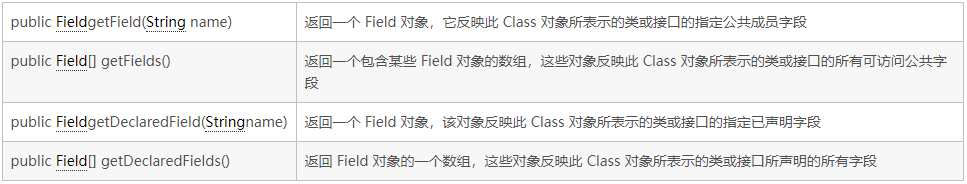
### 获取类的Class对象

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

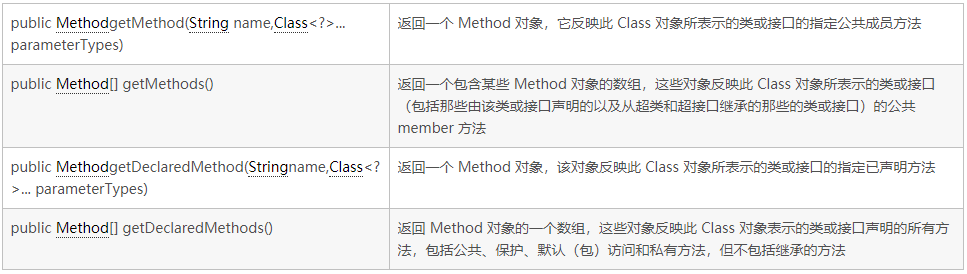
Class<T>类提供了几个方法获取类的方法。



### 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

### 静态编译和动态编译

静态编译： 在编译时确定类型，绑定对象

动态编译： 运行时确定类型，绑定对象

### 反射机制优缺点

优点： 运行期类型的判断，动态加载类，提高代码灵活度。

缺点： 1,性能瓶颈：反射相当于一系列解释操作，通知 JVM 要做的事情，性能比直接的 java 代码要慢很多。2,安全问题，让我们可以动态操作改变类的属性同时也增加了类的安全隐患。

### 反射的应用场景

反射是框架设计的灵魂。

在我们平时的项目开发过程中，基本上很少会直接使用到反射机制，但这不能说明反射机制没有用，实际上有很多设计、开发都与反射机制有关，例如模块化的开发，通过反射去调用对应的字节码；动态代理设计模式也采用了反射机制，还有我们日常使用的 Spring／Hibernate 等框架也大量使用到了反射机制。

举例：

我们在使用 JDBC 连接数据库时使用 Class.forName()通过反射加载数据库的驱动程序；

Spring 框架的 IOC（动态加载管理 Bean）创建对象以及 AOP（动态代理）功能都和反射有联系；

动态配置实例的属性；

## 常用关键字

### final关键字

意思是最终的、不可修改的，最见不得变化 ，用来修饰类、方法和变量，具有以下特点：

final修饰的类不能被继承，final类中的所有成员方法都会被隐式的指定为final方法；

final修饰的方法不能被重写；

final修饰的变量是常量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能让其指向另一个对象。

说明：使用final方法的原因有两个。第一个原因是把方法锁定，以防任何继承类修改它的含义；第二个原因是效率。在早期的Java实现版本中，会将final方法转为内嵌调用。但是如果方法过于庞大，可能看不到内嵌调用带来的任何性能提升（现在的Java版本已经不需要使用final方法进行这些优化了）。类中所有的private方法都隐式地指定为final。

### static 关键字

static 关键字主要有以下四种使用场景：

修饰成员变量和成员方法: 被 static 修饰的成员属于类，不属于单个这个类的某个对象，被类中所有对象共享，可以并且建议通过类名调用。被static 声明的成员变量属于静态成员变量，静态变量 存放在 Java 内存区域的方法区。调用格式：类名.静态变量名 类名.静态方法名()

静态代码块: 静态代码块定义在类中方法外, 静态代码块在非静态代码块之前执行(静态代码块—>非静态代码块—>构造方法)。 该类不管创建多少对象，静态代码块只执行一次.

静态内部类（static修饰类的话只能修饰内部类）：

静态内部类与非静态内部类之间存在一个最大的区别: 非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围类，但是静态内部类却没有。没有这个引用就意味着：1. 它的创建是不需要依赖外围类的创建。2. 它不能使用任何外围类的非static成员变量和方法。

静态导包(用来导入类中的静态资源，1.5之后的新特性): 格式为：import static 这两个关键字连用可以指定导入某个类中的指定静态资源，并且不需要使用类名调用类中静态成员，可以直接使用类中静态成员变量和成员方法。

方法区与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 Non-Heap（非堆），目的应该是与 Java 堆区分开来。

HotSpot 虚拟机中方法区也常被称为 “永久代”，本质上两者并不等价。仅仅是因为 HotSpot 虚拟机设计团队用永久代来实现方法区而已，这样 HotSpot 虚拟机的垃圾收集器就可以像管理 Java 堆一样管理这部分内存了。但是这并不是一个好主意，因为这样更容易遇到内存溢出问题。

### this 关键字

this关键字用于引用类的当前实例。

关键字是可选的，这意味着如果上面的示例在不使用此关键字的情况下表现相同。 但是，使用此关键字可能会使代码更易读或易懂。

### super 关键字

super关键字用于从子类访问父类的变量和方法

使用 this 和 super 要注意的问题：

在构造器中使用 super（） 调用父类中的其他构造方法时，该语句必须处于构造器的首行，否则编译器会报错。另外，this 调用本类中的其他构造方法时，也要放在首行。

this、super不能用在static方法中。

## 序列化

对于不想进行序列化的变量，使用 transient 关键字修饰。

transient 关键字的作用是：阻止实例中那些用此关键字修饰的的变量序列化；当对象被反序列化时，被 transient 修饰的变量值不会被持久化和恢复。transient 只能修饰变量，不能修饰类和方法。

## == 和hashcode 和 equals

### ==和equals的区别

== : 它的作用是判断两个对象的地址是不是相等。即判断两个对象是不是同一个对象。(基本数据类型==比较的是值，引用数据类型==比较的是内存地址)

因为 Java 只有值传递，所以，对于 == 来说，不管是比较基本数据类型，还是引用数据类型的变量，其本质比较的都是值，只是引用类型变量存的值是对象的地址。

equals() : 它的作用也是判断两个对象是否相等，它不能用于比较基本数据类型的变量。equals()方法存在于Object类中，而Object类是所有类的直接或间接父类。

equals() 方法存在两种使用情况：

情况 1：类没有覆盖 equals()方法。则通过 equals()比较该类的两个对象时，等价于通过“==”比较这两个对象。使用的默认是 Object类equals()方法。

情况 2：类覆盖了 equals()方法。一般，我们都覆盖 equals()方法来两个对象的内容相等；若它们的内容相等，则返回 true(即，认为这两个对象相等)。

### hashcode

hashCode()在散列表中才有用，在其它情况下没用

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。

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个 int 整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。hashCode() 定义在 JDK 的 Object 类中，这就意味着 Java 中的任何类都包含有 hashCode() 函数。另外需要注意的是： Object 的 hashcode 方法是本地方法，也就是用 c 语言或 c++ 实现的，该方法通常用来将对象的 内存地址 转换为整数之后返回。

public native int hashCode();

散列表存储的是键值对(key-value)，它的特点是：能根据“键”快速的检索出对应的“值”。这其中就利用到了散列码！（可以快速找到所需要的对象）

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。

### 为什么要有 hashCode？

我们以“HashSet 如何检查重复”为例子来说明为什么要有 hashCode？

当你把对象加入 HashSet 时，HashSet 会先计算对象的 hashcode 值来判断对象加入的位置，同时也会与其他已经加入的对象的 hashcode 值作比较，如果没有相符的 hashcode，HashSet 会假设对象没有重复出现。但是如果发现有相同 hashcode 值的对象，这时会调用 equals（）方法来检查 hashcode 相等的对象是否真的相同。如果两者相同，HashSet 就不会让其加入操作成功。如果不同的话，就会重新散列到其他位置。（摘自我的 Java 启蒙书《Head fist java》第二版）。这样我们就大大减少了 equals 的次数，相应就大大提高了执行速度。

### 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。hashCode() 所使用的哈希算法也许刚好会让多个对象传回相同的杂凑值。越糟糕的杂凑算法越容易碰撞，但这也与数据值域分布的特性有关（所谓碰撞也就是指的是不同的对象得到相同的 hashCode。

我们刚刚也提到了 HashSet,如果 HashSet 在对比的时候，同样的 hashcode 有多个对象，它会使用 equals() 来判断是否真的相同。也就是说 hashcode 只是用来缩小查找成本

### 覆盖equals时总要覆盖hashCode

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。

如果两个对象相等，则 hashcode 一定也是相同的。两个对象相等,对两个对象分别调用 equals 方法都返回 true。但是，两个对象有相同的 hashcode 值，它们也不一定是相等的 。因此，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖。

hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写 hashCode()，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）

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。

### 正确使用 equals 方法

Object的equals方法容易抛空指针异常，应使用常量或确定有值的对象来调用 equals。

不过更推荐使用 java.util.Objects#equals

## 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

在以下 4 种特殊情况下，finally 块不会被执行：

在 finally 语句块第一行发生了异常。 因为在其他行，finally 块还是会得到执行

在前面的代码中用了 System.exit(int)已退出程序。 exit 是带参函数 ；若该语句在异常语句之后，finally 会执行

程序所在的线程死亡。

关闭 CPU。

## 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（generics）是 JDK 5 中引入的一个新特性, 泛型提供了编译时类型安全检测机制，该机制允许程序员在编译时检测到非法的类型。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。

Java的泛型是伪泛型，这是因为Java在编译期间，所有的泛型信息都会被擦掉，这也就是通常所说类型擦除 。

泛型一般有三种使用方式: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

### 常用的通配符为： T，E，K，V，？

？ 表示不确定的 java 类型

T (type) 表示具体的一个java类型

K V (key value) 分别代表java键值中的Key Value

E (element) 代表Element

## 枚举

enum关键字在 java5 中引入，表示一种特殊类型的类，其总是继承java.lang.Enum类，

枚举在很多时候会和常量拿来对比，我们大量实际使用枚举的地方就是为了替代常量。

以这种方式定义的常量使代码更具可读性，允许进行编译时检查，预先记录可接受值的列表，并避免由于传入无效值而引起的意外行为。

### Singleton 模式

通常，使用类实现 Singleton 模式并非易事，枚举提供了一种实现单例的简便方法。

《Effective Java 》和《Java与模式》都非常推荐这种方式，使用这种方式方式实现枚举可以有什么好处呢？

《Effective Java》

这种方法在功能上与公有域方法相近，但是它更加简洁，无偿提供了序列化机制，绝对防止多次实例化，即使是在面对复杂序列化或者反射攻击的时候。虽然这种方法还没有广泛采用，但是单元素的枚举类型已经成为实现 Singleton的最佳方法。 —-《Effective Java 中文版 第二版》

《Java与模式》

《Java与模式》中，作者这样写道，使用枚举来实现单实例控制会更加简洁，而且无偿地提供了序列化机制，并由JVM从根本上提供保障，绝对防止多次实例化，是更简洁、高效、安全的实现单例的方式。

## 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

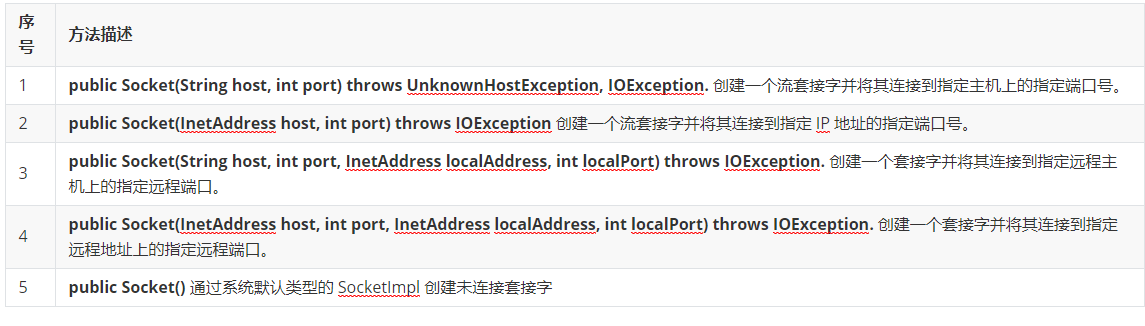
4xx: 请求出错 404 NotFound，403 Forbidden

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：



## 键盘输入常用的两种方法

方法 1：通过 Scanner

Scanner input = new Scanner(System.in);

String s = input.nextLine();

input.close();

方法 2：通过 BufferedReader

BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

String s = input.readLine();

# Java多线程

## 基础

### 并发和并行

并行是多个cpu同时执行多个任务。

单位时间内，多个任务同时执行。

并发是一个cpu采用时间片同时执行多个任务。

同一时间段，多个任务都在执行 (单位时间内不一定同时执行)；

### 进程和线程

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

线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享进程的堆和方法区资源，但每个线程有自己的程序计数器、虚拟机栈和本地方法栈，，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

线程 是 进程 划分成的更小的运行单位。

线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。

线程执行开销小，但不利于资源的管理和保护；而进程正相反

程序是含有指令和数据的文件，被存储在磁盘或其他的数据存储设备中，也就是说程序是静态的代码。

程序计数器为什么是私有的?

程序计数器主要有下面两个作用：

1. 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。

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

需要注意的是，如果执行的是 native 方法，那么程序计数器记录的是 undefined 地址，只有执行的是 Java 代码时程序计数器记录的才是下一条指令的地址。

所以，程序计数器私有主要是为了\*\*线程切换后能恢复到正确的执行位置\*\*。

虚拟机栈和本地方法栈为什么是私有的?

- \*\*虚拟机栈：\*\* 每个 Java 方法在执行的同时会创建一个栈帧用于存储局部变量表、操作数栈、常量池引用等信息。从方法调用直至执行完成的过程，就对应着一个栈帧在 Java 虚拟机栈中入栈和出栈的过程。

- \*\*本地方法栈：\*\* 和虚拟机栈所发挥的作用非常相似，区别是： \*\*虚拟机栈为虚拟机执行 Java 方法 （也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。\*\* 在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。

所以，为了\*\*保证线程中的局部变量不被别的线程访问到\*\*，虚拟机栈和本地方法栈是线程私有的。

一句话简单了解堆和方法区

堆和方法区是所有线程共享的资源，其中堆是进程中最大的一块内存，主要用于存放新创建的对象 (几乎所有对象都在这里分配内存)，方法区主要用于存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

### 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 。

### 并发编程可能会遇到很多问题

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

避免线程死锁

1. \*\*破坏互斥条件\*\* ：这个条件我们没有办法破坏，因为我们用锁本来就是想让他们互斥的（临界资源需要互斥访问）。

2. \*\*破坏请求与保持条件\*\* ：一次性申请所有的资源。

3. \*\*破坏不剥夺条件\*\* ：占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。

4. \*\*破坏循环等待条件\*\* ：靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件。

### 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

Runnable接口和Callable接口的区别

`Runnable`自Java 1.0以来一直存在，但`Callable`仅在Java 1.5中引入,目的就是为了来处理`Runnable`不支持的用例。\*\*`Runnable` 接口\*\*不会返回结果或抛出检查异常，但是\*\*`Callable` 接口\*\*可以。所以，如果任务不需要返回结果或抛出异常推荐使用 \*\*`Runnable` 接口\*\*，这样代码看起来会更加简洁。

工具类 `Executors` 可以实现 `Runnable` 对象和 `Callable` 对象之间的相互转换。（`Executors.callable（Runnable task`）或 `Executors.callable（Runnable task，Object resule）`）。

### 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。`synchronized ` 可以保证代码片段的原子性。

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）`volatile` 关键字可以保证共享变量的可见性。

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，`volatile` 关键字可以禁止指令进行重排序优化。

### 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

### start() 方法 run() 方法

这是另一个非常经典的 java 多线程面试问题，而且在面试中会经常被问到。很简单，但是很多人都会答不上来！

new 一个 Thread，线程进入了新建状态;调用 start() 方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了。 start() 会执行线程的相应准备工作，然后自动执行 run() 方法的内容，这是真正的多线程工作。 而直接执行 run() 方法，会把 run 方法当成一个 main 线程下的普通方法去执行，并不会在某个线程中执行它，所以这并不是多线程工作。

\*\*总结： 调用 start 方法方可启动线程并使线程进入就绪状态，而 run 方法只是 thread 的一个普通方法调用，还是在主线程里执行。

### sleep() 方法和 wait() 方法区别和共同点?

- 两者最主要的区别在于：sleep 方法没有释放锁，而 wait 方法释放了锁 。

- 两者都可以暂停线程的执行。

- Wait 通常被用于线程间交互/通信，sleep 通常被用于暂停执行。

- 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。

### happens-before的定义

happens-before定义了八条规则，这八条规则都是用来保证如果A happens-before B，那么A的执行结果对B可见且A的执行顺序排在B之前。

程序次序规则：在一个单独的线程中，按照程序代码的执行流顺序，（时间上）先执行的操作happen—before（时间上）后执行的操作。

管理锁定规则：一个unlock操作happen—before后面（时间上的先后顺序，下同）对同一个锁的lock操作。

volatile变量规则：对一个volatile变量的写操作happen—before后面对该变量的读操作。

线程启动规则：Thread对象的start（）方法happen—before此线程的每一个动作。

线程终止规则：线程的所有操作都happen—before对此线程的终止检测，可以通过Thread.join（）方法结束、Thread.isAlive（）的返回值等手段检测到线程已经终止执行。

线程中断规则：对线程interrupt（）方法的调用happen—before发生于被中断线程的代码检测到中断时事件的发生。

对象终结规则：一个对象的初始化完成（构造函数执行结束）happen—before它的finalize（）方法的开始。

传递性：如果操作A happen—before操作B，操作B happen—before操作C，那么可以得出A happen—before操作C。

happens-before定义了这么多规则，其实总结起来可以归纳为一句话：happens-before规则保证了单线程和正确同步的多线程的执行结果不会被改变。

## 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 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关键字解决的是多个线程之间访问资源的同步性，synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

另外，在 Java 早期版本中，synchronized属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的 synchronized 效率低的原因。庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

### synchronized 关键字的底层原理

synchronized 关键字底层原理属于 JVM 层面。

线程的生命周期存在5个状态，start、running、waiting、blocking和dead

对于一个synchronized修饰的方法(代码块)来说：

当多个线程同时访问该方法，那么这些线程会先被放进\_EntryList队列，此时线程处于blocking状态

当一个线程获取到了实例对象的监视器（monitor）锁，那么就可以进入running状态，执行方法，此时，ObjectMonitor对象的\_owner指向当前线程，\_count加1表示当前对象锁被一个线程获取

当running状态的线程调用wait()方法，那么当前线程释放monitor对象，进入waiting状态，ObjectMonitor对象的\_owner变为null，\_count减1，同时线程进入\_WaitSet队列，直到有线程调用notify()方法唤醒该线程，则该线程重新获取monitor对象进入\_Owner区

如果当前线程执行完毕，那么也释放monitor对象，进入waiting状态，ObjectMonitor对象的\_owner变为null，\_count减1

synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置。 当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor(monitor对象存在于每个Java对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因) 的持有权。当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行 monitorexit 指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的确实是 ACC\_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法，JVM 通过该 ACC\_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

JDK1.6 对锁的实现引入了大量的优化，如偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化等技术来减少锁操作的开销。

锁主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

### Java对象在内存中的布局

首先，我们要知道对象在内存中的布局：

已知对象是存放在堆内存中的，对象大致可以分为三个部分，分别是对象头、实例变量和填充字节。

对象头的主要是由MarkWord和Klass Point(类型指针)组成，其中Klass Point是是对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例，Mark Word用于存储对象自身的运行时数据。如果对象是数组对象，那么对象头占用3个字宽（Word），如果对象是非数组对象，那么对象头占用2个字宽。（1word = 2 Byte = 16 bit）

实例变量存储的是对象的属性信息，包括父类的属性信息，按照4字节对齐

填充字符，因为虚拟机要求对象字节必须是8字节的整数倍，填充字符就是用于凑齐这个整数倍的



### synchronized关键字最主要的三种使用方式：

1、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

2、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。

2、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！

### JDK1.6 之后的底层优化

JDK1.6 对锁的实现引入了大量的优化，如偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化等技术来减少锁操作的开销。

锁主要存在四中状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

①偏向锁

偏向锁的它的意思是会偏向于第一个获得它的线程，如果在接下来的执行中，该锁没有被其他线程获取，那么持有偏向锁的线程就不需要进行同步

② 轻量级锁

轻量级锁考虑的是竞争锁对象的线程不多，而且线程持有锁的时间也不长的情景。因为阻塞线程需要CPU从用户态转到内核态，代价较大，如果刚刚阻塞不久这个锁就被释放了，那这个代价就有点得不偿失了，因此这个时候就干脆不阻塞这个线程，让它自旋这等待锁释放。

轻量级锁能够提升程序同步性能的依据是“对于绝大部分锁，在整个同步周期内都是不存在竞争的”，这是一个经验数据。如果没有竞争，轻量级锁使用 CAS 操作避免了使用互斥操作的开销。但如果存在锁竞争，除了互斥量开销外，还会额外发生CAS操作，因此在有锁竞争的情况下，轻量级锁比传统的重量级锁更慢！如果锁竞争激烈，那么轻量级将很快膨胀为重量级锁！

重量级锁

如果自旋次数到了线程1还没有释放锁，或者线程1还在执行，线程2还在自旋等待，这时又有一个线程3过来竞争这个锁对象，那么这个时候轻量级锁就会膨胀为重量级锁。重量级锁把除了拥有锁的线程都阻塞，防止CPU空转。

③ 自旋锁和自适应自旋

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。

互斥同步对性能最大的影响就是阻塞的实现，因为挂起线程/恢复线程的操作都需要转入内核态中完成（用户态转换到内核态会耗费时间）。

一般线程持有锁的时间都不是太长，所以仅仅为了这一点时间去挂起线程/恢复线程是得不偿失的。所以，虚拟机的开发团队就这样去考虑：“我们能不能让后面来的请求获取锁的线程等待一会而不被挂起呢？看看持有锁的线程是否很快就会释放锁”。为了让一个线程等待，我们只需要让线程执行一个忙循环（自旋），这项技术就叫做自旋。

自旋锁在 JDK1.6 之前其实就已经引入了，不过是默认关闭的，需要通过`--XX:+UseSpinning`参数来开启。JDK1.6及1.6之后，就改为默认开启的了。需要注意的是：自旋等待不能完全替代阻塞，因为它还是要占用处理器时间。如果锁被占用的时间短，那么效果当然就很好了！反之，相反！自旋等待的时间必须要有限度。如果自旋超过了限定次数任然没有获得锁，就应该挂起线程。自旋次数的默认值是10次，用户可以修改`--XX:PreBlockSpin`来更改。

另外,在 JDK1.6 中引入了自适应的自旋锁。自适应的自旋锁带来的改进就是：自旋的时间不在固定了，而是和前一次同一个锁上的自旋时间以及锁的拥有者的状态来决定，虚拟机变得越来越“聪明”了。

\*\*④ 锁消除\*\*

锁消除理解起来很简单，它指的就是虚拟机即使编译器在运行时，如果检测到那些共享数据不可能存在竞争，那么就执行锁消除。锁消除可以节省毫无意义的请求锁的时间。

\*\*⑤ 锁粗化\*\*

原则上，我们在编写代码的时候，总是推荐将同步块的作用范围限制得尽量小，——直在共享数据的实际作用域才进行同步，这样是为了使得需要同步的操作数量尽可能变小，如果存在锁竞争，那等待线程也能尽快拿到锁。

大部分情况下，上面的原则都是没有问题的，但是如果一系列的连续操作都对同一个对象反复加锁和解锁，那么会带来很多不必要的性能消耗。

## 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。

### 原理

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

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

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

　　2）它会强制将对缓存的修改操作立即写入主存；

　　3）如果是写操作，它会导致其他CPU中对应的缓存行无效。

在Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

　　在Java里面，可以通过volatile关键字来保证一定的“有序性”（具体原理在下一节讲述）。另外可以通过synchronized和Lock来保证有序性，很显然，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。

　　另外，Java内存模型具备一些先天的“有序性”，即不需要通过任何手段就能够得到保证的有序性，这个通常也称为 happens-before 原则。如果两个操作的执行次序无法从happens-before原则推导出来，那么它们就不能保证它们的有序性，虚拟机可以随意地对它们进行重排序。

　　下面就来具体介绍下happens-before原则（先行发生原则）：

程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作

锁定规则：一个unLock操作先行发生于后面对同一个锁额lock操作

volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作

传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C

线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作

线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生

线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行

对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始

　　这8条原则摘自《深入理解Java虚拟机》。

　　这8条规则中，前4条规则是比较重要的，后4条规则都是显而易见的。

　　下面我们来解释一下前4条规则：

　　对于程序次序规则来说，我的理解就是一段程序代码的执行在单个线程中看起来是有序的。注意，虽然这条规则中提到“书写在前面的操作先行发生于书写在后面的操作”，这个应该是程序看起来执行的顺序是按照代码顺序执行的，因为虚拟机可能会对程序代码进行指令重排序。虽然进行重排序，但是最终执行的结果是与程序顺序执行的结果一致的，它只会对不存在数据依赖性的指令进行重排序。因此，在单个线程中，程序执行看起来是有序执行的，这一点要注意理解。事实上，这个规则是用来保证程序在单线程中执行结果的正确性，但无法保证程序在多线程中执行的正确性。

　　第二条规则也比较容易理解，也就是说无论在单线程中还是多线程中，同一个锁如果出于被锁定的状态，那么必须先对锁进行了释放操作，后面才能继续进行lock操作。

　　第三条规则是一条比较重要的规则，也是后文将要重点讲述的内容。直观地解释就是，如果一个线程先去写一个变量，然后一个线程去进行读取，那么写入操作肯定会先行发生于读操作。

　　第四条规则实际上就是体现happens-before原则具备传递性。

### Java内存模型

在 JDK1.2 之前，Java的内存模型实现总是从\*\*主存\*\*（即共享内存）读取变量，是不需要进行特别的注意的。而在当前的 Java 内存模型下，线程可以把变量保存\*\*本地内存\*\*（比如机器的寄存器）中，而不是直接在主存中进行读写。这就可能造成一个线程在主存中修改了一个变量的值，而另外一个线程还继续使用它在寄存器中的变量值的拷贝，造成\*\*数据的不一致\*\*。

### 实践

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)。

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

### synchronized、volatile、CAS 比较

`synchronized` 关键字和 `volatile` 关键字是两个互补的存在，而不是对立的存在：

- \*\*volatile关键字\*\*是线程同步的\*\*轻量级实现\*\*，所以\*\*volatile性能肯定比synchronized关键字要好\*\*。但是\*\*volatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块\*\*。synchronized关键字在JavaSE1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，\*\*实际开发中使用 synchronized 关键字的场景还是更多一些\*\*。

- \*\*多线程访问volatile关键字不会发生阻塞，而synchronized关键字可能会发生阻塞\*\*

- \*\*volatile关键字能保证数据的可见性，但不能保证数据的原子性。synchronized关键字两者都能保证。\*\*

- \*\*volatile关键字主要用于解决变量在多个线程之间的可见性，而 synchronized关键字解决的是多个线程之间访问资源的同步性。\*\*

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。

（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

### 谈谈 synchronized和ReentrantLock 的区别

① 两者都是可重入锁

两者都是可重入锁。“可重入锁”概念是：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

② synchronized 依赖于 JVM 而 ReentrantLock 依赖于 API

synchronized 是依赖于 JVM 实现的，前面我们也讲到了 虚拟机团队在 JDK1.6 为 synchronized 关键字进行了很多优化，但是这些优化都是在虚拟机层面实现的，并没有直接暴露给我们。ReentrantLock 是 JDK 层面实现的（也就是 API 层面，需要 lock() 和 unlock() 方法配合 try/finally 语句块来完成），所以我们可以通过查看它的源代码，来看它是如何实现的。

③ ReentrantLock 比 synchronized 增加了一些高级功能

相比synchronized，ReentrantLock增加了一些高级功能。主要来说主要有三点：①等待可中断；②可实现公平锁；③可实现选择性通知（锁可以绑定多个条件）

- ReentrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()来实现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。

- ReentrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。\*\* ReentrantLock默认情况是非公平的，可以通过 ReentrantLock类的`ReentrantLock(boolean fair)`构造方法来制定是否是公平的。

- synchronized关键字与wait()和notify()/notifyAll()方法相结合可以实现等待/通知机制，ReentrantLock类当然也可以实现，但是需要借助于Condition接口与newCondition() 方法。Condition是JDK1.5之后才有的，它具有很好的灵活性，比如可以实现多路通知功能也就是在一个Lock对象中可以创建多个Condition实例（即对象监视器），\*\*线程对象可以注册在指定的Condition中，从而可以有选择性的进行线程通知，在调度线程上更加灵活。 在使用notify()/notifyAll()方法进行通知时，被通知的线程是由 JVM 选择的，用ReentrantLock类结合Condition实例可以实现“选择性通知”\*\* ，这个功能非常重要，而且是Condition接口默认提供的。而synchronized关键字就相当于整个Lock对象中只有一个Condition实例，所有的线程都注册在它一个身上。如果执行notifyAll()方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题，而Condition实例的signalAll()方法 只会唤醒注册在该Condition实例中的所有等待线程。

如果你想使用上述功能，那么选择ReentrantLock是一个不错的选择。

④ 性能已不是选择标准

在JDK1.6之前，synchronized 的性能是比 ReenTrantLock 差很多。具体表示为：synchronized 关键字吞吐量随线程数的增加，下降得非常严重。而ReenTrantLock 基本保持一个比较稳定的水平。我觉得这也侧面反映了， synchronized 关键字还有非常大的优化余地。后续的技术发展也证明了这一点，我们上面也讲了在 JDK1.6 之后 JVM 团队对 synchronized 关键字做了很多优化。JDK1.6 之后，synchronized 和 ReenTrantLock 的性能基本是持平了。所以网上那些说因为性能才选择 ReenTrantLock 的文章都是错的！JDK1.6之后，性能已经不是选择synchronized和ReenTrantLock的影响因素了！而且虚拟机在未来的性能改进中会更偏向于原生的synchronized，所以还是提倡在synchronized能满足你的需求的情况下，优先考虑使用synchronized关键字来进行同步！优化后的synchronized和ReenTrantLock一样，在很多地方都是用到了CAS操作\*\*。

## ThreadLocal

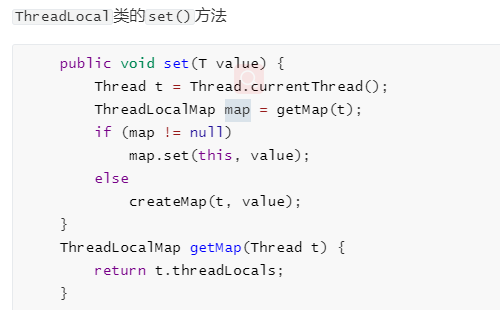
通常情况下，我们创建的变量是可以被任何一个线程访问并修改的。\*\*如果想实现每一个线程都有自己的专属本地变量该如何解决呢？\*\* JDK中提供的`ThreadLocal`类正是为了解决这样的问题。 \*\*`ThreadLocal`类主要解决的就是让每个线程绑定自己的值，可以将`ThreadLocal`类形象的比喻成存放数据的盒子，盒子中可以存储每个线程的私有数据。\*\*

\*\*如果你创建了一个`ThreadLocal`变量，那么访问这个变量的每个线程都会有这个变量的本地副本，这也是`ThreadLocal`变量名的由来。他们可以使用 `get（）` 和 `set（）` 方法来获取默认值或将其值更改为当前线程所存的副本的值，从而避免了线程安全问题。

### ThreadLocal原理



从上面`Thread`类 源代码可以看出`Thread` 类中有一个 `threadLocals` 和 一个 `inheritableThreadLocals` 变量，它们都是 `ThreadLocalMap` 类型的变量,我们可以把 `ThreadLocalMap` 理解为`ThreadLocal` 类实现的定制化的 `HashMap`。默认情况下这两个变量都是null，只有当前线程调用 `ThreadLocal` 类的 `set`或`get`方法时才创建它们，实际上调用这两个方法的时候，我们调用的是`ThreadLocalMap`类对应的 `get()`、`set() `方法。



通过上面这些内容，我们足以通过猜测得出结论：\*\*最终的变量是放在了当前线程的 `ThreadLocalMap` 中，并不是存在 `ThreadLocal` 上，`ThreadLocal` 可以理解为只是`ThreadLocalMap`的封装，传递了变量值。\*\* `ThrealLocal` 类中可以通过`Thread.currentThread()`获取到当前线程对象后，直接通过`getMap(Thread t)`可以访问到该线程的`ThreadLocalMap`对象。

每个`Thread`中都具备一个`ThreadLocalMap`，而`ThreadLocalMap`可以存储以`ThreadLocal`为key ，Object 对象为 value的键值对。

ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {

......

}

比如我们在同一个线程中声明了两个 `ThreadLocal` 对象的话，会使用 `Thread`内部都是使用仅有那个`ThreadLocalMap` 存放数据的，`ThreadLocalMap`的 key 就是 `ThreadLocal`对象，value 就是 `ThreadLocal` 对象调用`set`方法设置的值

## 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。

CLH(Craig,Landin,and Hagersten)队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）。AQS是将每条请求共享资源的线程封装成一个CLH锁队列的一个结点（Node）来实现锁的分配。

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

1、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

公平锁和非公平锁只有两处不同：

1. 非公平锁在调用 lock 后，首先就会调用 CAS 进行一次抢锁，如果这个时候恰巧锁没有被占用，那么直接就获取到锁返回了。

2. 非公平锁在 CAS 失败后，和公平锁一样都会进入到 tryAcquire 方法，在 tryAcquire 方法中，如果发现锁这个时候被释放了（state == 0），非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁，乖乖排到后面。

公平锁和非公平锁就这两点区别，如果这两次 CAS 都不成功，那么后面非公平锁和公平锁是一样的，都要进入到阻塞队列等待唤醒。

相对来说，非公平锁会有更好的性能，因为它的吞吐量比较大。当然，非公平锁让获取锁的时间变得更加不确定，可能会导致在阻塞队列中的线程长期处于饥饿状态。

2、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。

ReentrantReadWriteLock 可以看成是组合式，因为ReentrantReadWriteLock也就是读写锁允许多个线程同时对某一资源进行读。

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。

### AQS底层使用了模板方法模式

同步器的设计是基于模板方法模式的，如果需要自定义同步器一般的方式是这样（模板方法模式很经典的一个应用）：

1. 使用者继承AbstractQueuedSynchronizer并重写指定的方法。（这些重写方法很简单，无非是对于共享资源state的获取和释放）

2. 将AQS组合在自定义同步组件的实现中，并调用其模板方法，而这些模板方法会调用使用者重写的方法。

这和我们以往通过实现接口的方式有很大区别，这是模板方法模式很经典的一个运用。

\*\*AQS使用了模板方法模式，自定义同步器时需要重写下面几个AQS提供的模板方法：\*\*

isHeldExclusively()//该线程是否正在独占资源。只有用到condition才需要去实现它。

tryAcquire(int)//独占方式。尝试获取资源，成功则返回true，失败则返回false。

tryRelease(int)//独占方式。尝试释放资源，成功则返回true，失败则返回false。

tryAcquireShared(int)//共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。

tryReleaseShared(int)//共享方式。尝试释放资源，成功则返回true，失败则返回false。

默认情况下，每个方法都抛出 `UnsupportedOperationException`。 这些方法的实现必须是内部线程安全的，并且通常应该简短而不是阻塞。AQS类中的其他方法都是final ，所以无法被其他类使用，只有这几个方法可以被其他类使用。

以ReentrantLock为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A线程自己是可以重复获取此锁的（state会累加），这就是可重入的概念。但要注意，获取多少次就要释放多么次，这样才能保证state是能回到零态的。

再以CountDownLatch以例，任务分为N个子线程去执行，state也初始化为N（注意N要与线程个数一致）。这N个子线程是并行执行的，每个子线程执行完后countDown()一次，state会CAS(Compare and Swap)减1。等到所有子线程都执行完后(即state=0)，会unpark()主调用线程，然后主调用线程就会从await()函数返回，继续后余动作。

一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现`tryAcquire-tryRelease`、`tryAcquireShared-tryReleaseShared`中的一种即可。但AQS也支持自定义同步器同时实现独占和共享两种方式，如`ReentrantReadWriteLock`。

### 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。

CountDownLatch是共享锁的一种实现,它默认构造 AQS 的 state 值为 count。当线程使用countDown方法时,其实使用了`tryReleaseShared`方法以CAS的操作来减少state,直至state为0就代表所有的线程都调用了countDown方法。当调用await方法的时候，如果state不为0，就代表仍然有线程没有调用countDown方法，那么就把已经调用过countDown的线程都放入阻塞队列Park,并自旋CAS判断state == 0，直至最后一个线程调用了countDown，使得state == 0，于是阻塞的线程便判断成功，全部往下执行。

两个方法：

1. 计数方法：countDownLath.countDown();

2. 等待方法：await（）；

CountDownLatch 的两种典型用法

1. 某一线程在开始运行前等待 n 个线程执行完毕。将 CountDownLatch 的计数器初始化为 n ：`new CountDownLatch(n)`，每当一个任务线程执行完毕，就将计数器减 1 `countdownlatch.countDown()`，当计数器的值变为 0 时，在`CountDownLatch上 await()` 的线程就会被唤醒。一个典型应用场景就是启动一个服务时，主线程需要等待多个组件加载完毕，之后再继续执行。

2. 实现多个线程开始执行任务的最大并行性。注意是并行性，不是并发，强调的是多个线程在某一时刻同时开始执行。类似于赛跑，将多个线程放到起点，等待发令枪响，然后同时开跑。做法是初始化一个共享的 `CountDownLatch` 对象，将其计数器初始化为 1 ：`new CountDownLatch(1)`，多个线程在开始执行任务前首先 `coundownlatch.await()`，当主线程调用 countDown() 时，计数器变为 0，多个线程同时被唤醒。

CountDownLatch 的不足

CountDownLatch 是一次性的，计数器的值只能在构造方法中初始化一次，之后没有任何机制再次对其设置值，当 CountDownLatch 使用完毕后，它不能再次被使用。

### 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 的应用场景

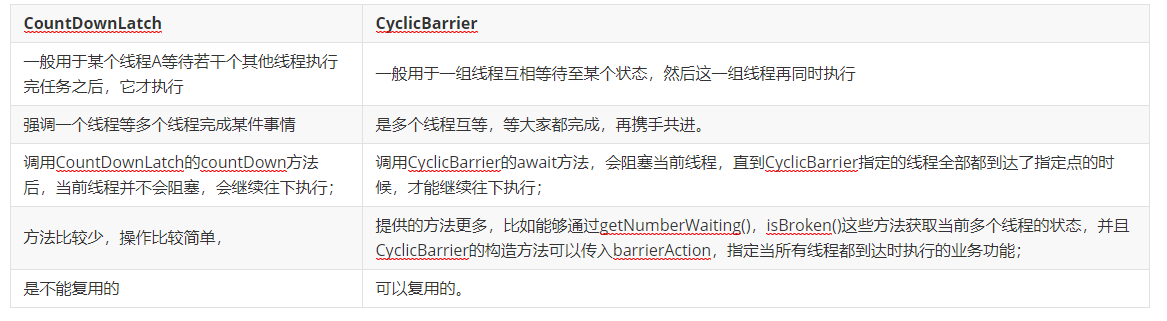
CyclicBarrier 可以用于多线程计算数据，最后合并计算结果的应用场景。比如我们用一个 Excel 保存了用户所有银行流水，每个 Sheet 保存一个帐户近一年的每笔银行流水，现在需要统计用户的日均银行流水，先用多线程处理每个 sheet 里的银行流水，都执行完之后，得到每个 sheet 的日均银行流水，最后，再用 barrierAction 用这些线程的计算结果，计算出整个 Excel 的日均银行流水。

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。

当调用 `CyclicBarrier` 对象调用 `await()` 方法时，实际上调用的是`dowait(false, 0L)`方法。 `await()` 方法就像树立起一个栅栏的行为一样，将线程挡住了，当拦住的线程数量达到 parties 的值时，栅栏才会打开，线程才得以通过执行。

`CyclicBarrier` 内部通过一个 count 变量作为计数器，cout 的初始值为 parties 属性的初始化值，每当一个线程到了栅栏这里了，那么就将计数器减一。如果 count 值为 0 了，表示这是这一代最后一个线程到达栅栏，就尝试执行我们构造方法中输入的任务。

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

Executor 框架是 Java5 之后引进的，在 Java 5 之后，通过 Executor 来启动线程比使用 Thread 的 start 方法更好，除了更易管理，效率更好（用线程池实现，节约开销）外，还有关键的一点：有助于避免 this 逃逸问题。

> 补充：this 逃逸是指在构造函数返回之前其他线程就持有该对象的引用. 调用尚未构造完全的对象的方法可能引发令人疑惑的错误。

Executor 框架不仅包括了线程池的管理，还提供了线程工厂、队列以及拒绝策略等，Executor 框架让并发编程变得更加简单。

### 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。

### Executors 返回线程池对象的弊端

FixedThreadPool 和 SingleThreadExecutor ： 允许请求的队列长度为 Integer.MAX\_VALUE ，可能堆积大量的请求，从而导致OOM。

CachedThreadPool 和 ScheduledThreadPool ： 允许创建的线程数量为 Integer.MAX\_VALUE ，可能会创建大量线程，从而导致OOM。

### Executor 框架的使用示意

1. \*\*主线程首先要创建实现 `Runnable` 或者 `Callable` 接口的任务对象。\*\*

2. \*\*把创建完成的实现 `Runnable`/`Callable`接口的 对象直接交给 `ExecutorService` 执行\*\*: `ExecutorService.execute（Runnable command）`）或者也可以把 `Runnable` 对象或`Callable` 对象提交给 `ExecutorService` 执行（`ExecutorService.submit（Runnable task）`或 `ExecutorService.submit（Callable <T> task）`）。

3. \*\*如果执行 `ExecutorService.submit（…）`，`ExecutorService` 将返回一个实现`Future`接口的对象\*\*（我们刚刚也提到过了执行 `execute()`方法和 `submit()`方法的区别，`submit()`会返回一个 `FutureTask 对象）。由于 FutureTask` 实现了 `Runnable`，我们也可以创建 `FutureTask`，然后直接交给 `ExecutorService` 执行。

4. \*\*最后，主线程可以执行 `FutureTask.get()`方法来等待任务执行完成。主线程也可以执行 `FutureTask.cancel（boolean mayInterruptIfRunning）`来取消此任务的执行。\*\*

### 线程池有什么优点？

1、降低资源消耗：重用存在的线程，减少对象创建销毁的开销。

2、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。

3、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。

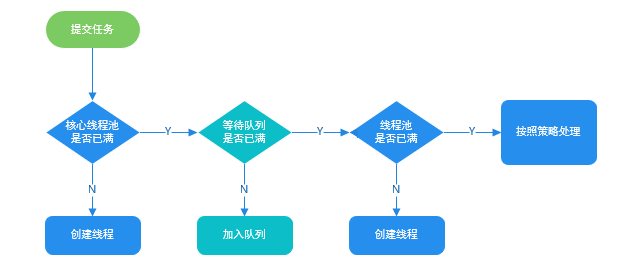
SHUTDOWN：不接受新的任务提交，但是会继续处理等待队列中的任务。

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。

### 线程池原理



### 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）

- \*\*CPU 密集型任务(N+1)：\*\* 这种任务消耗的主要是 CPU 资源，可以将线程数设置为 N（CPU 核心数）+1，比 CPU 核心数多出来的一个线程是为了防止线程偶发的缺页中断，或者其它原因导致的任务暂停而带来的影响。一旦任务暂停，CPU 就会处于空闲状态，而在这种情况下多出来的一个线程就可以充分利用 CPU 的空闲时间。

- \*\*I/O 密集型任务(2N)：\*\* 这种任务应用起来，系统会用大部分的时间来处理 I/O 交互，而线程在处理 I/O 的时间段内不会占用 CPU 来处理，这时就可以将 CPU 交出给其它线程使用。因此在 I/O 密集型任务的应用中，我们可以多配置一些线程，具体的计算方法是 2N。

### execute和submit的区别

1、有没有返回值`execute()`方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功与否；`submit()`方法用于提交需要返回值的任务。线程池会返回一个 `Future` 类型的对象，通过这个 `Future` 对象可以判断任务是否执行成功

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）

AtomicMarkableReference ：原子更新带有标记位的引用类型

### 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。

AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

CAS的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。UnSafe 类的 objectFieldOffset() 方法是一个本地方法，这个方法是用来拿到“原来的值”的内存地址，返回值是 valueOffset。另外 value 是一个volatile变量，在内存中可见，因此 JVM 可以保证任何时刻任何线程总能拿到该变量的最新值。

### AtomicInteger 类常用方法

public final int get() //获取当前的值

public final int getAndSet(int newValue)//获取当前的值，并设置新的值

public final int getAndIncrement()//获取当前的值，并自增

public final int getAndDecrement() //获取当前的值，并自减

public final int getAndAdd(int delta) //获取当前的值，并加上预期的值

boolean compareAndSet(int expect, int update) //如果输入的数值等于预期值，则以原子方式将该值设置为输入值（update）

public final void lazySet(int newValue)//最终设置为newValue,使用 lazySet 设置之后可能导致其他线程在之后的一小段时间内还是可以读到旧的值。

# Java虚拟机

## JDK 和 JRE 有什么区别？

Java 虚拟机（JVM）是运行 Java 字节码的虚拟机。JVM 有针对不同系统的特定实现（Windows，Linux，macOS），目的是使用相同的字节码，它们都会给出相同的结果。字节码和不同系统的 JVM 实现是 Java 语言“一次编译，随处可以运行”的关键所在。

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。它拥有 JRE 所拥有的一切，还有编译器（javac）和工具（如 javadoc 和 jdb）。它能够创建和编译程序。

JRE：Java Runtime Environment 的简称，java 运行环境，为 java 的运行提供了所需环境。包括 Java 虚拟机（JVM），Java 类库，java 命令和其他的一些基础构件。但是，它不能用于创建新程序。

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。

## Oracle JDK 和 OpenJDK 的对比

Oracle JDK 大概每 6 个月发一次主要版本，而 OpenJDK 版本大概每三个月发布一次。但这不是固定的，我觉得了解这个没啥用处。

OpenJDK 是一个参考模型并且是完全开源的，而 Oracle JDK 是 OpenJDK 的一个实现，并不是完全开源的；

Oracle JDK 比 OpenJDK 更稳定。OpenJDK 和 Oracle JDK 的代码几乎相同，但 Oracle JDK 有更多的类和一些错误修复。因此，如果您想开发企业/商业软件，我建议您选择 Oracle JDK，因为它经过了彻底的测试和稳定。某些情况下，有些人提到在使用 OpenJDK 可能会遇到了许多应用程序崩溃的问题，但是，只需切换到 Oracle JDK 就可以解决问题；

在响应性和 JVM 性能方面，Oracle JDK 与 OpenJDK 相比提供了更好的性能；

Oracle JDK 不会为即将发布的版本提供长期支持，用户每次都必须通过更新到最新版本获得支持来获取最新版本；

## 编译与解释并存

Oracle JDK 根据二进制代码许可协议获得许可，而 OpenJDK 根据 GPL v2 许可获得许可。

高级编程语言按照程序的执行方式分为编译型和解释型两种。简单来说，编译型语言是指编译器针对特定的操作系统将源代码一次性翻译成可被该平台执行的机器码；解释型语言是指解释器对源程序逐行解释成特定平台的机器码并立即执行。比如，你想阅读一本英文名著，你可以找一个英文翻译人员帮助你阅读， 有两种选择方式，你可以先等翻译人员将全本的英文名著（也就是源码）都翻译成汉语，再去阅读，也可以让翻译人员翻译一段，你在旁边阅读一段，慢慢把书读完。

Java 语言既具有编译型语言的特征，也具有解释型语言的特征，因为 Java 程序要经过先编译，后解释两个步骤，由 Java 编写的程序需要先经过编译步骤，生成字节码（\*.class 文件），这种字节码必须由 Java 解释器来解释执行。因此，我们可以认为 Java 语言编译与解释并存。

## 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；确保加载的类符合 JVM 规范和安全，保证被校验类的方法在运行时不会做出危害虚拟机的事件，其实就是一个安全检查为static变量在方法区中分配内存空间，设置变量的初始值，例如 static int a = 3 （注意：准备阶段只设置类中的静态变量（方法区中），不包括实例变量（堆内存中），实例变量是对象初始化时赋值的）

- 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。初始化其实就是一个赋值的操作，它会执行一个类构造器的<<clinit>clinit>()方法。由编译器自动收集类中所有变量的赋值动作，此时准备阶段时的那个 static int a = 3 的例子，在这个时候就正式赋值为3

## 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

两个子系统和两个组件：

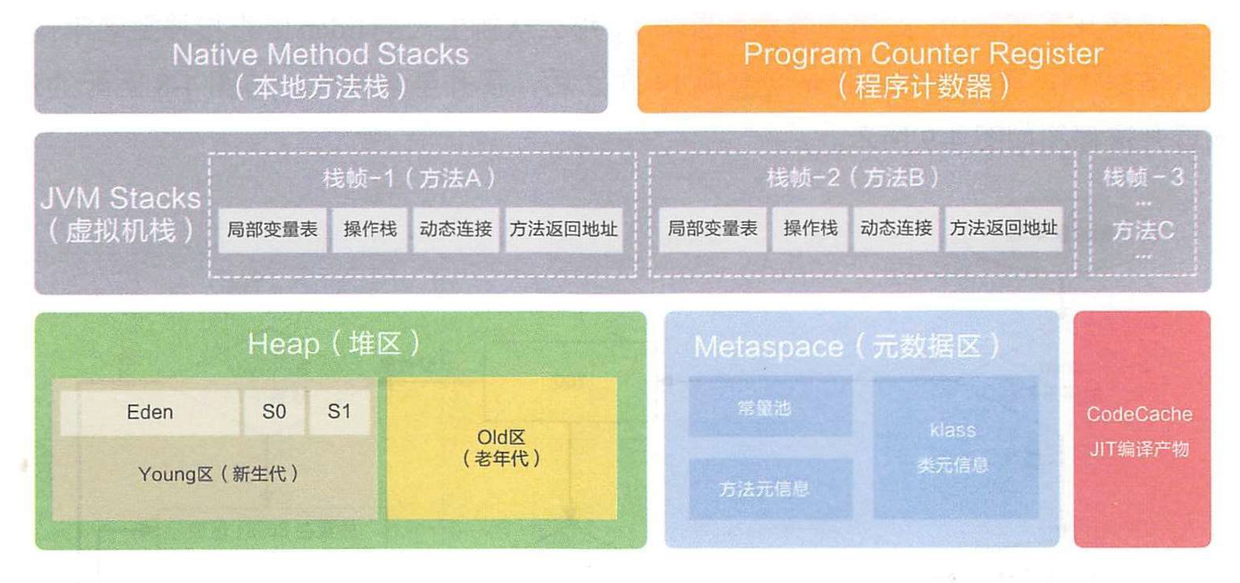
- Class loader(类装载)：根据给定的全限定名类名(如：java.lang.Object)来装载class文件到Runtime data area中的method area。

- 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。



1、- 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；

程序计数器其实就是一个指针，它指向了我们程序中下一句需要执行的指令，它也是内存区域中唯一一个不会出现OutOfMemoryError的区域，而且占用内存空间小到基本可以忽略不计。这个内存仅代表当前线程所执行的字节码的行号指示器，字节码解析器通过改变这个计数器的值选取下一条需要执行的字节码指令。

2、- 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；

如果线程请求的栈的深度大于虚拟机栈的最大深度，就会报 \*\*StackOverflowError\*\* （这种错误经常出现在递归中）。Java虚拟机也可以动态扩展，但随着扩展会不断地申请内存，当无法申请足够内存时就会报错 \*\*OutOfMemoryError\*\*。

对于栈来说，不存在垃圾回收。只要程序运行结束，栈的空间自然就会释放了。栈的生命周期和所处的线程是一致的。

这里补充一句：8种基本类型的变量+对象的引用变量+实例方法都是在栈里面分配内存。

3、- 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；

4、- 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

5、- 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。

## 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。

初始标记 ：在这个阶段，需要虚拟机停顿正在执行的任务，官方的叫法STW(Stop The Word)。这个过程从垃圾回收的"根对象"开始，只扫描到能够和"根对象"直接关联的对象，并作标记。所以这个过程虽然暂停了整个JVM，但是很快就完成了。

并发标记 ：这个阶段紧随初始标记阶段，在初始标记的基础上继续向下追溯标记。并发标记阶段，应用程序的线程和并发标记的线程并发执行，所以用户不会感受到停顿。

并发预清理 ：并发预清理阶段仍然是并发的。在这个阶段，虚拟机查找在执行并发标记阶段新进入老年代的对象(可能会有一些对象从新生代晋升到老年代， 或者有一些对象被分配到老年代)。通过重新扫描，减少下一个阶段"重新标记"的工作，因为下一个阶段会Stop The World。

重新标记 ：这个阶段会暂停虚拟机，收集器线程扫描在CMS堆中剩余的对象。扫描从"跟对象"开始向下追溯，并处理对象关联。

并发清理 ：清理垃圾对象，这个阶段收集器线程和应用程序线程并发执行。

并发重置 ：这个阶段，重置CMS收集器的数据结构，等待下一次垃圾回收。

### 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

#### 特点

并行与并发：G1能充分利用多CPU、多核环境下的硬件优势，使用多个CPU来缩短Stop-The-World停顿时间。部分收集器原本需要停顿Java线程来执行GC动作，G1收集器仍然可以通过并发的方式让Java程序继续运行。

分代收集：G1能够独自管理整个Java堆，并且采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次GC的旧对象以获取更好的收集效果。

空间整合：G1运作期间不会产生空间碎片，收集后能提供规整的可用内存。

可预测的停顿：G1除了追求低停顿外，还能建立可预测的停顿时间模型。能让使用者明确指定在一个长度为M毫秒的时间段内，消耗在垃圾收集上的时间不得超过N毫秒。

工作过程：

针对eden进行收集，数据移动到survivor区

初始标记：仅标记GC Roots能直接到的对象，并且修改TAMS（Next Top at Mark Start）的值，让下一阶段用户程序并发运行时，能在正确可用的Region中创建新对象。（需要线程停顿，但耗时很短。）

并发标记：从GC Roots开始对堆中对象进行可达性分析，找出存活对象。（耗时较长，但可与用户程序并发执行）

最终标记：为了修正在并发标记期间因用户程序执行而导致标记产生变化的那一部分标记记录。且对象的变化记录在线程Remembered Set Logs里面，把Remembered Set Logs里面的数据合并到Remembered Set中。（需要线程停顿，但可并行执行。）

筛选回收：对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划。（可并发执行）



### 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。

### 将数组转换为ArrayList

最简便的方法(推荐)

List list = new ArrayList<>(Arrays.asList("a", "b", "c"))

使用 Java8 的Stream(推荐)

Integer [] myArray = { 1, 2, 3 };

List myList = Arrays.stream(myArray).collect(Collectors.toList());

//基本类型也可以实现转换（依赖boxed的装箱操作）

int [] myArray2 = { 1, 2, 3 };

List myList = Arrays.stream(myArray2).boxed().collect(Collectors.toList());

## 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）

## 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()

如果要进行remove操作，可以调用迭代器的 remove 方法而不是集合类的 remove 方法。因为如果列表在任何时间从结构上修改创建迭代器之后，以任何方式除非通过迭代器自身remove/add方法，迭代器都将抛出一个ConcurrentModificationException,这就是单线程状态下产生的 fail-fast 机制。

fail-fast 机制 ：多个线程对 fail-fast 集合进行修改的时，可能会抛出ConcurrentModificationException，单线程下也会出现这种情况，上面已经提到过。

java.util包下面的所有的集合类都是fail-fast的，而java.util.concurrent包下面的所有的类都是fail-safe的。

### 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。

### 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。

### 安全失败(fail-safe)

明白了快速失败(fail-fast)之后，安全失败(fail-safe)我们就很好理解了。

采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。所以，在遍历过程中对原集合所作的修改并不能被迭代器检测到，故不会抛 `ConcurrentModificationException` 异常。

## Collections 和 Arrays

### Collections

Collections 工具类常用方法:

### 排序

查找,替换操作

同步控制(不推荐，需要线程安全的集合类型时请考虑使用 JUC 包下的并发集合)

排序操作

void reverse(List list)//反转

void shuffle(List list)//随机排序

void sort(List list)//按自然排序的升序排序

void sort(List list, Comparator c)//定制排序，由Comparator控制排序逻辑

void swap(List list, int i , int j)//交换两个索引位置的元素

void rotate(List list, int distance)//旋转。当distance为正数时，将list后distance个元素整体移到前面。当distance为负数时，将 list的前distance个元素整体移到后面。

### 查找,替换操作

int binarySearch(List list, Object key)//对List进行二分查找，返回索引，注意List必须是有序的

int max(Collection coll)//根据元素的自然顺序，返回最大的元素。 类比int min(Collection coll)

int max(Collection coll, Comparator c)//根据定制排序，返回最大元素，排序规则由Comparatator类控制。类比int min(Collection coll, Comparator c)

void fill(List list, Object obj)//用指定的元素代替指定list中的所有元素。

int frequency(Collection c, Object o)//统计元素出现次数

int indexOfSubList(List list, List target)//统计target在list中第一次出现的索引，找不到则返回-1，类比int lastIndexOfSubList(List source, list target).

boolean replaceAll(List list, Object oldVal, Object newVal), 用新元素替换旧元素

### 同步控制

Collections提供了多个synchronizedXxx()方法·，该方法可以将指定集合包装成线程同步的集合，从而解决多线程并发访问集合时的线程安全问题。

我们知道 HashSet，TreeSet，ArrayList,LinkedList,HashMap,TreeMap 都是线程不安全的。Collections提供了多个静态方法可以把他们包装成线程同步的集合。

最好不要用下面这些方法，效率非常低，需要线程安全的集合类型时请考虑使用 JUC 包下的并发集合。

方法如下：

synchronizedCollection(Collection<T> c) //返回指定 collection 支持的同步（线程安全的）collection。

synchronizedList(List<T> list)//返回指定列表支持的同步（线程安全的）List。

synchronizedMap(Map<K,V> m) //返回由指定映射支持的同步（线程安全的）Map。

synchronizedSet(Set<T> s) //返回指定 set 支持的同步（线程安全的）set。

### Collections设置不可变集合

emptyXxx(): 返回一个空的、不可变的集合对象，此处的集合既可以是List，也可以是Set，还可以是Map。

singletonXxx(): 返回一个只包含指定对象（只有一个或一个元素）的不可变的集合对象，此处的集合可以是：List，Set，Map。

unmodifiableXxx(): 返回指定集合对象的不可变视图，此处的集合可以是：List，Set，Map。

上面三类方法的参数是原有的集合对象，返回值是该集合的”只读“版本。

### Arrays类的常见操作

排序 : sort()

查找 : binarySearch()

比较: equals()

填充 : fill()

转列表: asList()

Arrays.asList()将数组转换为集合后,底层其实还是数组

传递的数组必须是对象数组，而不是基本类型。

使用集合的修改方法:add()、remove()、clear()会抛出异常。运行时报错：UnsupportedOperationException

Arrays.asList() 方法返回的并不是 java.util.ArrayList ，而是 java.util.Arrays 的一个内部类,这个内部类并没有实现集合的修改方法或者说并没有重写这些方法。

转字符串 : toString()

复制: copyOf()

## List

### Arraylist 与 LinkedList 区别?

1. \*\*是否保证线程安全：\*\* `ArrayList` 和 `LinkedList` 都是不同步的，也就是不保证线程安全；

2. \*\*底层数据结构：\*\* `Arraylist` 底层使用的是 \*\*`Object` 数组\*\*；`LinkedList` 底层使用的是 \*\*双向链表\*\* 数据结构（JDK1.6 之前为循环链表，JDK1.7 取消了循环。注意双向链表和双向循环链表的区别，下面有介绍到！）

3. \*\*插入和删除是否受元素位置的影响：\*\* ① \*\*`ArrayList` 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。\*\* 比如：执行`add(E e)`方法的时候， `ArrayList` 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是 O(1)。但是如果要在指定位置 i 插入和删除元素的话（`add(int index, E element)`）时间复杂度就为 O(n-i)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向前移一位的操作。 ② \*\*`LinkedList` 采用链表存储，所以对于`add(E e)`方法的插入，删除元素时间复杂度不受元素位置的影响，近似 O(1)，如果是要在指定位置`i`插入和删除元素的话（`(add(int index, E element)`） 时间复杂度近似为`o(n))`因为需要先移动到指定位置再插入。\*\*

4. \*\*是否支持快速随机访问：\*\* `LinkedList` 不支持高效的随机元素访问，而 `ArrayList` 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于`get(int index)`方法)。

5. \*\*内存空间占用：\*\* ArrayList 的空 间浪费主要体现在在 list 列表的结尾会预留一定的容量空间，而 LinkedList 的空间花费则体现在它的每一个元素都需要消耗比 ArrayList 更多的空间（因为要存放直接后继和直接前驱以及数据）。

## ArrayList

ArrayList 的底层是数组队列，相当于动态数组。与 Java 中的数组相比，它的容量能动态增长。在添加大量元素前，应用程序可以使用`ensureCapacity`操作来增加 ArrayList 实例的容量。这可以减少递增式再分配的数量。

它继承于 AbstractList，实现了 List, RandomAccess, Cloneable, java.io.Serializable 这些接口。

在我们学数据结构的时候就知道了线性表的顺序存储，插入删除元素的时间复杂度为O（n）,求表长以及增加元素，取第 i 元素的时间复杂度为O（1）

　 ArrayList 继承了AbstractList，实现了List。它是一个数组队列，提供了相关的添加、删除、修改、遍历等功能。

　　ArrayList 实现了RandomAccess 接口， RandomAccess 是一个标志接口，表明实现这个这个接口的 List 集合是支持快速随机访问的。在 ArrayList 中，我们即可以通过元素的序号快速获取元素对象，这就是快速随机访问。

　　ArrayList 实现了Cloneable 接口，即覆盖了函数 clone()，能被克隆。

　　ArrayList 实现java.io.Serializable 接口，这意味着ArrayList支持序列化，能通过序列化去传输。

　　和 Vector 不同，ArrayList 中的操作不是线程安全的！所以，建议在单线程中才使用 ArrayList，而在多线程中可以选择 Vector 或者 CopyOnWriteArrayList。

### 优点：

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。

### 缺点：

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；

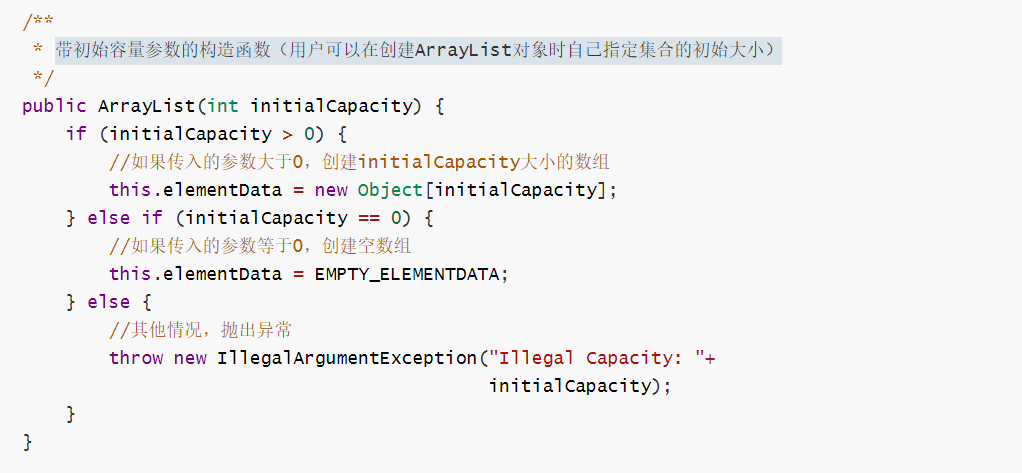
综合来说，在需要频繁读取集合中的元素时，更推荐使用 ArrayList，而在插入和删除操作较多时，更推荐使用 LinkedList

### 核心源码

默认初始容量大小 10，使用Object[]保存数据

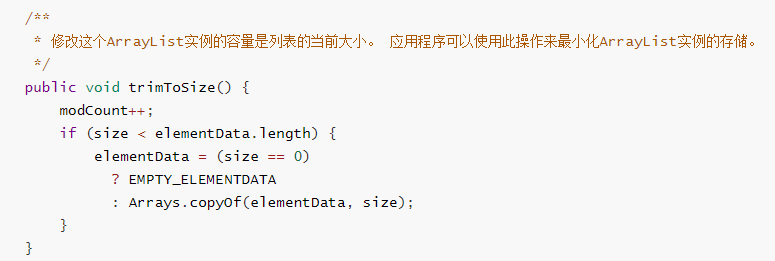
默认无参构造函数

\*DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA 为0.初始化为10，也就是说初始其实是空数组 当添加第一个元素的时候数组容量才变成10



通过集合来创建

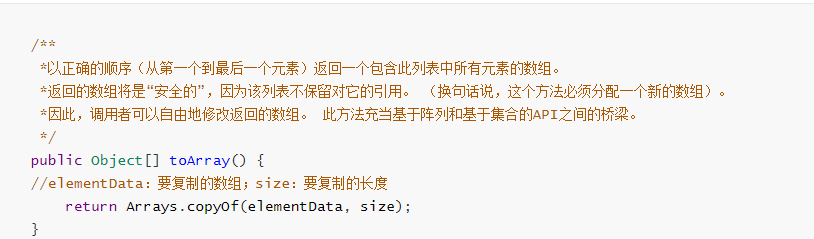




#### System.arraycopy()和Arrays.copyOf()方法

　　通过上面源码我们发现这两个实现数组复制的方法被广泛使用而且很多地方都特别巧妙。比如下面<font color="red">add(int index, E element)</font>方法就很巧妙的用到了<font color="red">arraycopy()方法</font>让数组自己复制自己实现让index开始之后的所有成员后移一个位置:





##### 两者联系与区别

联系：

看两者源代码可以发现`copyOf()`内部调用了`System.arraycopy()`方法

区别：

1. arraycopy()需要目标数组，将原数组拷贝到你自己定义的数组里，而且可以选择拷贝的起点和长度以及放入新数组中的位置

2. copyOf()是系统自动在内部新建一个数组，并返回该数组。

### 扩容

以无参数构造方法创建 ArrayList 时，实际上初始化赋值的是一个空数组。当真正对数组进行添加元素操作时，才真正分配容量。即向数组中添加第一个元素时，数组容量扩为10。

- 当我们要 add 进第1个元素到 ArrayList 时，elementData.length 为0 （因为还是一个空的 list），因为执行了 `ensureCapacityInternal()` 方法 ，所以 minCapacity 此时为10。此时，`minCapacity - elementData.length > 0 `成立，所以会进入 `grow(minCapacity)` 方法。

- 当add第2个元素时，minCapacity 为2，此时e lementData.length(容量)在添加第一个元素后扩容成 10 了。此时，`minCapacity - elementData.length > 0 ` 不成立，所以不会进入 （执行）`grow(minCapacity)` 方法。

- 添加第3、4···到第10个元素时，依然不会执行grow方法，数组容量都为10。

直到添加第11个元素，minCapacity(为11)比elementData.length（为10）要大。进入grow方法进行扩容。

在add方法的时候判断，size+1是否超出当前容量

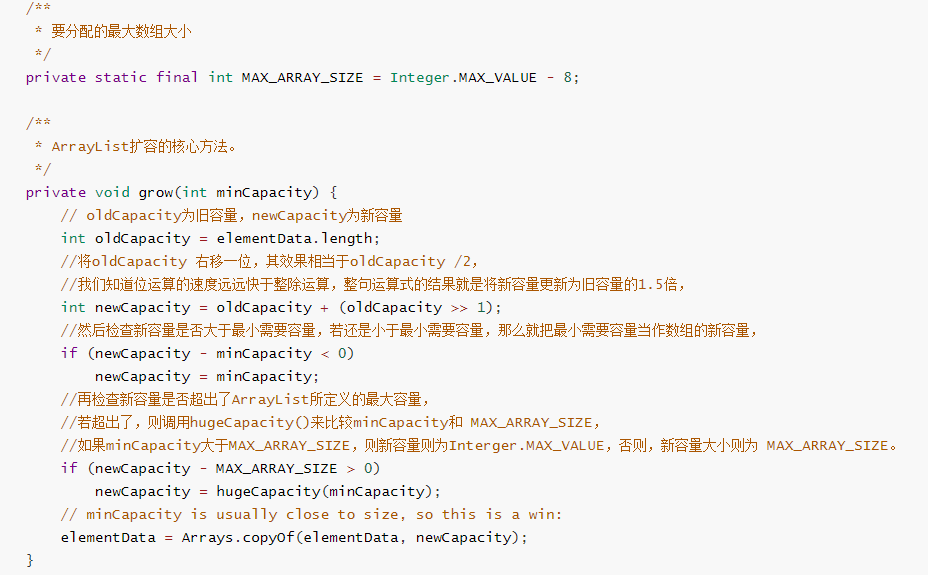
将oldCapacity 右移一位，其效果相当于oldCapacity /2，

我们知道位运算的速度远远快于整除运算，整句运算式的结果就是将新容量更新为旧容量的1.5倍，

这是个给用户调用的方法：

最好在 add 大量元素之前用 `ensureCapacity` 方法，以减少增量重新分配的次数





扩容机制代码已经做了详细的解释。另外值得注意的是大家很容易忽略的一个运算符：移位运算符

　　简介：移位运算符就是在二进制的基础上对数字进行平移。按照平移的方向和填充数字的规则分为三种:<font color="red"><<(左移)</font>、<font color="red">>>(带符号右移)</font>和<font color="red">>>>(无符号右移)</font>。

　　作用：对于大数据的2进制运算,位移运算符比那些普通运算符的运算要快很多,因为程序仅仅移动一下而已,不去计算,这样提高了效率,节省了资源

　　比如这里：int newCapacity = oldCapacity + (oldCapacity >> 1);

右移一位相当于除2，右移n位相当于除以 2 的 n 次方。这里 oldCapacity 明显右移了1位所以相当于oldCapacity /2。

另外需要注意的是：

1. java 中的\*\*length 属性\*\*是针对数组说的,比如说你声明了一个数组,想知道这个数组的长度则用到了 length 这个属性.

2. java 中的\*\*length()方法\*\*是针对字 符串String说的,如果想看这个字符串的长度则用到 length()这个方法.

3. .java 中的\*\*size()方法\*\*是针对泛型集合说的,如果想看这个泛型有多少个元素,就调用此方法来查看!

### 内部类

(1)private class Itr implements Iterator<E>

(2)private class ListItr extends Itr implements ListIterator<E>

(3)private class SubList extends AbstractList<E> implements RandomAccess

(4)static final class ArrayListSpliterator<E> implements Spliterator<E>

　　ArrayList有四个内部类，其中的Itr是实现了Iterator接口，同时重写了里面的hasNext()， next()， remove() 等方法；其中的ListItr 继承 Itr，实现了ListIterator接口，同时重写了hasPrevious()， nextIndex()， previousIndex()， previous()， set(E e)， add(E e) 等方法，所以这也可以看出了

Iterator和ListIterator的区别: ListIterator在Iterator的基础上增加了添加对象，修改对象，逆向遍历等方法，这些是Iterator不能实现的。

## LinkedList

List接口和Deque接口的双端链表。

LinkedList底层的链表结构使它支持高效的插入和删除操作，另外它实现了Deque接口，使得LinkedList类也具有队列的特性;

LinkedList不是线程安全的，如果想使LinkedList变成线程安全的，可以调用静态类Collections类中的synchronizedLis方法：

### 内部私有类Node



### Add

将元素添加到链表尾部

add(int index,E e)：在指定位置添加元素

addAll(Collection c )：将集合插入到链表尾部

addAll(int index, Collection c)： 将集合从指定位置开始插入

上面可以看出addAll方法通常包括下面四个步骤：

1. 检查index范围是否在size之内

2. toArray()方法把集合的数据存到对象数组中

3. 得到插入位置的前驱和后继节点

4. 遍历数据，将数据插入到指定位置

addFirst(E e)： 将元素添加到链表头部

### Get

\*\*get(int index)：\*\* 根据指定索引返回数据

获取头节点（index=0）数据方法:

getFirst(),element(),peek(),peekFirst()

这四个获取头结点方法的区别在于对链表为空时的处理，是抛出异常还是返回null，其中\*\*getFirst()\*\* 和\*\*element()\*\* 方法将会在链表为空时，抛出异常

element()方法的内部就是使用getFirst()实现的。它们会在链表为空时，抛出NoSuchElementException

获取尾节点（index=-1）数据方法:

\*\*getLast()\*\* 方法在链表为空时，会抛出\*\*NoSuchElementException\*\*，而\*\*peekLast()\*\* 则不会，只是会返回 \*\*null\*\*。

### 根据对象得到索引的方法

int indexOf(Object o)：\*\* 从头遍历找

\*\*int lastIndexOf(Object o)：\*\* 从尾遍历找

### 删除方法

\*\*remove()\*\* ,\*\*removeFirst(),pop():\*\* 删除头节点

\*\*removeLast(),pollLast():\*\* 删除尾节点

removeLast()在链表为空时将抛出NoSuchElementException，而pollLast()方法返回null。

\*\*remove(Object o):\*\* 删除指定元素

当删除指定对象时，只需调用remove(Object o)即可，不过该方法一次只会删除一个匹配的对象，如果删除了匹配对象，返回true，否则false。

\*\*remove(int index)\*\*：删除指定位置的元素

## CopyOnWriteArrayList

### CopyOnWriteArrayList 简介

```java

public class CopyOnWriteArrayList<E>

extends Object

implements List<E>, RandomAccess, Cloneable, Serializable

```

在很多应用场景中，读操作可能会远远大于写操作。由于读操作根本不会修改原有的数据，因此对于每次读取都进行加锁其实是一种资源浪费。我们应该允许多个线程同时访问 List 的内部数据，毕竟读取操作是安全的。

这和我们之前在多线程章节讲过 `ReentrantReadWriteLock` 读写锁的思想非常类似，也就是读读共享、写写互斥、读写互斥、写读互斥。JDK 中提供了 `CopyOnWriteArrayList` 类比相比于在读写锁的思想又更进一步。为了将读取的性能发挥到极致，`CopyOnWriteArrayList` 读取是完全不用加锁的，并且更厉害的是：写入也不会阻塞读取操作。只有写入和写入之间需要进行同步等待。这样一来，读操作的性能就会大幅度提升。\*\*那它是怎么做的呢？\*\*

### CopyOnWriteArrayList 是如何做到的？

`CopyOnWriteArrayList` 类的所有可变操作（add，set 等等）都是通过创建底层数组的新副本来实现的。当 List 需要被修改的时候，我并不修改原有内容，而是对原有数据进行一次复制，将修改的内容写入副本。写完之后，再将修改完的副本替换原来的数据，这样就可以保证写操作不会影响读操作了。

从 `CopyOnWriteArrayList` 的名字就能看出`CopyOnWriteArrayList` 是满足`CopyOnWrite` 的 ArrayList，所谓`CopyOnWrite` 也就是说：在计算机，如果你想要对一块内存进行修改时，我们不在原有内存块中进行写操作，而是将内存拷贝一份，在新的内存中进行写操作，写完之后呢，就将指向原来内存指针指向新的内存，原来的内存就可以被回收掉了。

### CopyOnWriteArrayList 读取操作的实现

读取操作没有任何同步控制和锁操作，理由就是内部数组 array 不会发生修改，只会被另外一个 array 替换，因此可以保证数据安全。

### CopyOnWriteArrayList 写入操作的实现

CopyOnWriteArrayList 写入操作 add() 方法在添加集合的时候加了锁，保证了同步，避免了多线程写的时候会 copy 出多个副本出来。

## 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

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);

}

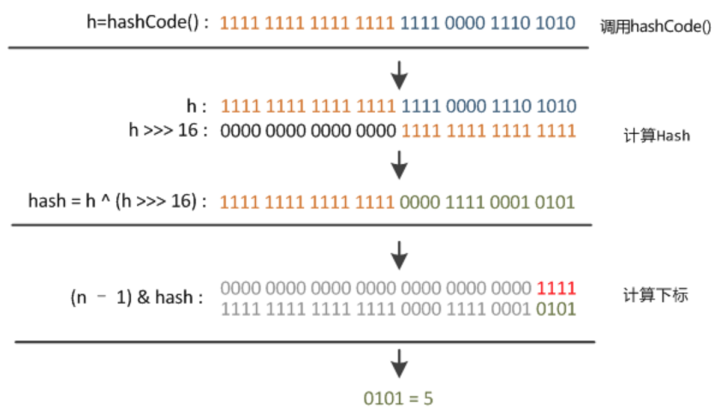
方法二：

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

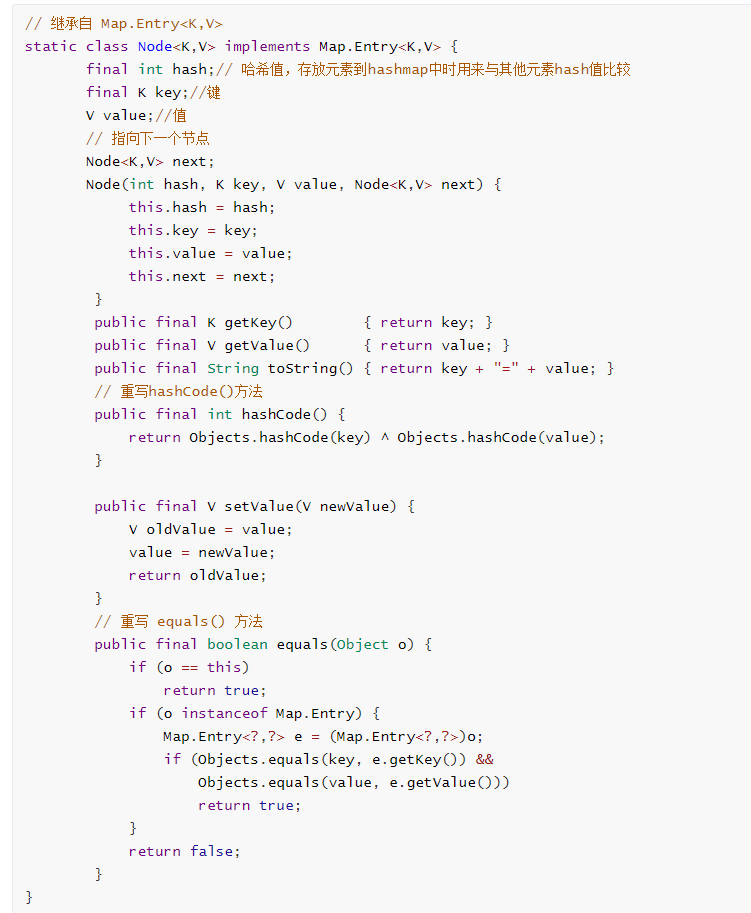
return h & (length-1); //第三步 取模运算

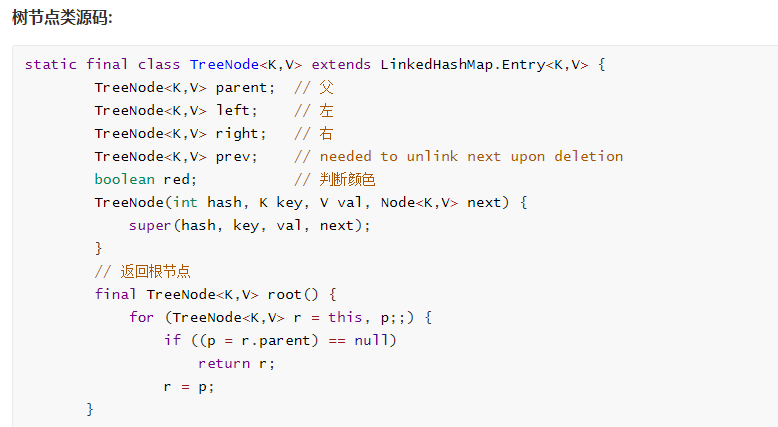
}

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。



### Node节点类源码:





### 构造方法

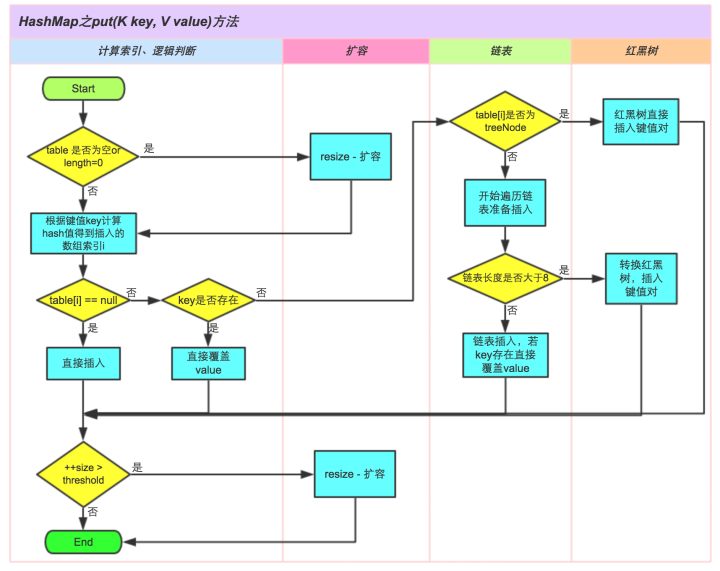
默认构造函数。

包含另一个“Map”的构造函数

指定“容量大小”的构造函数

指定“容量大小”和“加载因子”的构造函数

### 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。

对比一下 JDK1.7 put方法的代码

\*\*对于put方法的分析如下：\*\*

- ①如果定位到的数组位置没有元素 就直接插入。

- ②如果定位到的数组位置有元素，遍历以这个元素为头结点的链表，依次和插入的key比较，如果key相同就直接覆盖，不同就采用头插法插入元素。

### 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。

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

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

int oldCapacity = oldTable.length;

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

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

return;

}

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

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

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

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

}

void transfer(Entry[] newTable) {

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

int newCapacity = newTable.length;

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

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

if (e != null) {

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）

do {

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

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

e.next = newTable[i]; //标记[1]

newTable[i] = e; //将元素放在数组上

e = next; //访问下一个Entry链上的元素

} while (e != null);

}

}

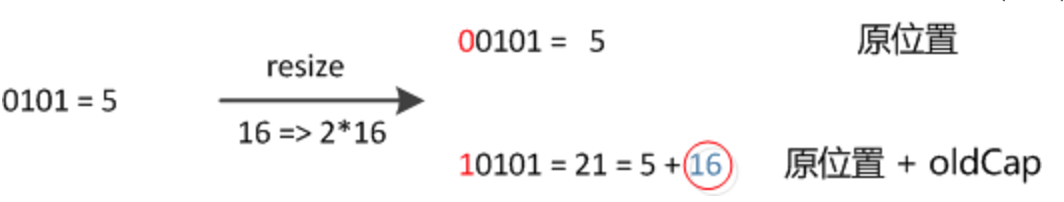
}

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。

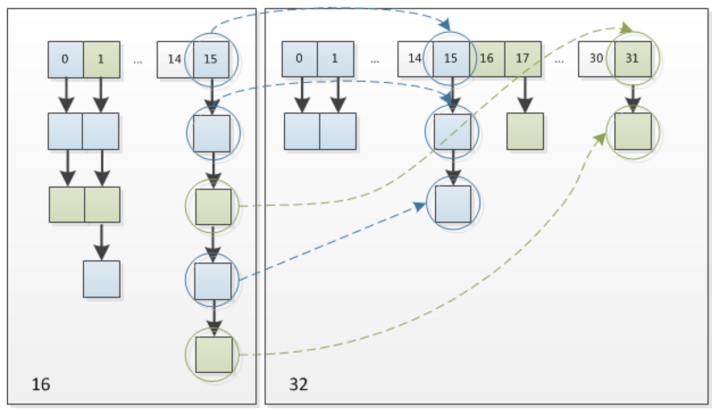
下面我们讲解下JDK1.8做了哪些优化。经过观测可以发现，我们使用的是2次幂的扩展(指长度扩为原来2倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。看下图可以明白这句话的意思，n为table的长度，图（a）表示扩容前的key1和key2两种key确定索引位置的示例，图（b）表示扩容后key1和key2两种key确定索引位置的示例，其中hash1是key1对应的哈希与高位运算结果。



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)：

### HashMap 的长度为什么是 2 的幂次方

为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀。我们上面也讲到了过了，Hash 值的范围值-2147483648 到 2147483647，前后加起来大概 40 亿的映射空间，只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个 40 亿长度的数组，内存是放不下的。所以这个散列值是不能直接拿来用的。用之前还要先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数组下标。这个数组下标的计算方法是“ `(n - 1) & hash`”。（n 代表数组长度）。这也就解释了 HashMap 的长度为什么是 2 的幂次方。

\*\*这个算法应该如何设计呢？\*\*

得到低位值确定元素在hash数组中的位置，就是为了提高hash数组的利用率。

计算hash值，使用2次幂-1取余的方式是为了让数组中的元素分布均匀，充分利用数组空间。

我们首先可能会想到采用%取余的操作来实现。但是，重点来了：\*\*“取余(%)操作中如果除数是 2 的幂次则等价于与其除数减一的与(&)操作（也就是说 hash%length==hash&(length-1)的前提是 length 是 2 的 n 次方；）。” 并且 采用二进制位操作 &，相对于%能够提高运算效率，这就解释了 HashMap 的长度为什么是 2 的幂次方。\*\*

### ****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。

### java1.7

Java 7 中 ConcurrentHashMap 的存储结构如上图，ConcurrnetHashMap 由很多个 Segment 组合，而每一个 Segment 是一个类似于 HashMap 的结构，所以每一个 HashMap 的内部可以进行扩容。但是 Segment 的个数一旦\*\*初始化就不能改变\*\*，默认 Segment 的个数是 16 个，你也可以认为 ConcurrentHashMap 默认支持最多 16 个线程并发。

### 初始化

无参构造中调用了有参构造，传入了三个参数的默认值，他们的值是。

\* 默认初始化容量

static final int DEFAULT\_INITIAL\_CAPACITY = 16;

\* 默认负载因子

static final float DEFAULT\_LOAD\_FACTOR = 0.75f;

\* 默认并发级别

static final int DEFAULT\_CONCURRENCY\_LEVEL = 16;

1. 必要参数校验。

2. 校验并发级别 concurrencyLevel 大小，如果大于最大值，重置为最大值1<<16为 65536。无惨构造默认值是 16.

3. 寻找并发级别 concurrencyLevel 之上最近的 2 的幂次方值，作为初始化容量大小，默认是 16。

4. 记录 segmentShift 偏移量，这个值为【容量 = 2 的N次方】中的 N，在后面 Put 时计算位置时会用到。默认是 32 - sshift = 28.

5. 记录 segmentMask，默认是 ssize - 1 = 16 -1 = 15.

6. 初始化 segments[0]，默认大小为 2，负载因子 0.75，扩容阀值是 2\*0.75=1.5，插入第二个值时才会进行扩容。

### Put

1. 计算要 put 的 key 的位置，获取指定位置的 Segment。

2. 如果指定位置的 Segment 为空，则初始化这个 Segment.

\*\*初始化 Segment 流程：\*\*

1. 检查计算得到的位置的 Segment 是否为null.

2. 为 null 继续初始化，使用 Segment[0] 的容量和负载因子创建一个 HashEntry 数组。

3. 再次检查计算得到的指定位置的 Segment 是否为null.

4. 使用创建的 HashEntry 数组初始化这个 Segment.

5. 自旋判断计算得到的指定位置的 Segment 是否为null，使用 CAS 在这个位置赋值为 Segment.

3. Segment.put 插入 key,value 值。

由于 Segment 继承了 ReentrantLock，所以 Segment 内部可以很方便的获取锁，put 流程就用到了这个功能。

1. tryLock() 获取锁，获取不到使用 \*\*`scanAndLockForPut`\*\* 方法继续获取。

2. 计算 put 的数据要放入的 index 位置，然后获取这个位置上的 HashEntry 。

3. 遍历 put 新元素，为什么要遍历？因为这里获取的 HashEntry 可能是一个空元素，也可能是链表已存在，所以要区别对待。

如果这个位置上的 \*\*HashEntry 不存在\*\*：

1. 如果当前容量大于扩容阀值，小于最大容量，\*\*进行扩容\*\*。

2. 直接头插法插入。

如果这个位置上的 \*\*HashEntry 存在\*\*：

1. 判断链表当前元素 Key 和 hash 值是否和要 put 的 key 和 hash 值一致。一致则替换值

2. 不一致，获取链表下一个节点，直到发现相同进行值替换，或者链表表里完毕没有相同的。

1. 如果当前容量大于扩容阀值，小于最大容量，\*\*进行扩容\*\*。

2. 直接链表头插法插入。

4. 如果要插入的位置之前已经存在，替换后返回旧值，否则返回 null.

这里面的第一步中的 scanAndLockForPut 操作这里没有介绍，这个方法做的操作就是不断的自旋 `tryLock()` 获取锁。当自旋次数大于指定次数时，使用 `lock()` 阻塞获取锁。在自旋时顺表获取下 hash 位置的 HashEntry。

### 扩容 rehash

ConcurrentHashMap 的扩容只会扩容到原来的两倍。老数组里的数据移动到新的数组时，位置要么不变，要么变为 index+ oldSize，参数里的 node 会在扩容之后使用链表\*\*头插法\*\*插入到指定位置。

### get

到这里就很简单了，get 方法只需要两步即可。

1. 计算得到 key 的存放位置。

2. 遍历指定位置查找相同 key 的 value 值。

### java 1.8

\*\*Node 数组 + 链表 / 红黑树\*\*。当冲突链表达到一定长度时，链表会转换成红黑树。

### 初始化

从源码中可以发现 ConcurrentHashMap 的初始化是通过\*\*自旋和 CAS\*\* 操作完成的。里面需要注意的是变量 `sizeCtl` ，它的值决定着当前的初始化状态。

1. -1 说明正在初始化

2. -N 说明有N-1个线程正在进行扩容

3. 表示 table 初始化大小，如果 table 没有初始化

4. 表示 table 容量，如果 table　已经初始化。

### Put

1. 根据 key 计算出 hashcode 。

2. 判断是否需要进行初始化。

3. 即为当前 key 定位出的 Node，如果为空表示当前位置可以写入数据，利用 CAS 尝试写入，失败则自旋保证成功。

4. 如果当前位置的 `hashcode == MOVED == -1`,则需要进行扩容。

5. 如果都不满足，则利用 synchronized 锁写入数据。

6. 如果数量大于 `TREEIFY\_THRESHOLD` 则要转换为红黑树。

### Get

1. 根据 hash 值计算位置。

2. 查找到指定位置，如果头节点就是要找的，直接返回它的 value.

3. 如果头节点 hash 值小于 0 ，说明正在扩容或者是红黑树，查找之。

4. 如果是链表，遍历查找之。

### 总结

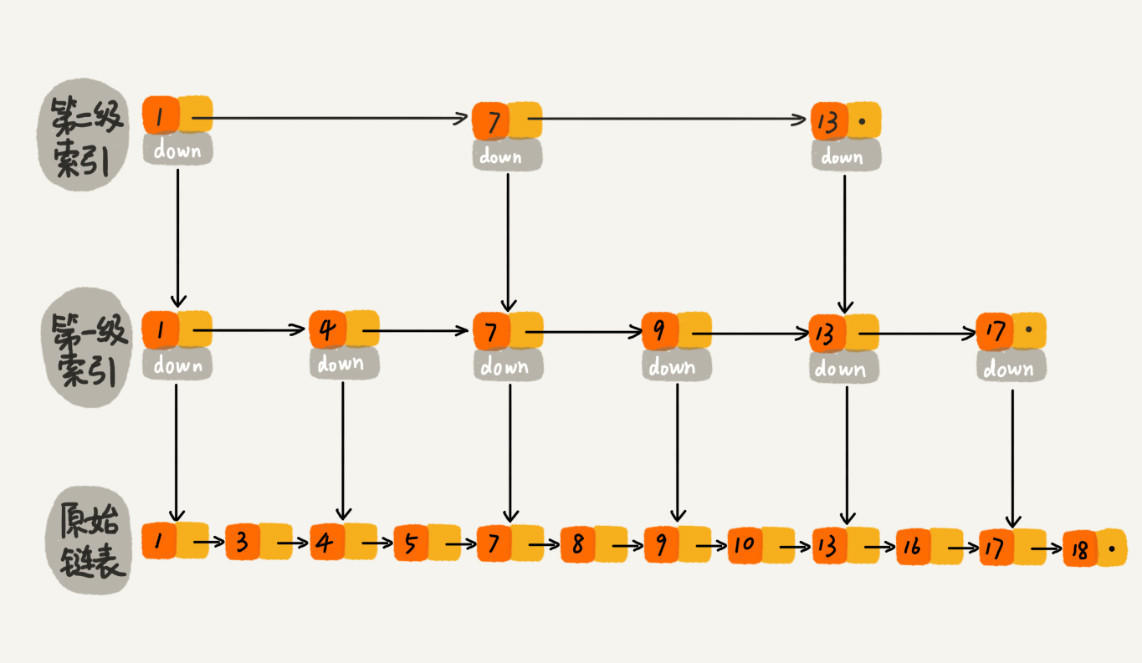
Java7 中 ConcruuentHashMap 使用的分段锁，也就是每一个 Segment 上同时只有一个线程可以操作，每一个 Segment 都是一个类似 HashMap 数组的结构，它可以扩容，它的冲突会转化为链表。但是 Segment 的个数一但初始化就不能改变。

Java8 中的 ConcruuentHashMap 使用的 Synchronized 锁加 CAS 的机制。结构也由 Java7 中的 \*\*Segment 数组 + HashEntry 数组 + 链表\*\* 进化成了 \*\*Node 数组 + 链表 / 红黑树\*\*，Node 是类似于一个 HashEntry 的结构。它的冲突再达到一定大小时会转化成红黑树，在冲突小于一定数量时又退回链表。

## ConcurrentSkipListMap

对于一个单链表，即使链表是有序的，如果我们想要在其中查找某个数据，也只能从头到尾遍历链表，这样效率自然就会很低，跳表就不一样了。跳表是一种可以用来快速查找的数据结构，有点类似于平衡树。它们都可以对元素进行快速的查找。但一个重要的区别是：对平衡树的插入和删除往往很可能导致平衡树进行一次全局的调整。而对跳表的插入和删除只需要对整个数据结构的局部进行操作即可。这样带来的好处是：在高并发的情况下，你会需要一个全局锁来保证整个平衡树的线程安全。而对于跳表，你只需要部分锁即可。这样，在高并发环境下，你就可以拥有更好的性能。而就查询的性能而言，跳表的时间复杂度也是 \*\*O(logn)\*\* 所以在并发数据结构中，JDK 使用跳表来实现一个 Map。

跳表的本质是同时维护了多个链表，并且链表是分层的，



低层的链表维护了跳表内所有的元素，每上面一层链表都是下面一层的子集。

跳表内的所有链表的元素都是排序的。查找时，可以从顶级链表开始找。一旦发现被查找的元素大于当前链表中的取值，就会转入下一层链表继续找。这也就是说在查找过程中，搜索是跳跃式的。如上图所示，在跳表中查找元素 18。

查找 18 的时候原来需要遍历 18 次，现在只需要 7 次即可。针对链表长度比较大的时候，构建索引查找效率的提升就会非常明显。

从上面很容易看出，\*\*跳表是一种利用空间换时间的算法。\*\*

使用跳表实现 Map 和使用哈希算法实现 Map 的另外一个不同之处是：哈希并不会保存元素的顺序，而跳表内所有的元素都是排序的。因此在对跳表进行遍历时，你会得到一个有序的结果。所以，如果你的应用需要有序性，那么跳表就是你不二的选择。JDK 中实现这一数据结构的类是 ConcurrentSkipListMap。

## Set

### comparable 和 Comparator 的区别

- `comparable` 接口实际上是出自`java.lang`包 它有一个 `compareTo(Object obj)`方法用来排序

- `comparator`接口实际上是出自 java.util 包它有一个`compare(Object obj1, Object obj2)`方法用来排序

一般我们需要对一个集合使用自定义排序时，我们就要重写`compareTo()`方法或`compare()`方法，当我们需要对某一个集合实现两种排序方式，比如一个 song 对象中的歌名和歌手名分别采用一种排序方法的话，我们可以重写`compareTo()`方法和使用自制的`Comparator`方法或者以两个 Comparator 来实现歌名排序和歌星名排序，第二种代表我们只能使用两个参数版的 `Collections.sort()`.

## HashSet

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。



### HashSet 如何检查重复

当你把对象加入`HashSet`时，HashSet 会先计算对象的`hashcode`值来判断对象加入的位置，同时也会与其他加入的对象的 hashcode 值作比较，如果没有相符的 hashcode，HashSet 会假设对象没有重复出现。但是如果发现有相同 hashcode 值的对象，这时会调用`equals()`方法来检查 hashcode 相等的对象是否真的相同。如果两者相同，HashSet 就不会让加入操作成功。（摘自我的 Java 启蒙书《Head fist java》第二版）

\*\*hashCode()与 equals()的相关规定：\*\*

1. 如果两个对象相等，则 hashcode 一定也是相同的

2. 两个对象相等,对两个 equals 方法返回 true

3. 两个对象有相同的 hashcode 值，它们也不一定是相等的

4. 综上，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖

5. hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写 hashCode()，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）。

\*\*==与 equals 的区别\*\*

对于基本类型来说，== 比较的是值是否相等；

对于引用类型来说，== 比较的是两个引用是否指向同一个对象地址（两者在内存中存放的地址（堆内存地址）是否指向同一个地方）；

对于引用类型（包括包装类型）来说，equals 如果没有被重写，对比它们的地址是否相等；如果 equals()方法被重写（例如 String），则比较的是地址里的内容。

## 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。

### 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。

### 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，

### 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。

### 阻塞队列

1、 ArrayBlockingQueue ：一个由数组支持的有界队列。 BlockingQueue 接口的有界队列实现类，底层采用\*\*数组\*\*来实现。ArrayBlockingQueue 一旦创建，容量不能改变。其并发控制采用可重入锁来控制，不管是插入操作还是读取操作，都需要获取到锁才能进行操作。当队列容量满时，尝试将元素放入队列将导致操作阻塞;尝试从一个空队列中取一个元素也会同样阻塞。

ArrayBlockingQueue 默认情况下不能保证线程访问队列的公平性，所谓公平性是指严格按照线程等待的绝对时间顺序，即最先等待的线程能够最先访问到 ArrayBlockingQueue。而非公平性则是指访问 ArrayBlockingQueue 的顺序不是遵守严格的时间顺序，有可能存在，当 ArrayBlockingQueue 可以被访问时，长时间阻塞的线程依然无法访问到 ArrayBlockingQueue。

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

LinkedBlockingQueue 底层基于单向链表实现的阻塞队列，可以当做无界队列也可以当做有界队列来使用，同样满足 FIFO 的特性，与 ArrayBlockingQueue 相比起来具有更高的吞吐量，为了防止 LinkedBlockingQueue 容量迅速增，损耗大量内存。通常在创建 LinkedBlockingQueue 对象时，会指定其大小，如果未指定，容量等于 Integer.MAX\_VALUE。

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

riorityBlockingQueue\*\* 是一个支持优先级的无界阻塞队列。默认情况下元素采用自然顺序进行排序，也可以通过自定义类实现 `compareTo()` 方法来指定元素排序规则，或者初始化时通过构造器参数 `Comparator` 来指定排序规则。

PriorityBlockingQueue 并发控制采用的是 \*\*ReentrantLock\*\*，队列为无界队列（ArrayBlockingQueue 是有界队列，LinkedBlockingQueue 也可以通过在构造函数中传入 capacity 指定队列最大的容量，但是 PriorityBlockingQueue 只能指定初始的队列大小，后面插入元素的时候，\*\*如果空间不够的话会自动扩容\*\*）。

简单地说，它就是 PriorityQueue 的线程安全版本。不可以插入 null 值，同时，插入队列的对象必须是可比较大小的（comparable），否则报 ClassCastException 异常。它的插入操作 put 方法不会 block，因为它是无界队列（take 方法在队列为空的时候会阻塞）。

4、DelayQueue ：一个由优先级堆支持的、基于时间的调度队列。

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

### 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

代理模式是一种比较好的理解的设计模式。简单来说就是 我们使用代理对象来代替对真实对象(real object)的访问，这样就可以在不修改原目标对象的前提下，提供额外的功能操作，扩展目标对象的功能。

代理模式的主要作用是扩展目标对象的功能，比如说在目标对象的某个方法执行前后你可以增加一些自定义的操作。

### - 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.

相比于静态代理来说，动态代理更加灵活。我们不需要针对每个目标类都单独创建一个代理类，并且也不需要我们必须实现接口，我们可以直接代理实现类( CGLIB 动态代理机制)。

从 JVM 角度来说，动态代理是在运行时动态生成类字节码，并加载到 JVM 中的。

动态代理在我们日常开发中使用的相对较小，但是在框架中的几乎是必用的一门技术。学会了动态代理之后，对于我们理解和学习各种框架的原理也非常有帮助。

无论是动态代理还是静态代理，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。

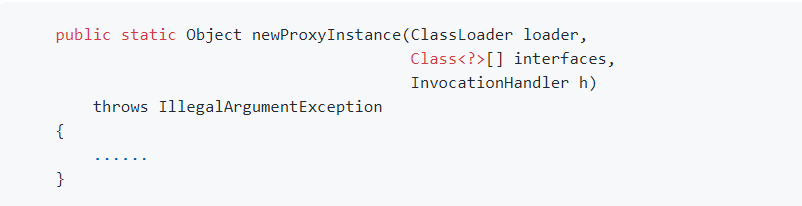
### 静态代理和动态代理的对比

灵活性 ：动态代理更加灵活，不需要必须实现接口，可以直接代理实现类，并且可以不需要针对每个目标类都创建一个代理类。另外，静态代理中，接口一旦新增加方法，目标对象和代理对象都要进行修改，这是非常麻烦的！

JVM 层面 ：静态代理在编译时就将接口、实现类、代理类这些都变成了一个个实际的 class 文件。而动态代理是在运行时动态生成类字节码，并加载到 JVM 中的。

### JDK 动态代理机制

在 Java 动态代理机制中 InvocationHandler 接口和 Proxy 类是核心,Proxy 类中使用频率最高的方法是：newProxyInstance() ，这个方法主要用来生成一个代理对象。



这个方法一共有 3 个参数：

loader :类加载器，用于加载代理对象。

interfaces : 被代理类实现的一些接口；

h : 实现了 InvocationHandler 接口的对象；



要实现动态代理的话，还必须需要实现InvocationHandler 来自定义处理逻辑。 当我们的动态代理对象调用一个方法时候，这个方法的调用就会被转发到实现InvocationHandler 接口类的 invoke 方法来调用。

invoke() 方法有下面三个参数：

proxy :动态生成的代理类

method : 与代理类对象调用的方法相对应

args : 当前 method 方法的参数

也就是说：你通过Proxy 类的 newProxyInstance() 创建的代理对象在调用方法的时候，实际会调用到实现InvocationHandler 接口的类的 invoke()方法。 你可以在 invoke() 方法中自定义处理逻辑，比如在方法执行前后做什么事情。

JDK 动态代理类使用步骤

定义一个接口及其实现类；

自定义 InvocationHandler 并重写invoke方法，在 invoke 方法中我们会调用原生方法（被代理类的方法）并自定义一些处理逻辑；

通过 Proxy.newProxyInstance(ClassLoader loader,Class<?>[] interfaces,InvocationHandler h) 方法创建代理对象；

### cglib代理

JDK 动态代理有一个最致命的问题是其只能代理实现了接口的类。

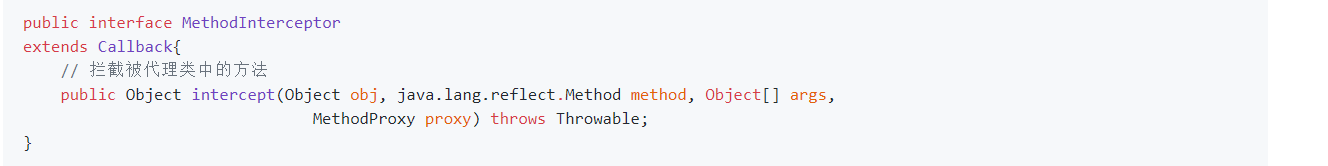
为了解决这个问题，我们可以用 CGLIB 动态代理机制来避免。

CGLIB(Code Generation Library)是一个基于ASM的字节码生成库，它允许我们在运行时对字节码进行修改和动态生成。CGLIB 通过继承方式实现代理。很多知名的开源框架都使用到了CGLIB， 例如 Spring 中的 AOP 模块中：如果目标对象实现了接口，则默认采用 JDK 动态代理，否则采用 CGLIB 动态代理。

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。

在 CGLIB 动态代理机制中 MethodInterceptor 接口和 Enhancer 类是核心。

你需要自定义 MethodInterceptor 并重写 intercept 方法，intercept 用于拦截增强被代理类的方法。



obj :被代理的对象（需要增强的对象）

method :被拦截的方法（需要增强的方法）

args :方法入参

methodProxy :用于调用原始方法

你可以通过 Enhancer类来动态获取被代理类，当代理类调用方法的时候，实际调用的是 MethodInterceptor 中的 intercept 方法。

CGLIB 动态代理类使用步骤

1、定义一个类；

2、自定义 MethodInterceptor 并重写 intercept 方法，intercept 用于拦截增强被代理类的方法，和 JDK 动态代理中的 invoke 方法类似；

3、通过 Enhancer 类的 create()创建代理类；

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

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。

### JDK 动态代理和 CGLIB 动态代理对比

JDK 动态代理只能只能代理实现了接口的类，而 CGLIB 可以代理未实现任何接口的类。 另外， CGLIB 动态代理是通过生成一个被代理类的子类来拦截被代理类的方法调用，因此不能代理声明为 final 类型的类和方法。

就二者的效率来说，大部分情况都是 JDK 动态代理更优秀，随着 JDK 版本的升级，这个优势更加明显。

## 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。

### 具体实现

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。

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：

1. 为 uniqueInstance 分配内存空间

2. 初始化 uniqueInstance

3. 将 uniqueInstance 指向分配的内存地址

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1->3->2。指令重排在单线程环境下不会出现问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 getUniqueInstance() 后发现 uniqueInstance 不为空，因此返回 uniqueInstance，但此时 uniqueInstance 还未被初始化。

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。

## 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）

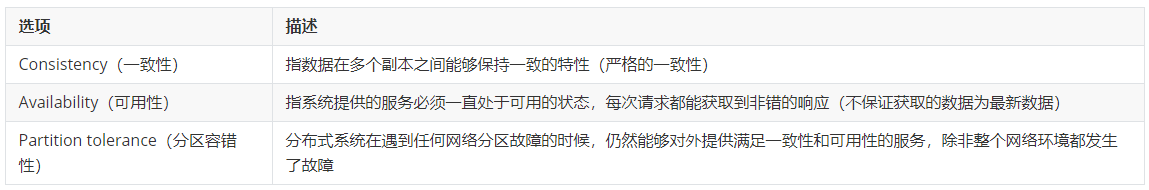
- 去中心化设计的问题： 去中心化设计里最难解决的一个问题是 “脑裂”问题 ，这种情况的发生概率很低，但影响很大。脑裂指一个集群由于网络的故障，被分为至少两个彼此无法通信的单独集群，此时如果两个集群都各自工作，则可能会产生严重的数据冲突和错误。一般的设计思路是，当集群判断发生了脑裂问题时，规模较小的集群就“自杀”或者拒绝服务。

### 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

## 分布式锁

1、在分布式系统环境下，一个方法在同一时间只能被一个机器的一个线程执行

2、高可用的获取锁与释放锁

3、高性能的获取锁与释放锁

4、具备可重入特性（可理解为重新进入，由多于一个任务并发使用，而不必担心数据错误）

5、具备锁失效机制，防止死锁

6、具备非阻塞锁特性，即没有获取到锁将直接返回获取锁失败

### 基于zookeeper的分布式锁

1.zookeeper的一些特性

有序节点：假如当前有一个父节点为/lock，我们可以在这个父节点下面创建子节点；zookeeper提供了一个可选的有序特性，例如我们可以创建子节点“/lock/node-”并且指明有序，那么zookeeper在生成子节点时会根据当前的子节点数量自动添加整数序号，也就是说如果是第一个创建的子节点，那么生成的子节点为/lock/node-0000000000，下一个节点则为/lock/node-0000000001，依次类推。

临时节点：客户端可以建立一个临时节点，在会话结束或者会话超时后，zookeeper会自动删除该节点。

事件监听：在读取数据时，我们可以同时对节点设置事件监听，当节点数据或结构变化时，zookeeper会通知客户端。当前zookeeper有如下四种事件：

1.节点创建；

2.节点删除；

3.节点数据修改；

4.子节点变更。

2.实现

①客户端连接zookeeper，并在/lock下创建临时的且有序的子节点，第一个客户端对应的子节点为/lock/lock-1，第二个为/lock/lock-2，以此类推。

②客户端获取/lock下的子节点列表，判断自己创建的子节点是否为当前子节点列表中序号最小的子节点，如果是则认为获得锁，否则监听/lock的子节点变更消息，获得子节点变更通知后重复此步骤直至获得锁；

③执行业务代码；

④完成业务流程后，删除对应的子节点释放锁。

### 基于redis的分布式锁

1.流程

1、服务A为了获得锁，向Redis发起如下命令: SET productId:lock 0xx9p03001 NX PX 30000 其中，"productId"由自己定义，可以是与本次业务有关的id，"0xx9p03001"是一串随机值，必须保证全局唯一(原因在后文中会提到)，“NX"指的是当且仅当key(也就是案例中的"productId:lock”)在Redis中不存在时，返回执行成功，否则执行失败。"PX 30000"指的是在30秒后，key将被自动删除。执行命令后返回成功，表明服务成功的获得了锁。

2、服务B为了获得锁，向Redis发起同样的命令: SET productId:lock 0000111 NX PX 30000 由于Redis内已经存在同名key，且并未过期，因此命令执行失败，服务B未能获得锁。服务B进入循环请求状态，比如每隔1秒钟(自行设置)向Redis发送请求，直到执行成功并获得锁。

3、服务A的业务代码执行时长超过了30秒，导致key超时，因此Redis自动删除了key。此时服务B再次发送命令执行成功，假设本次请求中设置的value值为0000222。

4、服务A执行完毕，为了释放锁，服务A会主动向Redis发起删除key的请求。

2.实现

①加锁

客户端集成Redisson，在加锁之前，首先需要通过hash算法选定集群内某一个Redis Master，后续加锁、解锁等各种过程都是在这个Redis Master和与之绑定的slave节点之间。

②执行lua脚本实现加锁

③watch dog自动延期

watch dog是一个后台线程，它会每隔10秒观察当前客户端是否仍然持有锁，如果持有，说明客户端可能仍然在使用锁，因此延长锁的剩余生存时间。

④释放锁机制 如果执行lock.unlock()，Redis会找到上方test数据结构，将加锁次数减一。如果减完后发现加锁次数为0，则说明当前客户端不再持有锁，因此执行: del test命令， 从Redis中删除这条key。

## 一致性hash

一致性Hash的除数是2^32^。从0到2^32^ - 1,首尾相连构成了一个环。我们先对服务器节点的IP进行Hash，然后除以2^32^得到服务器节点在这个Hash环中的位置：

### 一致性Hash的应用场景

除了前面提到的负载均衡、Web session和数据库分库分表可以应用外，其实一致性Hash应用得最多的领域是分布式缓存。

分布式缓存系统一般是对Key进行Hash的。试想一下，如果直接对节点数量取模，一旦节点数量发生变化，比如新增一个节点或者删除一个节点，那将使得几乎所有结点的缓存失效。而如果使用一致性Hash，就只有Hash环上相邻的节点缓存受到影响。

从Hash环和顺时针查找原理不难发现，在增加一个节点后，一些原来被分配到下一个节点的请求，被分配到了新的节点。在减少一个节点后，一些原来本分配到这个节点的请求，被分配到了下一个节点。

所以无论是增加还是减少一个节点，都只有下一个相邻的节点会被影响而已。

## 布隆过滤器

布隆过滤器（Bloom Filter）是一个数据结构，由布隆（Burton Howard Bloom）于1970年提出的。它实际上是一个很长的二进制向量和一系列随机映射函数。

布隆过滤器可以用于高效的检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远优于一般的算法，缺点是有一定的误识别率，而且难以删除（一般不支持，需要额外的实现）。

布隆过滤器之所以高效，因为它是一个概率数据结构，它能确认元素肯定不在集合中，或者元素可能在集合中。之所以说是可能，是因为它有一定的误识别率，使得无法100%确定元素一定在集合中。

### 基本原理

布隆过滤器的基本工作原理并不复杂，大致如下：

首先，建立一个二进制向量，并将所有位设置为0。

然后，选定K个散列函数，用于对元素进行K次散列，计算向量的位下标。

添加元素

当添加一个元素到集合中时，通过K个散列函数分别作用于元素，生成K个值作为下标，并对向量的相应位设置为1。

检查元素

如果要检查一个元素是否存在集合中，用同样的散列方法，生成K个下标，并检查向量的相应位是否全部是1。如果全为1，则该元素很可能在集合中；否则（只要有1个或以上的位为0），该元素肯定不在集合中。

这就是布隆过滤器的基本思想。

数学原理

布隆过滤器背后的数学原理是：

两个完全随机的数字相冲突的概率很小，因此可以在很小的误识别率条件下，用很少的空间存储大量信息。

### 解决误识别率的2种方法

白名单

解决误识别率的常见方法，是建立一个较小的白名单，用来存储那些可能被误识别的数据。

以垃圾邮件过滤为例。假设我们有一个垃圾邮件库，用于在接收邮件的时候过滤掉垃圾邮件。

这时可以先将这个垃圾邮件库存储到布隆过滤器中，当接收到邮件的时候，可以先通过布隆过滤器高效的过滤出大部分正常邮件。

而对于少部分命中（可能为）垃圾邮件的，其中有一部分可能为正常邮件。

再创建一个白名单库，当在布隆过滤器中判断可能为垃圾邮件时，通过查询白名单来确认是否为正常邮件。

对于没在白名单中的邮件，默认会被移动到垃圾箱。通过人工识别的方式，当发现垃圾箱中存在正常邮件的时候，将其移入白名单。

### 回源确认

很多时候，使用布隆过滤器是为了低成本，高效率的拦截掉大量数据不在集合中的场景。

例如：

Google Bigtable，Apache HBase以及Apache Cassandra和PostgreSQL 使用Bloom过滤器来减少对不存在的行或列的磁盘查找。避免进行昂贵的磁盘查找，可大大提高数据库查询操作的性能。

在谷歌浏览器用于使用布隆过滤器来识别恶意URL的网页浏览器。首先会针对本地Bloom过滤器检查所有URL，只有在Bloom过滤器返回肯定结果的情况下，才对执行的URL进行全面检查（如果该结果也返回肯定结果，则用户会发出警告）。

拦截掉大量非IP黑名单请求，对于少量可能在黑名单中的IP，再查询一次黑名单库。

这是布隆过滤器非常典型的应用场景，先过滤掉大部分请求，然后只处理少量不明确的请求。

这个方法，和白名单库的区别是，不需要再另外建立一套库来处理，而是使用本来就已经存在的数据和逻辑。

例如Google Bigtable查询数据行本来就是需要查的，只不过使用布隆过滤器拦截掉了大部分不必要的请求。而IP是否为黑名单也是需要查询的，同样是先使用布隆过滤器来拦截掉大部分IP。

而上面垃圾邮件的处理，对于可能为垃圾邮件的情况，不是通过完整的垃圾邮件库再查询一次进行确认，而是用增加白名单来进行判断的方式。因为通常来说，白名单库会更小，便于缓存。

这里所说的回源，实际上是对可能被误识别的请求，最后要回到数据源头或逻辑确认一次。

## RPC

Remote Procedure Call Protocol，远程过程调用

RPC框架，要向调用方屏蔽各种复杂性，要向服务提供方也屏蔽各类复杂性：

服务调用方client感觉就像调用本地函数一样，来调用服务

服务提供方server感觉就像实现一个本地函数一样，来实现服务

所以整个RPC框架又分为client部分与server部分，实现上面的目标，把复杂性屏蔽，就是RPC框架的职责。

### RPC框架的职责是，

client端：序列化、反序列化、连接池管理、负载均衡、故障转移、队列管理，超时管理、异步管理等等

server端：服务端组件、服务端收发包队列、io线程、工作线程、序列化反序列化等

### 有哪些常见的序列化方式？

xml/json：解析效率，压缩率都较差，扩展性、可读性、通用性较好

## 生产者-消费者模型

网上有很多生产者-消费者模型的定义和实现。本文研究最常用的有界生产者-消费者模型，简单概括如下：

生产者持续生产，直到缓冲区满，阻塞；缓冲区不满后，继续生产

消费者持续消费，直到缓冲区空，阻塞；缓冲区不空后，继续消费

生产者可以有多个，消费者也可以有多个

可通过如下条件验证模型实现的正确性：

同一产品的消费行为一定发生在生产行为之后

任意时刻，缓冲区大小不小于0，不大于限制容量

该模型的应用和变种非常多，不赘述。

### 实现一：BlockingQueue

BlockingQueue的写法最简单。核心思想是，把并发和容量控制封装在缓冲区中。而BlockingQueue的性质天生满足这个要求。

public class BlockingQueueModel implements Model {

private final BlockingQueue<Task> queue;

private final AtomicInteger increTaskNo = new AtomicInteger(0);

public BlockingQueueModel(int cap) {

// LinkedBlockingQueue 的队列是 lazy-init 的，但 ArrayBlockingQueue 在创建时就已经 init

this.queue = new LinkedBlockingQueue<>(cap);

}

@Override

public Runnable newRunnableConsumer() {

return new ConsumerImpl();

}

@Override

public Runnable newRunnableProducer() {

return new ProducerImpl();

}

private class ConsumerImpl extends AbstractConsumer implements Consumer, Runnable {

@Override

public void consume() throws InterruptedException {

Task task = queue.take();

// 固定时间范围的消费，模拟相对稳定的服务器处理过程

Thread.sleep(500 + (long) (Math.random() \* 500));

System.out.println("consume: " + task.no);

}

}

private class ProducerImpl extends AbstractProducer implements Producer, Runnable {

@Override

public void produce() throws InterruptedException {

// 不定期生产，模拟随机的用户请求

Thread.sleep((long) (Math.random() \* 1000));

Task task = new Task(increTaskNo.getAndIncrement());

System.out.println("produce: " + task.no);

queue.put(task);

}

}

public static void main(String[] args) {

Model model = new BlockingQueueModel(3);

for (int i = 0; i < 2; i++) {

new Thread(model.newRunnableConsumer()).start();

}

for (int i = 0; i < 5; i++) {

new Thread(model.newRunnableProducer()).start();

}

}

}

由于操作“出队/入队+日志输出”不是原子的，所以上述日志的绝对顺序与实际的出队/入队顺序有出入，但对于同一个任务号task.no，其consume日志一定出现在其produce日志之后，即：同一任务的消费行为一定发生在生产行为之后。缓冲区的容量留给读者验证。符合两个验证条件。

BlockingQueue写法的核心只有两行代码，并发和容量控制都封装在了BlockingQueue中，正确性由BlockingQueue保证。面试中首选该写法，自然美观简单。

### 实现二：wait && notify

如果不能将并发与容量控制都封装在缓冲区中，就只能由消费者与生产者完成。最简单的方案是使用朴素的wait && notify机制。

public class WaitNotifyModel implements Model {

private final Object BUFFER\_LOCK = new Object();

private final Queue<Task> buffer = new LinkedList<>();

private final int cap;

private final AtomicInteger increTaskNo = new AtomicInteger(0);

public WaitNotifyModel(int cap) {

this.cap = cap;

}

@Override

public Runnable newRunnableConsumer() {

return new ConsumerImpl();

}

@Override

public Runnable newRunnableProducer() {

return new ProducerImpl();

}

private class ConsumerImpl extends AbstractConsumer implements Consumer, Runnable {

@Override

public void consume() throws InterruptedException {

synchronized (BUFFER\_LOCK) {

while (buffer.size() == 0) {

BUFFER\_LOCK.wait();

}

Task task = buffer.poll();

assert task != null;

// 固定时间范围的消费，模拟相对稳定的服务器处理过程

Thread.sleep(500 + (long) (Math.random() \* 500));

System.out.println("consume: " + task.no);

BUFFER\_LOCK.notifyAll();

}

}

}

private class ProducerImpl extends AbstractProducer implements Producer, Runnable {

@Override

public void produce() throws InterruptedException {

// 不定期生产，模拟随机的用户请求

Thread.sleep((long) (Math.random() \* 1000));

synchronized (BUFFER\_LOCK) {

while (buffer.size() == cap) {

BUFFER\_LOCK.wait();

}

Task task = new Task(increTaskNo.getAndIncrement());

buffer.offer(task);

System.out.println("produce: " + task.no);

BUFFER\_LOCK.notifyAll();

}

}

}

public static void main(String[] args) {

Model model = new WaitNotifyModel(3);

for (int i = 0; i < 2; i++) {

new Thread(model.newRunnableConsumer()).start();

}

for (int i = 0; i < 5; i++) {

new Thread(model.newRunnableProducer()).start();

}

}

}

### 实现三：简单的Lock && Condition

我们要保证理解wait && notify机制。实现时可以使用Object类提供的wait()方法与notifyAll()方法，但更推荐的方式是使用java.util.concurrent包提供的Lock && Condition。

public class LockConditionModel1 implements Model {

private final Lock BUFFER\_LOCK = new ReentrantLock();

private final Condition BUFFER\_COND = BUFFER\_LOCK.newCondition();

private final Queue<Task> buffer = new LinkedList<>();

private final int cap;

private final AtomicInteger increTaskNo = new AtomicInteger(0);

public LockConditionModel1(int cap) {

this.cap = cap;

}

@Override

public Runnable newRunnableConsumer() {

return new ConsumerImpl();

}

@Override

public Runnable newRunnableProducer() {

return new ProducerImpl();

}

private class ConsumerImpl extends AbstractConsumer implements Consumer, Runnable {

@Override

public void consume() throws InterruptedException {

BUFFER\_LOCK.lockInterruptibly();

try {

while (buffer.size() == 0) {

BUFFER\_COND.await();

}

Task task = buffer.poll();

assert task != null;

// 固定时间范围的消费，模拟相对稳定的服务器处理过程

Thread.sleep(500 + (long) (Math.random() \* 500));

System.out.println("consume: " + task.no);

BUFFER\_COND.signalAll();

} finally {

BUFFER\_LOCK.unlock();

}

}

}

private class ProducerImpl extends AbstractProducer implements Producer, Runnable {

@Override

public void produce() throws InterruptedException {

// 不定期生产，模拟随机的用户请求

Thread.sleep((long) (Math.random() \* 1000));

BUFFER\_LOCK.lockInterruptibly();

try {

while (buffer.size() == cap) {

BUFFER\_COND.await();

}

Task task = new Task(increTaskNo.getAndIncrement());

buffer.offer(task);

System.out.println("produce: " + task.no);

BUFFER\_COND.signalAll();

} finally {

BUFFER\_LOCK.unlock();

}

}

}

public static void main(String[] args) {

Model model = new LockConditionModel1(3);

for (int i = 0; i < 2; i++) {

new Thread(model.newRunnableConsumer()).start();

}

for (int i = 0; i < 5; i++) {

new Thread(model.newRunnableProducer()).start();

}

}

}

复制代码该写法的思路与实现二的思路完全相同，仅仅将锁与条件变量换成了Lock和Condition。

### 实现四：更高并发性能的Lock && Condition

现在，如果做一些实验，你会发现，实现一的并发性能高于实现二、三。暂且不关心BlockingQueue的具体实现，来分析看如何优化实现三（与实现二的思路相同，性能相当）的性能。

分析实现三的瓶颈

最好的查证方法是记录方法执行时间，这样可以直接定位到真正的瓶颈。但此问题较简单，我们直接用“瞪眼法”分析。

实现三的并发瓶颈很明显，因为在锁 BUFFER\_LOCK 看来，任何消费者线程与生产者线程都是一样的。换句话说，同一时刻，最多只允许有一个线程（生产者或消费者，二选一）操作缓冲区 buffer。

而实际上，如果缓冲区是一个队列的话，“生产者将产品入队”与“消费者将产品出队”两个操作之间没有同步关系，可以在队首出队的同时，在队尾入队。理想性能可提升至实现三的两倍。

去掉这个瓶颈

那么思路就简单了：需要两个锁 CONSUME\_LOCK与PRODUCE\_LOCK，CONSUME\_LOCK控制消费者线程并发出队，PRODUCE\_LOCK控制生产者线程并发入队；相应需要两个条件变量NOT\_EMPTY与NOT\_FULL，NOT\_EMPTY负责控制消费者线程的状态（阻塞、运行），NOT\_FULL负责控制生产者线程的状态（阻塞、运行）。以此让优化消费者与消费者（或生产者与生产者）之间是串行的；消费者与生产者之间是并行的。

public class LockConditionModel2 implements Model {

private final Lock CONSUME\_LOCK = new ReentrantLock();

private final Condition NOT\_EMPTY = CONSUME\_LOCK.newCondition();

private final Lock PRODUCE\_LOCK = new ReentrantLock();

private final Condition NOT\_FULL = PRODUCE\_LOCK.newCondition();

private final Buffer<Task> buffer = new Buffer<>();

private AtomicInteger bufLen = new AtomicInteger(0);

private final int cap;

private final AtomicInteger increTaskNo = new AtomicInteger(0);

public LockConditionModel2(int cap) {

this.cap = cap;

}

@Override

public Runnable newRunnableConsumer() {

return new ConsumerImpl();

}

@Override

public Runnable newRunnableProducer() {

return new ProducerImpl();

}

private class ConsumerImpl extends AbstractConsumer implements Consumer, Runnable {

@Override

public void consume() throws InterruptedException {

int newBufSize = -1;

CONSUME\_LOCK.lockInterruptibly();

try {

while (bufLen.get() == 0) {

System.out.println("buffer is empty...");

NOT\_EMPTY.await();

}

Task task = buffer.poll();

newBufSize = bufLen.decrementAndGet();

assert task != null;

// 固定时间范围的消费，模拟相对稳定的服务器处理过程

Thread.sleep(500 + (long) (Math.random() \* 500));

System.out.println("consume: " + task.no);

if (newBufSize > 0) {

NOT\_EMPTY.signalAll();

}

} finally {

CONSUME\_LOCK.unlock();

}

if (newBufSize < cap) {

PRODUCE\_LOCK.lockInterruptibly();

try {

NOT\_FULL.signalAll();

} finally {

PRODUCE\_LOCK.unlock();

}

}

}

}

private class ProducerImpl extends AbstractProducer implements Producer, Runnable {

@Override

public void produce() throws InterruptedException {

// 不定期生产，模拟随机的用户请求

Thread.sleep((long) (Math.random() \* 1000));

int newBufSize = -1;

PRODUCE\_LOCK.lockInterruptibly();

try {

while (bufLen.get() == cap) {

System.out.println("buffer is full...");

NOT\_FULL.await();

}

Task task = new Task(increTaskNo.getAndIncrement());

buffer.offer(task);

newBufSize = bufLen.incrementAndGet();

System.out.println("produce: " + task.no);

if (newBufSize < cap) {

NOT\_FULL.signalAll();

}

} finally {

PRODUCE\_LOCK.unlock();

}

if (newBufSize > 0) {

CONSUME\_LOCK.lockInterruptibly();

try {

NOT\_EMPTY.signalAll();

} finally {

CONSUME\_LOCK.unlock();

}

}

}

}

private static class Buffer<E> {

private Node head;

private Node tail;

Buffer() {

// dummy node

head = tail = new Node(null);

}

public void offer(E e) {

tail.next = new Node(e);

tail = tail.next;

}

public E poll() {

head = head.next;

E e = head.item;

head.item = null;

return e;

}

private class Node {

E item;

Node next;

Node(E item) {

this.item = item;

}

}

}

public static void main(String[] args) {

Model model = new LockConditionModel2(3);

for (int i = 0; i < 2; i++) {

new Thread(model.newRunnableConsumer()).start();

}

for (int i = 0; i < 5; i++) {

new Thread(model.newRunnableProducer()).start();

}

}

复制代码需要注意的是，由于需要同时在UnThreadSafe的缓冲区 buffer 上进行消费与生产，我们不能使用实现二、三中使用的队列了，需要自己实现一个简单的缓冲区 Buffer。Buffer要满足以下条件：

在头部出队，尾部入队

在poll()方法中只操作head

在offer()方法中只操作tail

还能进一步优化吗

我们已经优化掉了消费者与生产者之间的瓶颈，还能进一步优化吗？

如果可以，必然是继续优化消费者与消费者（或生产者与生产者）之间的并发性能。然而，消费者与消费者之间必须是串行的，因此，并发模型上已经没有地方可以继续优化了。

不过在具体的业务场景中，一般还能够继续优化。如：

并发规模中等，可考虑使用CAS代替重入锁

模型上不能优化，但一个消费行为或许可以进一步拆解、优化，从而降低消费的延迟

一个队列的并发性能达到了极限，可采用“多个队列”（如分布式消息队列等）

# 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

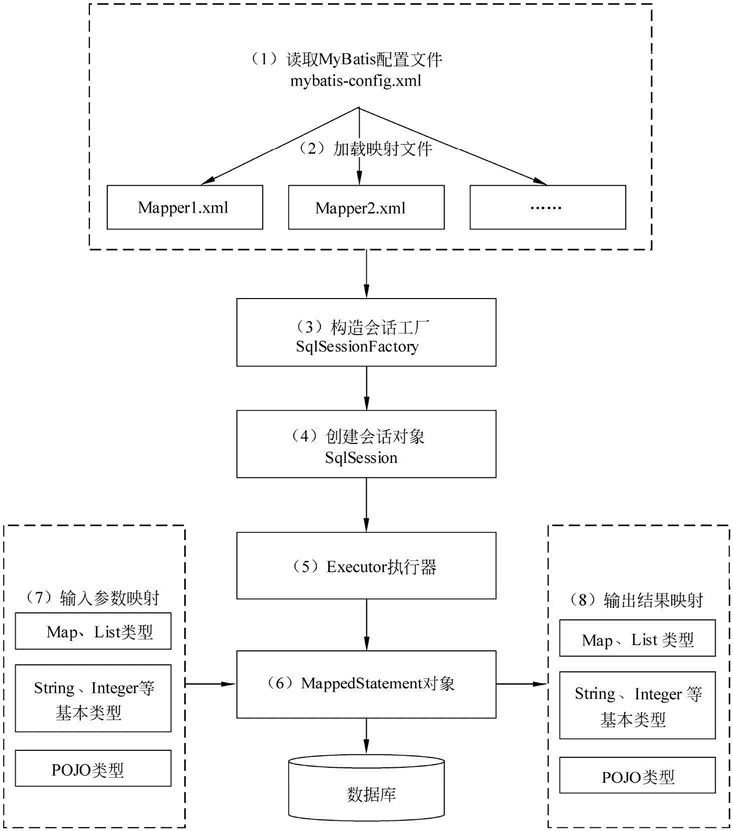
而MyBatis也有四大核心对象：

（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。

### 原理

通过@EnableAspectJAutoProxy；注解，【看给容器中注册了什么组件，这个组件什么时候工作，这个组件的功能是什么？】

1、@EnableAspectJAutoProxy作用

@Import(AspectJAutoProxyRegistrar.class)：

给容器中导入AspectJAutoProxyRegistrar

利用AspectJAutoProxyRegistrar自定义给容器中注册bean；BeanDefinetion

internalAutoProxyCreator=AnnotationAwareAspectJAutoProxyCreator

给容器中注册一个AnnotationAwareAspectJAutoProxyCreator；

2、 AnnotationAwareAspectJAutoProxyCreator继承关系：

AnnotationAwareAspectJAutoProxyCreator（注解感知AspectJ自动代理创建器）

->AspectJAwareAdvisorAutoProxyCreator

->AbstractAdvisorAutoProxyCreator

->AbstractAutoProxyCreator

实现了两个接口，

implements SmartInstantiationAwareBeanPostProcessor（智能实例化感知bean后置处理器）, BeanFactoryAware(Bean工厂感知)

一个是后置处理器在bean初始化完成前后做事情，

一个是bean工厂的Aware ，自动装配BeanFactory

AbstractAutoProxyCreator.setBeanFactory()

AbstractAutoProxyCreator.有后置处理器的逻辑；

AbstractAdvisorAutoProxyCreator.setBeanFactory()-》initBeanFactory()

AnnotationAwareAspectJAutoProxyCreator.initBeanFactory()

#### 流程：

一、传入配置类，创建ioc容器

二、注册配置类，调用refresh（）刷新容器；

三、registerBeanPostProcessors(beanFactory);

注册bean的后置处理器来方便拦截bean的创建；

1）、先获取ioc容器已经定义了的需要创建对象的所有BeanPostProcessor

2）、给容器中加别的BeanPostProcessor

3）、优先注册实现了PriorityOrdered接口的BeanPostProcessor；

4）、再给容器中注册实现了Ordered接口的BeanPostProcessor；

5）、注册没实现优先级接口的BeanPostProcessor；

6）、注册BeanPostProcessor，实际上就是创建BeanPostProcessor对象，保存在容器中；

创建internalAutoProxyCreator的BeanPostProcessor【AnnotationAwareAspectJAutoProxyCreator】aspectJAdvisorsBuilder

7）、把BeanPostProcessor注册到BeanFactory中；

beanFactory.addBeanPostProcessor(postProcessor);

=======以上是创建和注册AnnotationAwareAspectJAutoProxyCreator的过程========

AnnotationAwareAspectJAutoProxyCreator => InstantiationAwareBeanPostProcessor

四、finishBeanFactoryInitialization(beanFactory);

完成BeanFactory初始化工作；创建剩下的单实例bean

1）、遍历获取容器中所有的Bean，依次创建对象getBean(beanName);

getBean->doGetBean()->getSingleton()->

2）、创建bean

【AnnotationAwareAspectJAutoProxyCreator在所有bean创建之前会有一个拦截，InstantiationAwareBeanPostProcessor，会调用postProcessBeforeInstantiation()】

1）、先从缓存中获取当前bean，如果能获取到，说明bean是之前被创建过的，直接使用，否则再创建；

只要创建好的Bean都会被缓存起来

2）、createBean（）;创建bean；如果需要代理对象则用后置处理器创建对象，

AnnotationAwareAspectJAutoProxyCreator 会在任何bean创建之前先尝试返回bean的实例

【BeanPostProcessor是在Bean对象创建完成初始化前后调用的】

【InstantiationAwareBeanPostProcessor是在创建Bean实例之前先尝试用后置处理器返回对象的】

1）、resolveBeforeInstantiation(beanName, mbdToUse);解析BeforeInstantiation

希望后置处理器在此能返回一个代理对象；如果能返回代理对象就使用，如果不能就继续

1）、后置处理器先尝试返回对象； 拿到所有后置处理器，如果是InstantiationAwareBeanPostProcessor;

就执行postProcessBeforeInstantiation

bean = applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);//执行实例化之前的后置处理方法

if (bean != null) {

bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);//执行初始化之后的后置处理方法

}

2）、doCreateBean(beanName, mbdToUse, args);真正的去创建一个bean实例；不使用代理方式创建对象。

#### AnnotationAwareAspectJAutoProxyCreator【InstantiationAwareBeanPostProcessor】 的作用：

1）、每一个bean创建之前，调用postProcessBeforeInstantiation()；

关心MathCalculator和LogAspect的创建

1）、判断当前bean是否在advisedBeans中（保存了所有需要增强bean）

2）、判断当前bean是否是基础类型的Advice、Pointcut、Advisor、AopInfrastructureBean，

或者是否是切面（@Aspect）

3）、是否需要跳过

1）、获取候选的增强器（切面里面的通知方法）【List<Advisor> candidateAdvisors】

每一个封装的通知方法的增强器是 InstantiationModelAwarePointcutAdvisor；

判断每一个增强器是否是 AspectJPointcutAdvisor 类型的；返回true

2）、永远返回false

2）、创建对象

postProcessAfterInitialization；

return wrapIfNecessary(bean, beanName, cacheKey);//包装如果需要的情况下

1）、获取当前bean的所有增强器（通知方法） Object[] specificInterceptors

1、找到候选的所有的增强器（找哪些通知方法是需要切入当前bean方法的）

2、获取到能在bean使用的增强器。

3、给增强器排序

2）、保存当前bean在advisedBeans中；

3）、如果当前bean需要增强，创建当前bean的代理对象；

1）、获取所有增强器（通知方法）

2）、保存到proxyFactory

3）、创建代理对象：Spring自动决定

JdkDynamicAopProxy(config);jdk动态代理；

ObjenesisCglibAopProxy(config);cglib的动态代理；

4）、给容器中返回当前组件使用cglib增强了的代理对象；

5）、以后容器中获取到的就是这个组件的代理对象，执行目标方法的时候，代理对象就会执行通知方法的流程；

3）、目标方法执行 ；

容器中保存了组件的代理对象（cglib增强后的对象），这个对象里面保存了详细信息（比如增强器，目标对象，xxx）；

1）、CglibAopProxy.intercept();拦截目标方法的执行

2）、根据ProxyFactory对象获取将要执行的目标方法拦截器链；

List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);

1）、List<Object> interceptorList保存所有拦截器 5

一个默认的ExposeInvocationInterceptor 和 4个增强器；

2）、遍历所有的增强器，将其转为Interceptor；

registry.getInterceptors(advisor);

3）、将增强器转为List<MethodInterceptor>；

如果是MethodInterceptor，直接加入到集合中

如果不是，使用AdvisorAdapter将增强器转为MethodInterceptor；

转换完成返回MethodInterceptor数组；

3）、如果没有拦截器链，直接执行目标方法;

拦截器链（每一个通知方法又被包装为方法拦截器，利用MethodInterceptor机制）

4）、如果有拦截器链，把需要执行的目标对象，目标方法，

拦截器链等信息传入创建一个 CglibMethodInvocation 对象，

并调用 Object retVal = mi.proceed();

5）、拦截器链的触发过程;

1)、如果没有拦截器执行执行目标方法，或者拦截器的索引和拦截器数组-1大小一样（指定到了最后一个拦截器）执行目标方法；

2)、链式获取每一个拦截器，拦截器执行invoke方法，每一个拦截器等待下一个拦截器执行完成返回以后再来执行；

拦截器链的机制，保证通知方法与目标方法的执行顺序；

#### 总结

1）、 @EnableAspectJAutoProxy 开启AOP功能

2）、 @EnableAspectJAutoProxy 会给容器中注册一个组件 AnnotationAwareAspectJAutoProxyCreator

3）、AnnotationAwareAspectJAutoProxyCreator是一个后置处理器；

4）、容器的创建流程：

1）、registerBeanPostProcessors（）注册后置处理器；创建AnnotationAwareAspectJAutoProxyCreator对象

2）、finishBeanFactoryInitialization（）初始化剩下的单实例bean

1）、创建业务逻辑组件和切面组件

2）、AnnotationAwareAspectJAutoProxyCreator拦截组件的创建过程

3）、组件创建完之后，判断组件是否需要增强

是：切面的通知方法，包装成增强器（Advisor）;给业务逻辑组件创建一个代理对象（cglib）；

5）、执行目标方法：

1）、代理对象执行目标方法

2）、CglibAopProxy.intercept()；

1）、得到目标方法的拦截器链（增强器包装成拦截器MethodInterceptor）

2）、利用拦截器的链式机制，依次进入每一个拦截器进行执行；

3）、效果：

正常执行：前置通知-》目标方法-》后置通知-》返回通知

出现异常：前置通知-》目标方法-》后置通知-》异常通知

## 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。

## Spring解决循环依赖

构造器的循环依赖。Spring解决不了

Spring先用构造器实例化Bean对象----->将实例化结束的对象放到一个Map中，并且Spring提供获取这个未设置属性的实例化对象的引用方法。结合我们的实例来看，，当Spring实例化了StudentA、StudentB、StudentC后，紧接着会去设置对象的属性，此时StudentA依赖StudentB，就会去Map中取出存在里面的单例StudentB对象，以此类推，不会出来循环的问题喽。

多例的也解决不了

①：createBeanInstance：实例化，其实也就是 调用对象的构造方法实例化对象

②：populateBean：填充属性，这一步主要是多bean的依赖属性进行填充

③：initializeBean：调用springbean中的init() 方法。

singletonFactories ： 单例对象工厂的cache

earlySingletonObjects ：提前暴光的单例对象的Cache 。【用于检测循环引用，与singletonFactories互斥】

singletonObjects：单例对象的cache

Spring首先从一级缓存singletonObjects中获取。如果获取不到，并且对象正在创建中，就再从二级缓存earlySingletonObjects中获取。如果还是获取不到且允许singletonFactories通过getObject()获取，就从三级缓存singletonFactory.getObject()(三级缓存)获取，如果获取到了则从singletonFactories中移除，并放入earlySingletonObjects中。其实也就是从三级缓存移动到了二级缓存。

createBeanInstance之后三级缓存中就有这个对象了。

# 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。

## Servlet总结

在Java Web程序中，Servlet主要负责接收用户请求 HttpServletRequest,在doGet(),doPost()中做相应的处理，并将回应HttpServletResponse反馈给用户。Servlet 可以设置初始化参数，供Servlet内部使用。一个Servlet类只会有一个实例，在它初始化时调用init()方法，销毁时调用destroy()方法\*\*。\*\*Servlet需要在web.xml中配置（MyEclipse中创建Servlet会自动配置），一个Servlet可以设置多个URL访问。Servlet不是线程安全，因此要谨慎使用类变量。

### 生命周期：

Web容器加载Servlet并将其实例化后，Servlet生命周期开始，容器运行其init()方法进行Servlet的初始化；请求到达时调用Servlet的service()方法，service()方法会根据需要调用与请求对应的doGet或doPost等方法；当服务器关闭或项目被卸载时服务器会将Servlet实例销毁，此时会调用Servlet的destroy()方法。init方法和destroy方法只会执行一次，service方法客户端每次请求Servlet都会执行。Servlet中有时会用到一些需要初始化与销毁的资源，因此可以把初始化资源的代码放入init方法中，销毁资源的代码放入destroy方法中，这样就不需要每次处理客户端的请求都要初始化与销毁资源。

### get和post请求的区别

get和post请求实际上是没有区别，

可以把 get 和 post 当作两个不同的行为，两者并没有什么本质区别，底层都是 TCP 连接。 get请求用来从服务器上获得资源，而post是用来向服务器提交数据。比如你要获取人员列表可以用 get 请求，你需要创建一个人员可以用 post 。这也是 Restful API 最基本的一个要求。

### Cookie和Session的的区别

Cookie 和 Session都是用来跟踪浏览器用户身份的会话方式，但是两者的应用场景不太一样。

Cookie 一般用来保存用户信息 比如

①我们在 Cookie 中保存已经登录过得用户信息，下次访问网站的时候页面可以自动帮你登录的一些基本信息给填了；

②一般的网站都会有保持登录也就是说下次你再访问网站的时候就不需要重新登录了，这是因为用户登录的时候我们可以存放了一个 Token 在 Cookie 中，下次登录的时候只需要根据 Token 值来查找用户即可(为了安全考虑，重新登录一般要将 Token 重写)；

③登录一次网站后访问网站其他页面不需要重新登录。Session 的主要作用就是通过服务端记录用户的状态。 典型的场景是购物车，当你要添加商品到购物车的时候，系统不知道是哪个用户操作的，因为 HTTP 协议是无状态的。服务端给特定的用户创建特定的 Session 之后就可以标识这个用户并且跟踪这个用户了。

Cookie 数据保存在客户端(浏览器端)，Session 数据保存在服务器端。

Cookie 存储在客户端中，而Session存储在服务器上，相对来说 Session 安全性更高。如果使用 Cookie 的一些敏感信息不要写入 Cookie 中，最好能将 Cookie 信息加密然后使用到的时候再去服务器端解密。

## 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 )。\*\*

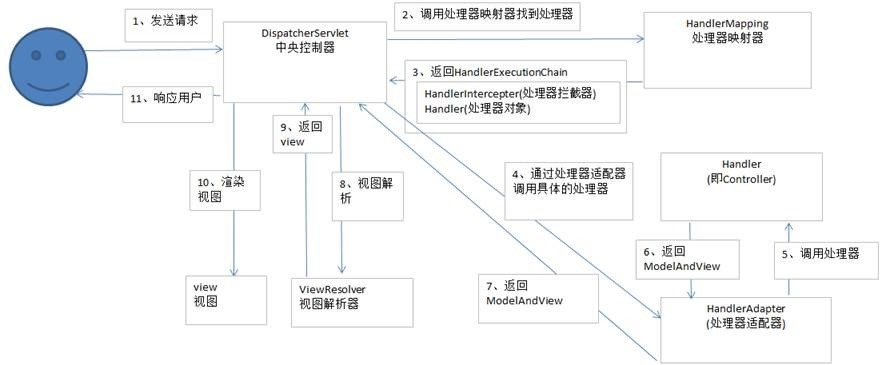
- HandlerAdapter 处理器适配器接口 因为，处理器 `handler` 的类型是 Object 类型，需要有一个调用者来实现 `handler` 是怎么被使用，怎么被执行。而 HandlerAdapter 的用途就在于此

- 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

### 为什么要使用联合索引

减少开销。建一个联合索引 (col1,col2,col3)，实际相当于建了 (col1)，(col1,col2)，(col1,col2,col3) 三个索引。每多一个索引，都会增加写操作的开销和磁盘空间的开销。对于大量数据的表，使用联合索引会大大的减少开销！

覆盖索引。对联合索引 (col1,col2,col3)，如果有如下的 SQL：select col1,col2,col3 from test where col1=1 and col2=2;。那么 MySQL 可以直接通过遍历索引取得数据，而无需回表，这减少了很多的随机 IO 操作。减少 IO 操作，特别的随机 IO 其实是 DBA 主要的优化策略。所以，在真正的实际应用中，覆盖索引是主要的提升性能的优化手段之一。

效率高。索引列越多，通过索引筛选出的数据越少。有 1000W 条数据的表，有如下 SQL：select from table where col1=1 and col2=2 and col3=3，假设假设每个条件可以筛选出 10% 的数据，如果只有单值索引，那么通过该索引能筛选出 1000W10%=100w 条数据，然后再回表从 100w 条数据中找到符合 col2=2 and col3=3 的数据，然后再排序，再分页；如果是联合索引，通过索引筛选出 1000w \* 10% \* 10% \* 10% =1w，效率提升可想而知

### 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

最左前缀原则：即当你创建了一个联合索引，该索引的任何最左前缀都可以用于查询。比如当你有一个联合索引 (col1, col2, col3)，该索引的所有前缀为 (col1)、(col1, col2)、(col1, col2, col3)，包含这些列的所有查询都会使用该索引进行查询。

联合索引最多包含 16 列。

BLOB 和 TEXT 创建索引必须执行前多少位

### 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。

- 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。

### 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

磁盘读写次数更少 : 在B+树里面,由于内节点不存储数据,因此每个内节点中能存放更多的key,在相同的数据量下,内节点数量更少,树的高度更低,磁盘的读写时间开销更低.

遍历更加方便 : 由于B+树的数据都存储在叶子结点中,内结点均为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，所以通常B+树用于数据库索引。

范围查找更加方便 : B+树的叶子节点通过链表与相邻的叶子节点相连,更适合于范围查找,而在数据库中基于范围的查询是非常频繁的.

### 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。

### 那ACID靠什么保证的呢？

A原子性由undo log日志保证，它记录了需要回滚的日志信息，事务回滚时撤销已经执行成功的sql

C一致性一般由代码层面来保证

I隔离性由MVCC来保证

D持久性由内存+redo log来保证，mysql修改数据同时在内存和redo log记录这次操作，事务提交的时候通过redo log刷盘，宕机的时候可以从redo log恢复

### 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

### MySQL的锁

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。

### 意向锁到底有什么作用？

innodb的意向锁主要用户多粒度的锁并存的情况。比如事务A要在一个表上加S锁，如果表中的一行已被事务B加了X锁，那么该锁的申请也应被阻塞。如果表中的数据很多，逐行检查锁标志的开销将很大，系统的性能将会受到影响。为了解决这个问题，可以在表级上引入新的锁类型来表示其所属行的加锁情况，这就引出了“意向锁”的概念。

举个例子，如果表中记录1亿，事务A把其中有几条记录上了行锁了，这时事务B需要给这个表加表级锁，如果没有意向锁的话，那就要去表中查找这一亿条记录是否上锁了。如果存在意向锁，那么假如事务Ａ在更新一条记录之前，先加意向锁，再加Ｘ锁，事务B先检查该表上是否存在意向锁，存在的意向锁是否与自己准备加的锁冲突，如果有冲突，则等待直到事务Ａ释放，而无须逐条记录去检测。事务Ｂ更新表时，其实无须知道到底哪一行被锁了，它只要知道反正有一行被锁了就行了。

主要作用是处理行锁和表锁之间的矛盾，能够显示“某个事务正在某一行上持有了锁，或者准备去持有锁”

### 什么是间隙锁吗？

间隙锁是可重复读级别下才会有的锁，结合MVCC和间隙锁可以解决幻读的问题。我们还是以user举例，假设现在user表有几条记录

当我们执行：

begin;

select \* from user where age=20 for update;

begin;

insert into user(age) values(10); #成功

insert into user(age) values(11); #失败

insert into user(age) values(20); #失败

insert into user(age) values(21); #失败

insert into user(age) values(30); #失败

只有10可以插入成功，那么因为表的间隙mysql自动帮我们生成了区间(左开右闭)

(negative infinity，10],(10,20],(20,30],(30,positive infinity)

由于20存在记录，所以(10,20]，(20,30]区间都被锁定了无法插入、删除。

如果查询21呢？就会根据21定位到(20,30)的区间(都是开区间)。

需要注意的是唯一索引是不会有间隙索引的。

### 读的类型

（1）一致性非锁定读（快照读）

在事务隔离级别提交读（RC）和可重复读（RR）下，InnoDB存储引擎使用非锁定的一致性读

① RC模式下，读取最新的快照

② RR模式下，读取事务开始时的快照

（2）一致性锁定读 （当前读）

隔离级别为未提交读(RN）时读取都是当前读

SELECT…FOR UPDATE （加写锁）

SELECT…LOCK IN SHARE MODE （加读锁）

### 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

### mysql是如何预防死锁的？

1. innodb\_lock\_wait\_timeout 等待锁超时回滚事务

直观方法是在两个事务相互等待时，当一个等待时间超过设置的某一阀值时，对其中一个事务进行回滚，另一个事务就能继续执行。

2. wait-for graph算法来主动进行死锁检测

每当加锁请求无法立即满足需要并进入等待时，wait-for graph算法都会被触发。

wait-for graph要求数据库保存以下两种信息：

锁的信息链表

事务等待链表

通过上述链表可以构造出一张图，而在这个图中若存在回路，就代表存在死锁，因此资源间相互发生等待。在wait-for graph中，事务为图中的节点。而在图中，事务T1指向T2边的定义为：

事务T1等待事务T2所占用的资源

事务T1最终等待T2所占用的资源，也就是事务之间在等待相同的资源，而事务T1发生在事务T2的后面

### 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。

## MVCC？

MVCC叫做多版本并发控制，实际上就是保存了数据在某个时间节点的快照。

我们每行数实际上隐藏了两列，创建时间版本号，过期(删除)时间版本号，每开始一个新的事务，版本号都会自动递增。

为什么需要MVCC呢？数据库通常使用锁来实现隔离性。最原生的锁，锁住一个资源后会禁止其他任何线程访问同一个资源。但是很多应用的一个特点都是读多写少的场景，很多数据的读取次数远大于修改的次数，而读取数据间互相排斥显得不是很必要。所以就使用了一种读写锁的方法，读锁和读锁之间不互斥，而写锁和写锁、读锁都互斥。这样就很大提升了系统的并发能力。之后人们发现并发读还是不够，又提出了能不能让读写之间也不冲突的方法，就是读取数据时通过一种类似快照的方式将数据保存下来，这样读锁就和写锁不冲突了，不同的事务session会看到自己特定版本的数据。当然快照是一种概念模型，不同的数据库可能用不同的方式来实现这种功能。

Redo log, bin log, Undo log

InnoDB中通过undo log实现了数据的多版本，而并发控制通过锁来实现。

undo log除了实现MVCC外，还用于事务的回滚。MySQL Innodb中存在多种日志，除了错误日志、查询日志外，还有很多和数据持久性、一致性有关的日志。

binlog，是mysql服务层产生的日志，常用来进行数据恢复、数据库复制，常见的mysql主从架构，就是采用slave同步master的binlog实现的, 另外通过解析binlog能够实现mysql到其他数据源（如ElasticSearch)的数据复制。

redo log记录了数据操作在物理层面的修改，mysql中使用了大量缓存，缓存存在于内存中，修改操作时会直接修改内存，而不是立刻修改磁盘，当内存和磁盘的数据不一致时，称内存中的数据为脏页(dirty page)。为了保证数据的安全性，事务进行中时会不断的产生redo log，在事务提交时进行一次flush操作，保存到磁盘中, redo log是按照顺序写入的，磁盘的顺序读写的速度远大于随机读写。当数据库或主机失效重启时，会根据redo log进行数据的恢复，如果redo log中有事务提交，则进行事务提交修改数据。这样实现了事务的原子性、一致性和持久性。

undo log: 除了记录redo log外，当进行数据修改时还会记录undo log，undo log用于数据的撤回操作，它记录了修改的反向操作，比如，插入对应删除，修改对应修改为原来的数据，通过undo log可以实现事务回滚，并且可以根据undo log回溯到某个特定的版本的数据，实现MVCC。

## 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

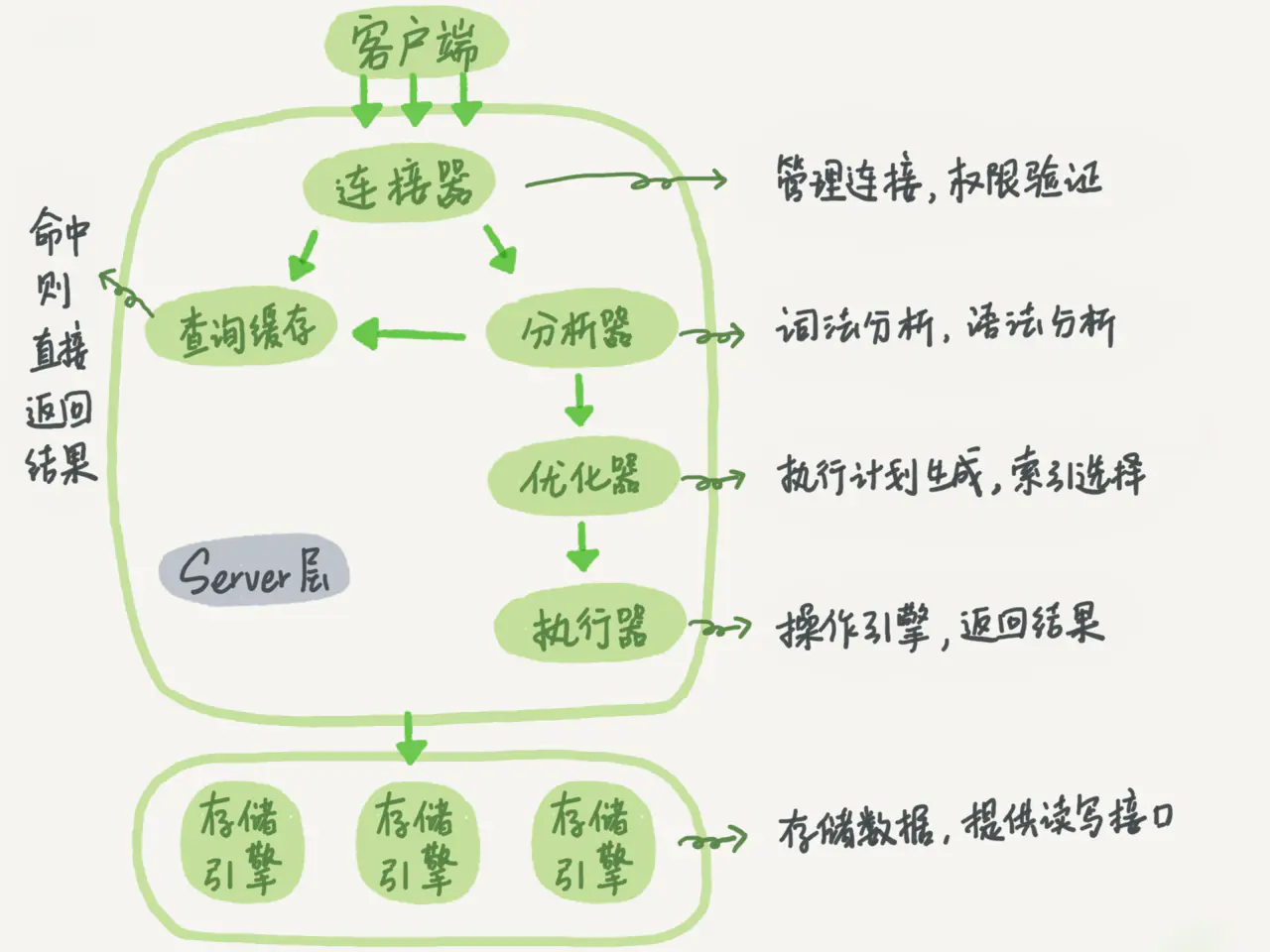
### 优化UNION查询

UNION ALL的效率高于UNION

### 优化WHERE子句

## SQL语句是如何执行的?

### SELECT是如何执行的



### UPDATE是如何执行的

update同样需要经过以上步骤,不过在执行时有所不同.

select时有缓存,其实update也有.试想,如果某一时刻mysql接到大量的写请求,对于每个请求mysql都得去写一次磁盘,那用于磁盘I/O的时间就会严重影响mysql的性能,导致qps上不去.

mysql的策略是先写日志(内存缓存中),等到不忙的时候再去写磁盘.

这个技术叫做WAL,Write-Ahead Logging.和Redis刚好是反过来的,redis先保存数据,再写aof.

这里的日志指的就是redo log,重做日志.有了redo log，InnoDB就可以保证即使数据库发生异常重启，之前提交的记录都不会丢失，这个能力称为crash-safe。

InnoDB的redo log是固定大小的，比如可以配置为一组4个文件，每个文件的大小是1GB，那么redo log总共就可以记录4GB的操作。从头开始写，写到末尾就又回到开头循环写，

write\_pos是记录点,每写一份数据,它就往后挪一挪.

check\_point是擦除点,每把一份数据写回磁盘,它就往后挪一挪.

可以这么理解,write\_pos在前面写(记录数据),check\_point在后面追(把这些数据写回磁盘).当check\_point追上write\_pos,就说明全部数据都写回磁盘了.当write\_pos追上check\_point,说明缓存区写满了,暂时无法处理新的写请求.

除了redo log,更新数据时也需要写binlog.它们的不同在于:

redo log是InnoDB引擎特有的；binlog是MySQL的Server层实现的，所有引擎都可以使用。redo log是物理日志，记录的是“在某个数据页上做了什么修改”；binlog是逻辑日志，记录的是这个语句的原始逻辑，比如“给ID=2这一行的c字段加1 ”。redo log是循环写的，空间固定会用完；binlog是可以追加写入的。“追加写”是指binlog文件写到一定大小后会切换到下一个，并不会覆盖以前的日志。

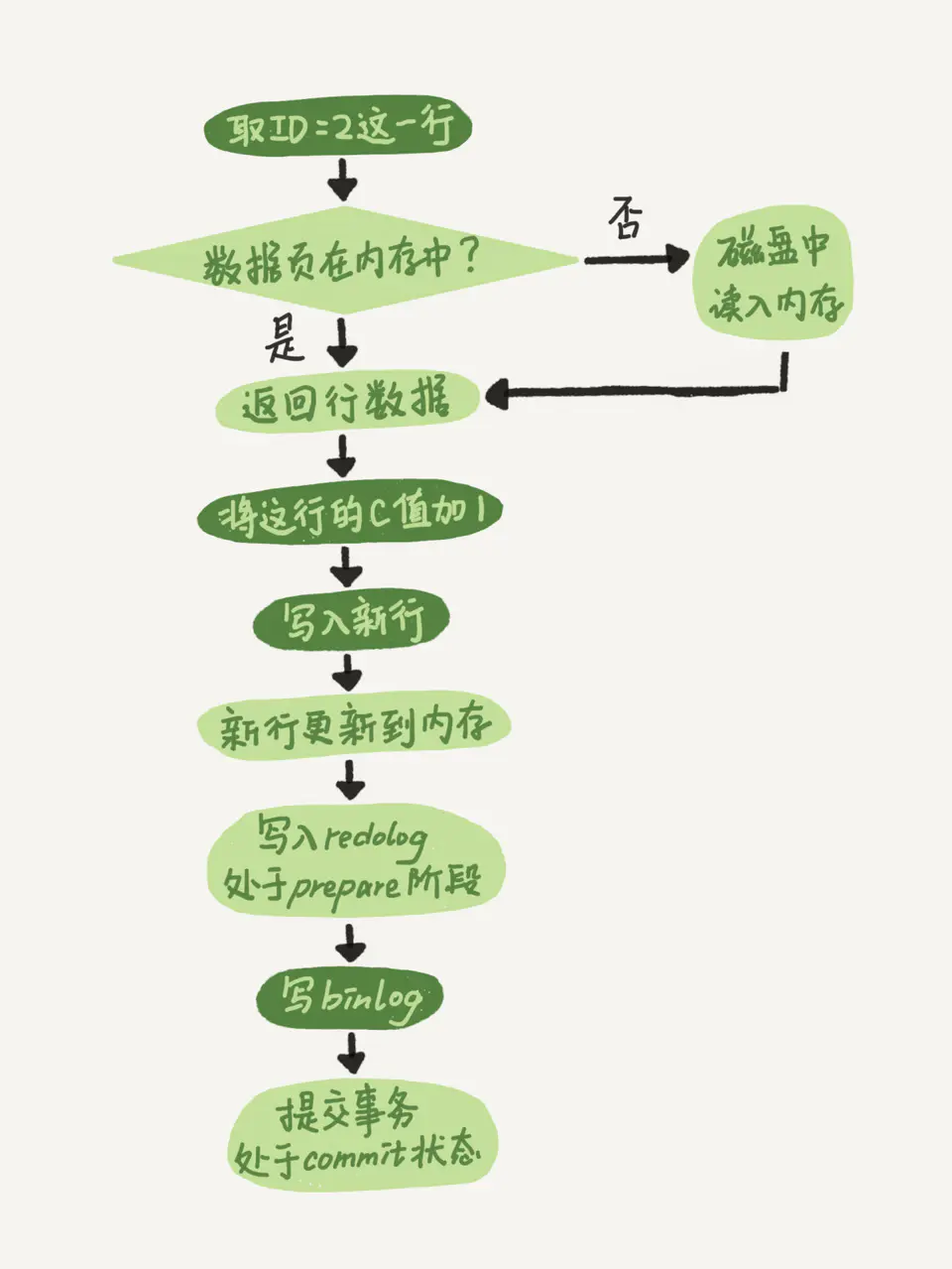
所以redo log用于数据库宕机时保证持久性,而binlog用于还原数据版本.可以这么理解,redo log是主,binlog是备.

对redo log和binlog的操作需要是原子性的.

如果一个事务先写redo log,写完以后数据库重启binlog丢失了,那么以后利用binlog恢复的话就会丢失这个事务.如果一个事务先写binlog,写完写完以后数据库重启redo log丢失了,那么重启后redo log写回磁盘的数据就丢失了这个事务,可是以后利用binlog恢复的话这个事务又会回来.

mysql使用二阶段提交来保证redo log和binlog的操作是原子性的,要么全部保存,要么全部丢失.

以下是它们的二阶段提交流程:



### ORDER BY是如何执行的

在select 阶段,如果有where条件,那么使用索引的逻辑和没有order by是相同的.

如果没有where条件,order by的字段被索引覆盖,且select的字段刚好都在这个索引上(不需要回表的情况),那么就会使用这个索引.否则就全表扫描.

在select完之后,如果数据的排序已经满足要求了,就不需要排序.否则就要.

mysql会在内存中开辟一块空间用于排序,这个叫内排序.

如果数据量太大,缓冲区放不下,就得利用硬盘来协助排序了,这个叫外排序.

### GROUP BY是如何执行的

在select阶段,如果有where条件,那么where能走索引就肯定走索引,不过还会判断group by能不能走索引,如果group的字段也能被索引覆盖,那么就会使用联合索引,先取出where的索引列,再到group by的索引中对where的索引列进行分组计算,最后取数据.

如果没有where条件,就直接判断group by能不能走索引,能的话就利用对应的索引进行分组.

不过这里要强调,group by的字段必须完全包含在索引中,而不仅仅只是满足最左前缀原则.假设索引列为(x,y)那么仅当group by的字段为(x)或者(x,y)时才能使用索引,(x,y,z)这样多出了z这列的情况会使索引失效.

group by的索引逻辑和回表与否没有关系,所以select的字段随意.

如果无法利用索引,mysql就得分配一个临时表(temporary)用于分组

此外group by还涉及松散索引和紧凑索引等概念,不过这些会留到索引优化的文章里再详细展开了.

## 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

mysql主从同步的原理

master提交完事务后，写入binlog

slave连接到master，获取binlog

master创建dump线程，推送binglog到slave

slave启动一个IO线程读取同步过来的master的binlog，记录到relay log中继日志中

slave再开启一个sql线程读取relay log事件并在slave执行，完成同步

slave记录自己的binglog

### 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；

多路I/O复用模型是利用 select、poll、epoll 可以同时监察多个流的 I/O 事件的能力，在空闲的时候，会把当前线程阻塞掉，当有一个或多个流有 I/O 事件时，就从阻塞态中唤醒，于是程序就会轮询一遍所有的流（epoll 是只轮询那些真正发出了事件的流），并且只依次顺序的处理就绪的流，这种做法就避免了大量的无用操作。

这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。

采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络 IO 的时间消耗），且 Redis 在内存中操作数据的速度非常快，也就是说内存内的操作不会成为影响Redis性能的瓶颈，主要由以上几点造就了 Redis 具有很高的吞吐量。

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；

## 为什么Redis是单线程的？

我们首先要明白，上边的种种分析，都是为了营造一个Redis很快的氛围！官方FAQ表示，因为Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了（毕竟采用多线程会有很多麻烦！）。

看到这里，你可能会气哭！本以为会有什么重大的技术要点才使得Redis使用单线程就可以这么快，没想到就是一句官方看似糊弄我们的回答！但是，我们已经可以很清楚的解释了为什么Redis这么快，并且正是由于在单线程模式的情况下已经很快了，就没有必要在使用多线程了！

但是，我们使用单线程的方式是无法发挥多核CPU 性能，不过我们可以通过在单机开多个Redis 实例来完善！

## 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。

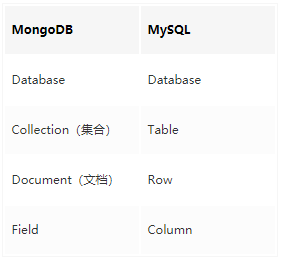
6. 为了Master的稳定性，主从复制不要用图状结构，用单向链表结构更稳定，即主从关系为：Master<–Slave1<–Slave2<–Slave3…，这样的结构也方便解决单点故障问题，实现Slave对Master的替换，也即，如果Master挂了，可以立马启用Slave1做Master，其他不变。

\*\*缓存降级\*\*的最终目的是保证核心服务可用，即使是有损的。

# MongoDB

基于分布式文件存储的数据库，介于非关系型和关系型数据库之间。

MongoDB和MySQL对比



## MongoDB有什么特点

（1）面向集合存储数据，数据以json格式存储；

（2）查询功能强大，几乎可以实现所有关系型数据库的单表查询功能；

（3）自带分布式文件系统，方便集群部署；

（4）自带了对map-reduce运算框架的支持。

## MongoDB为什么快

（1）没有事务约束；

（2）使用的内存映射技术 ，写入数据时候只要在内存里完成就可以返回给应用程序，写入磁盘是异步操作；热点数据放入内存；数据在磁盘中相对集中，减少随机读写耗费的磁头定位时间；

（3）没有join，不使用关系。没有关系的存在，就表示每个数据都好比是拥有一个单独的存储空间，然后一个聚集索引来指向。

## MongoDB适用场景

（1）网站实时数据处理，如PV、UV、统计商品的收藏次数、加入购物车次数、曝光次数等。

（2）服务器日志记录；

（3）大数据量，TB甚至PB级别的数据存储；

（4）数据模型经常变化，需要支持水平扩展

（5）游戏场景，使用 MongoDB 存储游戏用户信息，用户的装备、积分等直接以内嵌文档的形式存储，方便查询、更新

物流场景，使用 MongoDB 存储订单信息，订单状态在运送过程中会不断更新，以 MongoDB 内嵌数组的形式来存储，一次查询就能将订单所有的变更读取出来。

社交场景，使用 MongoDB 存储存储用户信息，以及用户发表的朋友圈信息，通过地理位置索引实现附近的人、地点等功能

物联网场景，使用 MongoDB 存储所有接入的智能设备信息，以及设备汇报的日志信息，并对这些信息进行多维度的分析

视频直播，使用 MongoDB 存储用户信息、礼物信息等。

## MongoDB不适用场景

（1）数据修改需要严格的事物控制；

（2）功能的业务关系复杂、涉及到多表关联；

（3）比较重要的业务数据，如订单、商品、用户等数据。

## MongoDB集合

MongoDB的集合，可类比于MySQL的表，字段（field）可类比于MySQL的列（column）。区别是，集合的filed是可动态扩展的，并且数据类型没有限制，不像Mysql需要先定义列名、列的字段类型等。

此外，集合无需像mysql的表一样事先定义。使用db.collectionName.insert()语句，插入一条json数据，该集合便生成了。

集合由文档组成，每个文档就是一条json数据。

# 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

# 各种三方件

## 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

### nginx可以提供的服务

web 服务.

负载均衡 （反向代理）

web cache（web 缓存）

### nginx 的优点

高并发。静态小文件

占用资源少。2万并发、10个线程，内存消耗几百M。

功能种类比较多。web,cache,proxy。每一个功能都不是特别强。

支持epoll模型，使得nginx可以支持高并发。

nginx 配合动态服务和Apache有区别。（FASTCGI 接口）

利用nginx可以对IP限速，可以限制连接数。

配置简单，更灵活。

### nginx应用场合

静态服务器。（图片，视频服务）另一个lighttpd。并发几万，html，js，css，flv，jpg，gif等。

动态服务，nginx——fastcgi 的方式运行PHP，jsp。（PHP并发在500-1500，MySQL 并发在300-1500）。

反向代理，负载均衡。日pv2000W以下，都可以直接用nginx做代理。

缓存服务。类似 SQUID,VARNISH。

### 主流web服务产品对比说明

5.1 Apache-特性

2.2版本本身稳定强大，据官方说：其2.4版本性能更强。

prefork模式取消了进程创建开销，性能很高。

处理动态业务数据时，因关联到后端的引擎和数据库，瓶颈不在与Apache本身。

高并发时消耗系统资源相对多一些。

基于传统的select模型。

扩展库，DSO方法。

5.2 nginx-特性

基于异步IO模型，（epoll，kqueue），性能强，能够支持上万并发。

对小文件支持很好，性能很高（限静态小文件1M）。

代码优美，扩展库必须编译进主程序。

消耗代码资源比较低。

lighttpd（百度贴吧，豆瓣）

基于异步IO模式，性能和nginx相近。

扩展库是SO模式，比nginx要灵活。

8.通过差距（mod\_secdownload）可实现文件URL地址加密。

### nginx配置文件

配置基础配置文件

worker\_processes 1;

events {

worker\_connections 1024;

}

http {

include mime.types;

default\_type application/octet-stream;

sendfile on;

keepalive\_timeout 65;

server {

listen 80;

server\_name localhost;

location / {

root html;

index index.html index.htm;

}

error\_page 500 502 503 504 /50x.html;

location = /50x.html {

root html;

}

}

}

### 测试配置文件是否正常

shell> /data/nginx/sbin/nginx -t

nginx: the configuration file /data/nginx-1.10.3/conf/nginx.conf syntax is ok

nginx: configuration file /data/nginx-1.10.3/conf/nginx.conf test is successful

shell> curl -I http://localhost

HTTP/1.1 200 OK

## Tomcat

### Tomcat是什么？

Tomcat 服务器Apache软件基金会项目中的一个核心项目，是一个免费的开放源代码的Web 应用服务器，属于轻量级应用服务器，在中小型系统和并发访问用户不是很多的场合下被普遍使用，是开发和调试JSP 程序的首选。

### Tomcat的缺省端口是多少，怎么修改

找到Tomcat目录下的conf文件夹

进入conf文件夹里面找到server.xml文件

打开server.xml文件

在server.xml文件里面找到下列信息

把Connector标签的8080端口改成你想要的端口

<Service name="Catalina">

<Connector port="8080" protocol="HTTP/1.1"

connectionTimeout="20000"

redirectPort="8443" />

### tomcat 有哪几种Connector 运行模式(优化)？

下面，我们先大致了解Tomcat Connector的三种运行模式。

BIO：同步并阻塞 一个线程处理一个请求。缺点：并发量高时，线程数较多，浪费资源。Tomcat7或以下，在Linux系统中默认使用这种方式。

​ 配制项：protocol=”HTTP/1.1”

NIO：同步非阻塞IO

利用Java的异步IO处理，可以通过少量的线程处理大量的请求，可以复用同一个线程处理多个connection(多路复用)。

Tomcat8在Linux系统中默认使用这种方式。

Tomcat7必须修改Connector配置来启动。

配制项：protocol=”org.apache.coyote.http11.Http11NioProtocol”

备注：我们常用的Jetty，Mina，ZooKeeper等都是基于java nio实现.

APR：即Apache Portable Runtime，从操作系统层面解决io阻塞问题。\*\*AIO方式，\*\*异步非阻塞IO(Java NIO2又叫AIO) 主要与NIO的区别主要是操作系统的底层区别.可以做个比喻:比作快递，NIO就是网购后要自己到官网查下快递是否已经到了(可能是多次)，然后自己去取快递；AIO就是快递员送货上门了(不用关注快递进度)。

配制项：protocol=”org.apache.coyote.http11.Http11AprProtocol”

备注：需在本地服务器安装APR库。Tomcat7或Tomcat8在Win7或以上的系统中启动默认使用这种方式。Linux如果安装了apr和native，Tomcat直接启动就支持apr。

### Tomcat有几种部署方式？

在Tomcat中部署Web应用的方式主要有如下几种：

利用Tomcat的自动部署。

把web应用拷贝到webapps目录。Tomcat在启动时会加载目录下的应用，并将编译后的结果放入work目录下。

使用Manager App控制台部署。

在tomcat主页点击“Manager App” 进入应用管理控制台，可以指定一个web应用的路径或war文件。

修改conf/server.xml文件部署。

修改conf/server.xml文件，增加Context节点可以部署应用。

增加自定义的Web部署文件。

在conf/Catalina/localhost/ 路径下增加 xyz.xml文件，内容是Context节点，可以部署应用。

### tomcat容器是如何创建servlet类实例？用到了什么原理？

当容器启动时，会读取在webapps目录下所有的web应用中的web.xml文件，然后对 xml文件进行解析，并读取servlet注册信息。然后，将每个应用中注册的servlet类都进行加载，并通过 反射的方式实例化。（有时候也是在第一次请求时实例化）

在servlet注册时加上1如果为正数，则在一开始就实例化，如果不写或为负数，则第一次请求实例化。

### Tomcat工作模式

Tomcat作为servlet容器，有三种工作模式：

1、独立的servlet容器，servlet容器是web服务器的一部分；

2、进程内的servlet容器，servlet容器是作为web服务器的插件和java容器的实现，web服务器插件在内部地址空间打开一个jvm使得java容器在内部得以运行。反应速度快但伸缩性不足；

3、进程外的servlet容器，servlet容器运行于web服务器之外的地址空间，并作为web服务器的插件和java容器实现的结合。反应时间不如进程内但伸缩性和稳定性比进程内优；

进入Tomcat的请求可以根据Tomcat的工作模式分为如下两类：

### Tomcat作为应用程序服务器：请求来自于前端的web服务器，这可能是Apache, IIS, Nginx等；

Tomcat作为独立服务器：请求来自于web浏览器；

面试时问到Tomcat相关问题的几率并不高，正式因为如此，很多人忽略了对Tomcat相关技能的掌握，下面这一篇文章整理了Tomcat相关的系统架构，介绍了Server、Service、Connector、Container之间的关系，各个模块的功能，可以说把这几个掌握住了，Tomcat相关的面试题你就不会有任何问题了！另外，在面试的时候你还要有意识无意识的往Tomcat这个地方引，就比如说常见的Spring MVC的执行流程，一个URL的完整调用链路，这些相关的题目你是可以往Tomcat处理请求的这个过程去说的！掌握了Tomcat这些技能，面试官一定会佩服你的！

学了本章之后你应该明白的是：

Server、Service、Connector、Container四大组件之间的关系和联系，以及他们的主要功能点；

Tomcat执行的整体架构，请求是如何被一步步处理的；

Engine、Host、Context、Wrapper相关的概念关系；

Container是如何处理请求的；

Tomcat用到的相关设计模式；

Tomcat顶层架构

俗话说，站在巨人的肩膀上看世界，一般学习的时候也是先总览一下整体，然后逐个部分个个击破，最后形成思路，了解具体细节，Tomcat的结构很复杂，但是 Tomcat 非常的模块化，找到了 Tomcat 最核心的模块，问题才可以游刃而解，了解了 Tomcat 的整体架构对以后深入了解 Tomcat 来说至关重要！

先上一张Tomcat的顶层结构图（图A），如下：

Tomcat中最顶层的容器是Server，代表着整个服务器，从上图中可以看出，一个Server可以包含至少一个Service，即可以包含多个Service，用于具体提供服务。

Service主要包含两个部分：Connector和Container。从上图中可以看出 Tomcat 的心脏就是这两个组件，他们的作用如下：

Connector用于处理连接相关的事情，并提供Socket与Request请求和Response响应相关的转化;

Container用于封装和管理Servlet，以及具体处理Request请求；

一个Tomcat中只有一个Server，一个Server可以包含多个Service，一个Service只有一个Container，但是可以有多个Connectors，这是因为一个服务可以有多个连接，如同时提供Http和Https链接，也可以提供向相同协议不同端口的连接，示意图如下（Engine、Host、Context下面会说到）：

多个 Connector 和一个 Container 就形成了一个 Service，有了 Service 就可以对外提供服务了，但是 Service 还要一个生存的环境，必须要有人能够给她生命、掌握其生死大权，那就非 Server 莫属了！所以整个 Tomcat 的生命周期由 Server 控制。

另外，上述的包含关系或者说是父子关系，都可以在tomcat的conf目录下的server.xml配置文件中看出，下图是删除了注释内容之后的一个完整的server.xml配置文件（Tomcat版本为8.0）

详细的配置文件内容可以到Tomcat官网查看：Tomcat配置文件

Server标签设置的端口号为8005，shutdown=”SHUTDOWN” ，表示在8005端口监听“SHUTDOWN”命令，如果接收到了就会关闭Tomcat。一个Server有一个Service，当然还可以进行配置，一个Service有多个Connector，Service左边的内容都属于Container的，Service下边是Connector。

### Tomcat顶层架构小结

Tomcat中只有一个Server，一个Server可以有多个Service，一个Service可以有多个Connector和一个Container；

Server掌管着整个Tomcat的生死大权；

Service 是对外提供服务的；

Connector用于接受请求并将请求封装成Request和Response来具体处理；

Container用于封装和管理Servlet，以及具体处理request请求；

知道了整个Tomcat顶层的分层架构和各个组件之间的关系以及作用，对于绝大多数的开发人员来说Server和Service对我们来说确实很远，而我们开发中绝大部分进行配置的内容是属于Connector和Container的，所以接下来介绍一下Connector和Container。

Connector和Container的微妙关系

由上述内容我们大致可以知道一个请求发送到Tomcat之后，首先经过Service然后会交给我们的Connector，Connector用于接收请求并将接收的请求封装为Request和Response来具体处理，Request和Response封装完之后再交由Container进行处理，Container处理完请求之后再返回给Connector，最后在由Connector通过Socket将处理的结果返回给客户端，这样整个请求的就处理完了！

Connector最底层使用的是Socket来进行连接的，Request和Response是按照HTTP协议来封装的，所以Connector同时需要实现TCP/IP协议和HTTP协议！

Tomcat既然需要处理请求，那么肯定需要先接收到这个请求，接收请求这个东西我们首先就需要看一下Connector！

### Connector架构分析

Connector用于接受请求并将请求封装成Request和Response，然后交给Container进行处理，Container处理完之后在交给Connector返回给客户端。

因此，我们可以把Connector分为四个方面进行理解：

### Connector如何接受请求的？

如何将请求封装成Request和Response的？

封装完之后的Request和Response如何交给Container进行处理的？

Container处理完之后如何交给Connector并返回给客户端的？

首先看一下Connector的结构图（图B），如下所示：

Connector就是使用ProtocolHandler来处理请求的，不同的ProtocolHandler代表不同的连接类型，比如：Http11Protocol使用的是普通Socket来连接的，Http11NioProtocol使用的是NioSocket来连接的。

其中ProtocolHandler由包含了三个部件：Endpoint、Processor、Adapter。

Endpoint用来处理底层Socket的网络连接，Processor用于将Endpoint接收到的Socket封装成Request，Adapter用于将Request交给Container进行具体的处理。

Endpoint由于是处理底层的Socket网络连接，因此Endpoint是用来实现TCP/IP协议的，而Processor用来实现HTTP协议的，Adapter将请求适配到Servlet容器进行具体的处理。

Endpoint的抽象实现AbstractEndpoint里面定义的Acceptor和AsyncTimeout两个内部类和一个Handler接口。Acceptor用于监听请求，AsyncTimeout用于检查异步Request的超时，Handler用于处理接收到的Socket，在内部调用Processor进行处理。

至此，我们应该很轻松的回答1，2，3的问题了，但是4还是不知道，那么我们就来看一下Container是如何进行处理的以及处理完之后是如何将处理完的结果返回给Connector的？

### Container架构分析

Container用于封装和管理Servlet，以及具体处理Request请求，在Container内部包含了4个子容器，结构图如下（图C）：

在这里插入图片描述

4个子容器的作用分别是：

Engine：引擎，用来管理多个站点，一个Service最多只能有一个Engine；

Host：代表一个站点，也可以叫虚拟主机，通过配置Host就可以添加站点；

Context：代表一个应用程序，对应着平时开发的一套程序，或者一个WEB-INF目录以及下面的web.xml文件；

Wrapper：每一Wrapper封装着一个Servlet；

下面找一个Tomcat的文件目录对照一下，如下图所示：

在这里插入图片描述

Context和Host的区别是Context表示一个应用，我们的Tomcat中默认的配置下webapps下的每一个文件夹目录都是一个Context，其中ROOT目录中存放着主应用，其他目录存放着子应用，而整个webapps就是一个Host站点。

我们访问应用Context的时候，如果是ROOT下的则直接使用域名就可以访问，例如：www.baidu.com，如果是Host（webapps）下的其他应用，则可以使用www.baidu.com/docs进行访问，当然默认指定的根应用（ROOT）是可以进行设定的，只不过Host站点下默认的主应用是ROOT目录下的。

看到这里我们知道Container是什么，但是还是不知道Container是如何进行请求处理的以及处理完之后是如何将处理完的结果返回给Connector的？别急！下边就开始探讨一下Container是如何进行处理的！

### Container如何处理请求的

Container处理请求是使用Pipeline-Valve管道来处理的！（Valve是阀门之意）

Pipeline-Valve是责任链模式，责任链模式是指在一个请求处理的过程中有很多处理者依次对请求进行处理，每个处理者负责做自己相应的处理，处理完之后将处理后的结果返回，再让下一个处理者继续处理。

但是！Pipeline-Valve使用的责任链模式和普通的责任链模式有些不同！区别主要有以下两点：

每个Pipeline都有特定的Valve，而且是在管道的最后一个执行，这个Valve叫做BaseValve，BaseValve是不可删除的；

在上层容器的管道的BaseValve中会调用下层容器的管道。

我们知道Container包含四个子容器，而这四个子容器对应的BaseValve分别在：StandardEngineValve、StandardHostValve、StandardContextValve、StandardWrapperValve。

Connector在接收到请求后会首先调用最顶层容器的Pipeline来处理，这里的最顶层容器的Pipeline就是EnginePipeline（Engine的管道）；

在Engine的管道中依次会执行EngineValve1、EngineValve2等等，最后会执行StandardEngineValve，在StandardEngineValve中会调用Host管道，然后再依次执行Host的HostValve1、HostValve2等，最后在执行StandardHostValve，然后再依次调用Context的管道和Wrapper的管道，最后执行到StandardWrapperValve。

当执行到StandardWrapperValve的时候，会在StandardWrapperValve中创建FilterChain，并调用其doFilter方法来处理请求，这个FilterChain包含着我们配置的与请求相匹配的Filter和Servlet，其doFilter方法会依次调用所有的Filter的doFilter方法和Servlet的service方法，这样请求就得到了处理！

当所有的Pipeline-Valve都执行完之后，并且处理完了具体的请求，这个时候就可以将返回的结果交给Connector了，Connector在通过Socket的方式将结果返回给客户端。

## Netty

### Netty 是什么？

Netty是 一个异步事件驱动的网络应用程序框架，用于快速开发可维护的高性能协议服务器和客户端。Netty是基于nio的，它封装了jdk的nio，让我们使用起来更加方法灵活。

### Netty 的特点是什么？

高并发：Netty 是一款基于 NIO（Nonblocking IO，非阻塞IO）开发的网络通信框架，对比于 BIO（Blocking I/O，阻塞IO），他的并发性能得到了很大提高。

传输快：Netty 的传输依赖于零拷贝特性，尽量减少不必要的内存拷贝，实现了更高效率的传输。

封装好：Netty 封装了 NIO 操作的很多细节，提供了易于使用调用接口。

### Netty 的优势有哪些？

使用简单：封装了 NIO 的很多细节，使用更简单。

功能强大：预置了多种编解码功能，支持多种主流协议。

定制能力强：可以通过 ChannelHandler 对通信框架进行灵活地扩展。

性能高：通过与其他业界主流的 NIO 框架对比，Netty 的综合性能最优。

稳定：Netty 修复了已经发现的所有 NIO 的 bug，让开发人员可以专注于业务本身。

社区活跃：Netty 是活跃的开源项目，版本迭代周期短，bug 修复速度快。

### Netty 的应用场景有哪些？

典型的应用有：阿里分布式服务框架 Dubbo，默认使用 Netty 作为基础通信组件，还有 RocketMQ 也是使用 Netty 作为通讯的基础。

### Netty 高性能表现在哪些方面？

IO 线程模型：同步非阻塞，用最少的资源做更多的事。

内存零拷贝：尽量减少不必要的内存拷贝，实现了更高效率的传输。

内存池设计：申请的内存可以重用，主要指直接内存。内部实现是用一颗二叉查找树管理内存分配情况。

串形化处理读写：避免使用锁带来的性能开销。

高性能序列化协议：支持 protobuf 等高性能序列化协议。

### BIO、NIO和AIO的区别？

BIO：一个连接一个线程，客户端有连接请求时服务器端就需要启动一个线程进行处理。线程开销大。

伪异步IO：将请求连接放入线程池，一对多，但线程还是很宝贵的资源。

NIO：一个请求一个线程，但客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

AIO：一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理，

BIO是面向流的，NIO是面向缓冲区的；BIO的各种流是阻塞的。而NIO是非阻塞的；BIO的Stream是单向的，而NIO的channel是双向的。

NIO的特点：事件驱动模型、单线程处理多任务、非阻塞I/O，I/O读写不再阻塞，而是返回0、基于block的传输比基于流的传输更高效、更高级的IO函数zero-copy、IO多路复用大大提高了Java网络应用的可伸缩性和实用性。基于Reactor线程模型。

在Reactor模式中，事件分发器等待某个事件或者可应用或个操作的状态发生，事件分发器就把这个事件传给事先注册的事件处理函数或者回调函数，由后者来做实际的读写操作。如在Reactor中实现读：注册读就绪事件和相应的事件处理器、事件分发器等待事件、事件到来，激活分发器，分发器调用事件对应的处理器、事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还控制权。

### NIO的组成？

Buffer：与Channel进行交互，数据是从Channel读入缓冲区，从缓冲区写入Channel中的

flip方法 ： 反转此缓冲区，将position给limit，然后将position置为0，其实就是切换读写模式

clear方法 ：清除此缓冲区，将position置为0，把capacity的值给limit。

rewind方法 ： 重绕此缓冲区，将position置为0

DirectByteBuffer可减少一次系统空间到用户空间的拷贝。但Buffer创建和销毁的成本更高，不可控，通常会用内存池来提高性能。直接缓冲区主要分配给那些易受基础系统的本机I/O 操作影响的大型、持久的缓冲区。如果数据量比较小的中小应用情况下，可以考虑使用heapBuffer，由JVM进行管理。

Channel：表示 IO 源与目标打开的连接，是双向的，但不能直接访问数据，只能与Buffer 进行交互。通过源码可知，FileChannel的read方法和write方法都导致数据复制了两次！

Selector可使一个单独的线程管理多个Channel，open方法可创建Selector，register方法向多路复用器器注册通道，可以监听的事件类型：读、写、连接、accept。注册事件后会产生一个SelectionKey：它表示SelectableChannel 和Selector 之间的注册关系，wakeup方法：使尚未返回的第一个选择操作立即返回，唤醒的

原因是：注册了新的channel或者事件；channel关闭，取消注册；优先级更高的事件触发（如定时器事件），希望及时处理。

Selector在Linux的实现类是EPollSelectorImpl，委托给EPollArrayWrapper实现，其中三个native方法是对epoll的封装，而EPollSelectorImpl. implRegister方法，通过调用epoll\_ctl向epoll实例中注册事件，还将注册的文件描述符(fd)与SelectionKey的对应关系添加到fdToKey中，这个map维护了文件描述符与SelectionKey的映射。

fdToKey有时会变得非常大，因为注册到Selector上的Channel非常多（百万连接）；过期或失效的Channel没有及时关闭。fdToKey总是串行读取的，而读取是在select方法中进行的，该方法是非线程安全的。

Pipe：两个线程之间的单向数据连接，数据会被写到sink通道，从source通道读取

NIO的服务端建立过程：Selector.open()：打开一个Selector；ServerSocketChannel.open()：创建服务端的Channel；bind()：绑定到某个端口上。并配置非阻塞模式；register()：注册Channel和关注的事件到Selector上；select()轮询拿到已经就绪的事件

### Netty的线程模型？

Netty通过Reactor模型基于多路复用器接收并处理用户请求，内部实现了两个线程池，boss线程池和work线程池，其中boss线程池的线程负责处理请求的accept事件，当接收到accept事件的请求时，把对应的socket封装到一个NioSocketChannel中，并交给work线程池，其中work线程池负责请求的read和write事件，由对应的Handler处理。

单线程模型：所有I/O操作都由一个线程完成，即多路复用、事件分发和处理都是在一个Reactor线程上完成的。既要接收客户端的连接请求,向服务端发起连接，又要发送/读取请求或应答/响应消息。一个NIO 线程同时处理成百上千的链路，性能上无法支撑，速度慢，若线程进入死循环，整个程序不可用，对于高负载、大并发的应用场景不合适。

多线程模型：有一个NIO 线程（Acceptor） 只负责监听服务端，接收客户端的TCP 连接请求；NIO 线程池负责网络IO 的操作，即消息的读取、解码、编码和发送；1 个NIO 线程可以同时处理N 条链路，但是1 个链路只对应1 个NIO 线程，这是为了防止发生并发操作问题。但在并发百万客户端连接或需要安全认证时，一个Acceptor 线程可能会存在性能不足问题。

主从多线程模型：Acceptor 线程用于绑定监听端口，接收客户端连接，将SocketChannel 从主线程池的Reactor 线程的多路复用器上移除，重新注册到Sub 线程池的线程上，用于处理I/O 的读写等操作，从而保证mainReactor只负责接入认证、握手等操作；

### TCP 粘包/拆包的原因及解决方法？

TCP是以流的方式来处理数据，一个完整的包可能会被TCP拆分成多个包进行发送，也可能把小的封装成一个大的数据包发送。

TCP粘包/分包的原因：

应用程序写入的字节大小大于套接字发送缓冲区的大小，会发生拆包现象，而应用程序写入数据小于套接字缓冲区大小，网卡将应用多次写入的数据发送到网络上，这将会发生粘包现象；

进行MSS大小的TCP分段，当TCP报文长度-TCP头部长度>MSS的时候将发生拆包

以太网帧的payload（净荷）大于MTU（1500字节）进行ip分片。

解决方法

消息定长：FixedLengthFrameDecoder类

包尾增加特殊字符分割：

行分隔符类：LineBasedFrameDecoder

或自定义分隔符类 ：DelimiterBasedFrameDecoder

将消息分为消息头和消息体：LengthFieldBasedFrameDecoder类。分为有头部的拆包与粘包、长度字段在前且有头部的拆包与粘包、多扩展头部的拆包与粘包。

### 什么是 Netty 的零拷贝？

Netty 的零拷贝主要包含三个方面：

Netty 的接收和发送 ByteBuffer 采用 DIRECT BUFFERS，使用堆外直接内存进行 Socket 读写，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存（HEAP BUFFERS）进行 Socket 读写，JVM 会将堆内存 Buffer 拷贝一份到直接内存中，然后才写入 Socket 中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。

Netty 提供了组合 Buffer 对象，可以聚合多个 ByteBuffer 对象，用户可以像操作一个 Buffer 那样方便的对组合 Buffer 进行操作，避免了传统通过内存拷贝的方式将几个小 Buffer 合并成一个大的 Buffer。

Netty 的文件传输采用了 transferTo 方法，它可以直接将文件缓冲区的数据发送到目标 Channel，避免了传统通过循环 write 方式导致的内存拷贝问题。

### Netty 中有哪种重要组件？

Channel：Netty 网络操作抽象类，它除了包括基本的 I/O 操作，如 bind、connect、read、write 等。

EventLoop：主要是配合 Channel 处理 I/O 操作，用来处理连接的生命周期中所发生的事情。

ChannelFuture：Netty 框架中所有的 I/O 操作都为异步的，因此我们需要 ChannelFuture 的 addListener()注册一个 ChannelFutureListener 监听事件，当操作执行成功或者失败时，监听就会自动触发返回结果。

ChannelHandler：充当了所有处理入站和出站数据的逻辑容器。ChannelHandler 主要用来处理各种事件，这里的事件很广泛，比如可以是连接、数据接收、异常、数据转换等。

ChannelPipeline：为 ChannelHandler 链提供了容器，当 channel 创建时，就会被自动分配到它专属的 ChannelPipeline，这个关联是永久性的。

### Netty 发送消息有几种方式？

Netty 有两种发送消息的方式：

直接写入 Channel 中，消息从 ChannelPipeline 当中尾部开始移动；

写入和 ChannelHandler 绑定的 ChannelHandlerContext 中，消息从 ChannelPipeline 中的下一个 ChannelHandler 中移动。

### 默认情况 Netty 起多少线程？何时启动？

Netty 默认是 CPU 处理器数的两倍，bind 完之后启动。

### 了解哪几种序列化协议？

序列化（编码）是将对象序列化为二进制形式（字节数组），主要用于网络传输、数据持久化等；而反序列化（解码）则是将从网络、磁盘等读取的字节数组还原成原始对象，主要用于网络传输对象的解码，以便完成远程调用。

影响序列化性能的关键因素：序列化后的码流大小（网络带宽的占用）、序列化的性能（CPU资源占用）；是否支持跨语言（异构系统的对接和开发语言切换）。

Java默认提供的序列化：无法跨语言、序列化后的码流太大、序列化的性能差

XML，优点：人机可读性好，可指定元素或特性的名称。缺点：序列化数据只包含数据本身以及类的结构，不包括类型标识和程序集信息；只能序列化公共属性和字段；不能序列化方法；文件庞大，文件格式复杂，传输占带宽。适用场景：当做配置文件存储数据，实时数据转换。

JSON，是一种轻量级的数据交换格式，优点：兼容性高、数据格式比较简单，易于读写、序列化后数据较小，可扩展性好，兼容性好、与XML相比，其协议比较简单，解析速度比较快。缺点：数据的描述性比XML差、不适合性能要求为ms级别的情况、额外空间开销比较大。适用场景（可替代ＸＭＬ）：跨防火墙访问、可调式性要求高、基于Web browser的Ajax请求、传输数据量相对小，实时性要求相对低（例如秒级别）的服务。

Fastjson，采用一种“假定有序快速匹配”的算法。优点：接口简单易用、目前java语言中最快的json库。缺点：过于注重快，而偏离了“标准”及功能性、代码质量不高，文档不全。适用场景：协议交互、Web输出、Android客户端

Thrift，不仅是序列化协议，还是一个RPC框架。优点：序列化后的体积小, 速度快、支持多种语言和丰富的数据类型、对于数据字段的增删具有较强的兼容性、支持二进制压缩编码。缺点：使用者较少、跨防火墙访问时，不安全、不具有可读性，调试代码时相对困难、不能与其他传输层协议共同使用（例如HTTP）、无法支持向持久层直接读写数据，即不适合做数据持久化序列化协议。适用场景：分布式系统的RPC解决方案

Avro，Hadoop的一个子项目，解决了JSON的冗长和没有IDL的问题。优点：支持丰富的数据类型、简单的动态语言结合功能、具有自我描述属性、提高了数据解析速度、快速可压缩的二进制数据形式、可以实现远程过程调用RPC、支持跨编程语言实现。缺点：对于习惯于静态类型语言的用户不直观。适用场景：在Hadoop中做Hive、Pig和MapReduce的持久化数据格式。

Protobuf，将数据结构以.proto文件进行描述，通过代码生成工具可以生成对应数据结构的POJO对象和Protobuf相关的方法和属性。优点：序列化后码流小，性能高、结构化数据存储格式（XML JSON等）、通过标识字段的顺序，可以实现协议的前向兼容、结构化的文档更容易管理和维护。缺点：需要依赖于工具生成代码、支持的语言相对较少，官方只支持Java 、C++ 、python。适用场景：对性能要求高的RPC调用、具有良好的跨防火墙的访问属性、适合应用层对象的持久化

其它

protostuff 基于protobuf协议，但不需要配置proto文件，直接导包即可

Jboss marshaling 可以直接序列化java类， 无须实java.io.Serializable接口

Message pack 一个高效的二进制序列化格式

Hessian 采用二进制协议的轻量级remoting onhttp工具

kryo 基于protobuf协议，只支持java语言,需要注册（Registration），然后序列化（Output），反序列化（Input）

### 如何选择序列化协议？

具体场景

对于公司间的系统调用，如果性能要求在100ms以上的服务，基于XML的SOAP协议是一个值得考虑的方案。

基于Web browser的Ajax，以及Mobile app与服务端之间的通讯，JSON协议是首选。对于性能要求不太高，或者以动态类型语言为主，或者传输数据载荷很小的的运用场景，JSON也是非常不错的选择。

对于调试环境比较恶劣的场景，采用JSON或XML能够极大的提高调试效率，降低系统开发成本。

当对性能和简洁性有极高要求的场景，Protobuf，Thrift，Avro之间具有一定的竞争关系。

对于T级别的数据的持久化应用场景，Protobuf和Avro是首要选择。如果持久化后的数据存储在hadoop子项目里，Avro会是更好的选择。

对于持久层非Hadoop项目，以静态类型语言为主的应用场景，Protobuf会更符合静态类型语言工程师的开发习惯。由于Avro的设计理念偏向于动态类型语言，对于动态语言为主的应用场景，Avro是更好的选择。

如果需要提供一个完整的RPC解决方案，Thrift是一个好的选择。

如果序列化之后需要支持不同的传输层协议，或者需要跨防火墙访问的高性能场景，Protobuf可以优先考虑。

protobuf的数据类型有多种：bool、double、float、int32、int64、string、bytes、enum、message。protobuf的限定符：required: 必须赋值，不能为空、optional:字段可以赋值，也可以不赋值、repeated: 该字段可以重复任意次数（包括0次）、枚举；只能用指定的常量集中的一个值作为其值；

protobuf的基本规则：每个消息中必须至少留有一个required类型的字段、包含0个或多个optional类型的字段；repeated表示的字段可以包含0个或多个数据；[1,15]之内的标识号在编码的时候会占用一个字节（常用），[16,2047]之内的标识号则占用2个字节，标识号一定不能重复、使用消息类型，也可以将消息嵌套任意多层，可用嵌套消息类型来代替组。

protobuf的消息升级原则：不要更改任何已有的字段的数值标识；不能移除已经存在的required字段，optional和repeated类型的字段可以被移除，但要保留标号不能被重用。新添加的字段必须是optional或repeated。因为旧版本程序无法读取或写入新增的required限定符的字段。

编译器为每一个消息类型生成了一个.java文件，以及一个特殊的Builder类（该类是用来创建消息类接口的）。如：UserProto.User.Builder builder = UserProto.User.newBuilder();builder.build()；

Netty中的使用：ProtobufVarint32FrameDecoder 是用于处理半包消息的解码类；ProtobufDecoder(UserProto.User.getDefaultInstance())这是创建的UserProto.java文件中的解码类；ProtobufVarint32LengthFieldPrepender 对protobuf协议的消息头上加上一个长度为32的整形字段，用于标志这个消息的长度的类；ProtobufEncoder 是编码类

将StringBuilder转换为ByteBuf类型：copiedBuffer()方法

### Netty 支持哪些心跳类型设置？

readerIdleTime：为读超时时间（即测试端一定时间内未接受到被测试端消息）。

writerIdleTime：为写超时时间（即测试端一定时间内向被测试端发送消息）。

allIdleTime：所有类型的超时时间。

### Netty 和 Tomcat 的区别？

作用不同：Tomcat 是 Servlet 容器，可以视为 Web 服务器，而 Netty 是异步事件驱动的网络应用程序框架和工具用于简化网络编程，例如TCP和UDP套接字服务器。

协议不同：Tomcat 是基于 http 协议的 Web 服务器，而 Netty 能通过编程自定义各种协议，因为 Netty 本身自己能编码/解码字节流，所有 Netty 可以实现，HTTP 服务器、FTP 服务器、UDP 服务器、RPC 服务器、WebSocket 服务器、Redis 的 Proxy 服务器、MySQL 的 Proxy 服务器等等。

### NIOEventLoopGroup源码？

NioEventLoopGroup(其实是MultithreadEventExecutorGroup) 内部维护一个类型为 EventExecutor children [], 默认大小是处理器核数 \* 2, 这样就构成了一个线程池，初始化EventExecutor时NioEventLoopGroup重载newChild方法，所以children元素的实际类型为NioEventLoop。

线程启动时调用SingleThreadEventExecutor的构造方法，执行NioEventLoop类的run方法，首先会调用hasTasks()方法判断当前taskQueue是否有元素。如果taskQueue中有元素，执行 selectNow() 方法，最终执行selector.selectNow()，该方法会立即返回。如果taskQueue没有元素，执行 select(oldWakenUp) 方法

select ( oldWakenUp) 方法解决了 Nio 中的 bug，selectCnt 用来记录selector.select方法的执行次数和标识是否执行过selector.selectNow()，若触发了epoll的空轮询bug，则会反复执行selector.select(timeoutMillis)，变量selectCnt 会逐渐变大，当selectCnt 达到阈值（默认512），则执行rebuildSelector方法，进行selector重建，解决cpu占用100%的bug。

rebuildSelector方法先通过openSelector方法创建一个新的selector。然后将old selector的selectionKey执行cancel。最后将old selector的channel重新注册到新的selector中。rebuild后，需要重新执行方法selectNow，检查是否有已ready的selectionKey。

接下来调用processSelectedKeys 方法（处理I/O任务），当selectedKeys != null时，调用processSelectedKeysOptimized方法，迭代 selectedKeys 获取就绪的 IO 事件的selectkey存放在数组selectedKeys中, 然后为每个事件都调用 processSelectedKey 来处理它，processSelectedKey 中分别处理OP\_READ；OP\_WRITE；OP\_CONNECT事件。

最后调用runAllTasks方法（非IO任务），该方法首先会调用fetchFromScheduledTaskQueue方法，把scheduledTaskQueue中已经超过延迟执行时间的任务移到taskQueue中等待被执行，然后依次从taskQueue中取任务执行，每执行64个任务，进行耗时检查，如果已执行时间超过预先设定的执行时间，则停止执行非IO任务，避免非IO任务太多，影响IO任务的执行。

每个NioEventLoop对应一个线程和一个Selector，NioServerSocketChannel会主动注册到某一个NioEventLoop的Selector上，NioEventLoop负责事件轮询。

Outbound 事件都是请求事件, 发起者是 Channel，处理者是 unsafe，通过 Outbound 事件进行通知，传播方向是 tail到head。Inbound 事件发起者是 unsafe，事件的处理者是 Channel, 是通知事件，传播方向是从头到尾。

内存管理机制，首先会预申请一大块内存Arena，Arena由许多Chunk组成，而每个Chunk默认由2048个page组成。Chunk通过AVL树的形式组织Page，每个叶子节点表示一个Page，而中间节点表示内存区域，节点自己记录它在整个Arena中的偏移地址。当区域被分配出去后，中间节点上的标记位会被标记，这样就表示这个中间节点以下的所有节点都已被分配了。大于8k的内存分配在poolChunkList中，而PoolSubpage用于分配小于8k的内存，它会把一个page分割成多段，进行内存分配。

ByteBuf的特点：支持自动扩容（4M），保证put方法不会抛出异常、通过内置的复合缓冲类型，实现零拷贝（zero-copy）；不需要调用flip()来切换读/写模式，读取和写入索引分开；方法链；引用计数基于AtomicIntegerFieldUpdater用于内存回收；PooledByteBuf采用二叉树来实现一个内存池，集中管理内存的分配和释放，不用每次使用都新建一个缓冲区对象。UnpooledHeapByteBuf每次都会新建一个缓冲区对象。

### Netty简介

Netty是 一个异步事件驱动的网络应用程序框架，用于快速开发可维护的高性能协议服务器和客户端。

JDK原生NIO程序的问题

JDK原生也有一套网络应用程序API，但是存在一系列问题，主要如下：

NIO的类库和API繁杂，使用麻烦，你需要熟练掌握Selector、ServerSocketChannel、SocketChannel、ByteBuffer等

需要具备其它的额外技能做铺垫，例如熟悉Java多线程编程，因为NIO编程涉及到Reactor模式，你必须对多线程和网路编程非常熟悉，才能编写出高质量的NIO程序

可靠性能力补齐，开发工作量和难度都非常大。例如客户端面临断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流的处理等等，NIO编程的特点是功能开发相对容易，但是可靠性能力补齐工作量和难度都非常大

JDK NIO的BUG，例如臭名昭著的epoll bug，它会导致Selector空轮询，最终导致CPU 100%。官方声称在JDK1.6版本的update18修复了该问题，但是直到JDK1.7版本该问题仍旧存在，只不过该bug发生概率降低了一些而已，它并没有被根本解决

Netty的特点

Netty的对JDK自带的NIO的API进行封装，解决上述问题，主要特点有：

设计优雅 适用于各种传输类型的统一API - 阻塞和非阻塞Socket 基于灵活且可扩展的事件模型，可以清晰地分离关注点 高度可定制的线程模型 - 单线程，一个或多个线程池 真正的无连接数据报套接字支持（自3.1起）

使用方便 详细记录的Javadoc，用户指南和示例 没有其他依赖项，JDK 5（Netty 3.x）或6（Netty 4.x）就足够了

高性能 吞吐量更高，延迟更低 减少资源消耗 最小化不必要的内存复制

安全 完整的SSL / TLS和StartTLS支持

社区活跃，不断更新 社区活跃，版本迭代周期短，发现的BUG可以被及时修复，同时，更多的新功能会被加入

Netty常见使用场景

Netty常见的使用场景如下：

互联网行业 在分布式系统中，各个节点之间需要远程服务调用，高性能的RPC框架必不可少，Netty作为异步高新能的通信框架,往往作为基础通信组件被这些RPC框架使用。 典型的应用有：阿里分布式服务框架Dubbo的RPC框架使用Dubbo协议进行节点间通信，Dubbo协议默认使用Netty作为基础通信组件，用于实现各进程节点之间的内部通信。

游戏行业 无论是手游服务端还是大型的网络游戏，Java语言得到了越来越广泛的应用。Netty作为高性能的基础通信组件，它本身提供了TCP/UDP和HTTP协议栈。 非常方便定制和开发私有协议栈，账号登录服务器，地图服务器之间可以方便的通过Netty进行高性能的通信

大数据领域 经典的Hadoop的高性能通信和序列化组件Avro的RPC框架，默认采用Netty进行跨界点通信，它的Netty Service基于Netty框架二次封装实现

有兴趣的读者可以了解一下目前有哪些开源项目使用了 Netty：Related projects

### Netty高性能设计

Netty作为异步事件驱动的网络，高性能之处主要来自于其I/O模型和线程处理模型，前者决定如何收发数据，后者决定如何处理数据

I/O模型

用什么样的通道将数据发送给对方，BIO、NIO或者AIO，I/O模型在很大程度上决定了框架的性能

阻塞I/O

传统阻塞型I/O(BIO)可以用下图表示：

Blocking I/O

特点

每个请求都需要独立的线程完成数据read，业务处理，数据write的完整操作

问题

当并发数较大时，需要创建大量线程来处理连接，系统资源占用较大

连接建立后，如果当前线程暂时没有数据可读，则线程就阻塞在read操作上，造成线程资源浪费

I/O复用模型

img

在I/O复用模型中，会用到select，这个函数也会使进程阻塞，但是和阻塞I/O所不同的的，这两个函数可以同时阻塞多个I/O操作，而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写时，才真正调用I/O操作函数

Netty的非阻塞I/O的实现关键是基于I/O复用模型，这里用Selector对象表示：

Nonblocking I/O

Netty的IO线程NioEventLoop由于聚合了多路复用器Selector，可以同时并发处理成百上千个客户端连接。当线程从某客户端Socket通道进行读写数据时，若没有数据可用时，该线程可以进行其他任务。线程通常将非阻塞 IO 的空闲时间用于在其他通道上执行 IO 操作，所以单独的线程可以管理多个输入和输出通道。

由于读写操作都是非阻塞的，这就可以充分提升IO线程的运行效率，避免由于频繁I/O阻塞导致的线程挂起，一个I/O线程可以并发处理N个客户端连接和读写操作，这从根本上解决了传统同步阻塞I/O一连接一线程模型，架构的性能、弹性伸缩能力和可靠性都得到了极大的提升。

基于buffer

传统的I/O是面向字节流或字符流的，以流式的方式顺序地从一个Stream 中读取一个或多个字节, 因此也就不能随意改变读取指针的位置。

在NIO中, 抛弃了传统的 I/O流, 而是引入了Channel和Buffer的概念. 在NIO中, 只能从Channel中读取数据到Buffer中或将数据 Buffer 中写入到 Channel。

基于buffer操作不像传统IO的顺序操作, NIO 中可以随意地读取任意位置的数据

线程模型

数据报如何读取？读取之后的编解码在哪个线程进行，编解码后的消息如何派发，线程模型的不同，对性能的影响也非常大。

事件驱动模型

通常，我们设计一个事件处理模型的程序有两种思路

轮询方式 线程不断轮询访问相关事件发生源有没有发生事件，有发生事件就调用事件处理逻辑。

事件驱动方式 发生事件，主线程把事件放入事件队列，在另外线程不断循环消费事件列表中的事件，调用事件对应的处理逻辑处理事件。事件驱动方式也被称为消息通知方式，其实是设计模式中观察者模式的思路。

以GUI的逻辑处理为例，说明两种逻辑的不同：

轮询方式 线程不断轮询是否发生按钮点击事件，如果发生，调用处理逻辑

事件驱动方式 发生点击事件把事件放入事件队列，在另外线程消费的事件列表中的事件，根据事件类型调用相关事件处理逻辑

这里借用O’Reilly 大神关于事件驱动模型解释图

事件驱动模型

主要包括4个基本组件：

事件队列（event queue）：接收事件的入口，存储待处理事件

分发器（event mediator）：将不同的事件分发到不同的业务逻辑单元

事件通道（event channel）：分发器与处理器之间的联系渠道

事件处理器（event processor）：实现业务逻辑，处理完成后会发出事件，触发下一步操作

可以看出，相对传统轮询模式，事件驱动有如下优点：

可扩展性好，分布式的异步架构，事件处理器之间高度解耦，可以方便扩展事件处理逻辑

高性能，基于队列暂存事件，能方便并行异步处理事件

Reactor线程模型

Reactor是反应堆的意思，Reactor模型，是指通过一个或多个输入同时传递给服务处理器的服务请求的事件驱动处理模式。 服务端程序处理传入多路请求，并将它们同步分派给请求对应的处理线程，Reactor模式也叫Dispatcher模式，即I/O多了复用统一监听事件，收到事件后分发(Dispatch给某进程)，是编写高性能网络服务器的必备技术之一。

Reactor模型中有2个关键组成：

Reactor Reactor在一个单独的线程中运行，负责监听和分发事件，分发给适当的处理程序来对IO事件做出反应。 它就像公司的电话接线员，它接听来自客户的电话并将线路转移到适当的联系人

Handlers 处理程序执行I/O事件要完成的实际事件，类似于客户想要与之交谈的公司中的实际官员。Reactor通过调度适当的处理程序来响应I/O事件，处理程序执行非阻塞操作

Reactor模型

取决于Reactor的数量和Hanndler线程数量的不同，Reactor模型有3个变种

单Reactor单线程

单Reactor多线程

主从Reactor多线程

可以这样理解，Reactor就是一个执行while (true) { selector.select(); …}循环的线程，会源源不断的产生新的事件，称作反应堆很贴切。

篇幅关系，这里不再具体展开Reactor特性、优缺点比较，有兴趣的读者可以参考我之前另外一篇文章：《理解高性能网络模型》

Netty线程模型

Netty主要基于主从Reactors多线程模型（如下图）做了一定的修改，其中主从Reactor多线程模型有多个Reactor：MainReactor和SubReactor：

MainReactor负责客户端的连接请求，并将请求转交给SubReactor

SubReactor负责相应通道的IO读写请求

非IO请求（具体逻辑处理）的任务则会直接写入队列，等待worker threads进行处理

这里引用Doug Lee大神的Reactor介绍：Scalable IO in Java里面关于主从Reactor多线程模型的图

主从Rreactor多线程模型

特别说明的是： 虽然Netty的线程模型基于主从Reactor多线程，借用了MainReactor和SubReactor的结构，但是实际实现上，SubReactor和Worker线程在同一个线程池中：

EventLoopGroup bossGroup = new NioEventLoopGroup();

EventLoopGroup workerGroup = new NioEventLoopGroup();

ServerBootstrap server = new ServerBootstrap();

server.group(bossGroup, workerGroup)

.channel(NioServerSocketChannel.class)

上面代码中的bossGroup 和workerGroup是Bootstrap构造方法中传入的两个对象，这两个group均是线程池

bossGroup线程池则只是在bind某个端口后，获得其中一个线程作为MainReactor，专门处理端口的accept事件，每个端口对应一个boss线程

workerGroup线程池会被各个SubReactor和worker线程充分利用

异步处理

异步的概念和同步相对。当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。

Netty中的I/O操作是异步的，包括bind、write、connect等操作会简单的返回一个ChannelFuture，调用者并不能立刻获得结果，通过Future-Listener机制，用户可以方便的主动获取或者通过通知机制获得IO操作结果。

当future对象刚刚创建时，处于非完成状态，调用者可以通过返回的ChannelFuture来获取操作执行的状态，注册监听函数来执行完成后的操，常见有如下操作：

通过isDone方法来判断当前操作是否完成

通过isSuccess方法来判断已完成的当前操作是否成功

通过getCause方法来获取已完成的当前操作失败的原因

通过isCancelled方法来判断已完成的当前操作是否被取消

通过addListener方法来注册监听器，当操作已完成(isDone方法返回完成)，将会通知指定的监听器；如果future对象已完成，则理解通知指定的监听器

例如下面的的代码中绑定端口是异步操作，当绑定操作处理完，将会调用相应的监听器处理逻辑

serverBootstrap.bind(port).addListener(future -> {

if (future.isSuccess()) {

System.out.println(new Date() + ": 端口[" + port + "]绑定成功!");

} else {

System.err.println("端口[" + port + "]绑定失败!");

}

});

相比传统阻塞I/O，执行I/O操作后线程会被阻塞住, 直到操作完成；异步处理的好处是不会造成线程阻塞，线程在I/O操作期间可以执行别的程序，在高并发情形下会更稳定和更高的吞吐量。

### Netty架构设计

前面介绍完Netty相关一些理论介绍，下面从功能特性、模块组件、运作过程来介绍Netty的架构设计

功能特性

Netty功能特性图

传输服务 支持BIO和NIO

容器集成 支持OSGI、JBossMC、Spring、Guice容器

协议支持 HTTP、Protobuf、二进制、文本、WebSocket等一系列常见协议都支持。 还支持通过实行编码解码逻辑来实现自定义协议

Core核心 可扩展事件模型、通用通信API、支持零拷贝的ByteBuf缓冲对象

模块组件

Bootstrap、ServerBootstrap

Bootstrap意思是引导，一个Netty应用通常由一个Bootstrap开始，主要作用是配置整个Netty程序，串联各个组件，Netty中Bootstrap类是客户端程序的启动引导类，ServerBootstrap是服务端启动引导类。

Future、ChannelFuture

正如前面介绍，在Netty中所有的IO操作都是异步的，不能立刻得知消息是否被正确处理，但是可以过一会等它执行完成或者直接注册一个监听，具体的实现就是通过Future和ChannelFutures，他们可以注册一个监听，当操作执行成功或失败时监听会自动触发注册的监听事件。

Channel

Netty网络通信的组件，能够用于执行网络I/O操作。 Channel为用户提供：

当前网络连接的通道的状态（例如是否打开？是否已连接？）

网络连接的配置参数 （例如接收缓冲区大小）

提供异步的网络I/O操作(如建立连接，读写，绑定端口)，异步调用意味着任何I / O调用都将立即返回，并且不保证在调用结束时所请求的I / O操作已完成。调用立即返回一个ChannelFuture实例，通过注册监听器到ChannelFuture上，可以I / O操作成功、失败或取消时回调通知调用方。

支持关联I/O操作与对应的处理程序

不同协议、不同的阻塞类型的连接都有不同的 Channel 类型与之对应，下面是一些常用的 Channel 类型

NioSocketChannel，异步的客户端 TCP Socket 连接

NioServerSocketChannel，异步的服务器端 TCP Socket 连接

NioDatagramChannel，异步的 UDP 连接

NioSctpChannel，异步的客户端 Sctp 连接

NioSctpServerChannel，异步的 Sctp 服务器端连接 这些通道涵盖了 UDP 和 TCP网络 IO以及文件 IO.

Selector

Netty基于Selector对象实现I/O多路复用，通过 Selector, 一个线程可以监听多个连接的Channel事件, 当向一个Selector中注册Channel 后，Selector 内部的机制就可以自动不断地查询(select) 这些注册的Channel是否有已就绪的I/O事件(例如可读, 可写, 网络连接完成等)，这样程序就可以很简单地使用一个线程高效地管理多个 Channel 。

NioEventLoop

NioEventLoop中维护了一个线程和任务队列，支持异步提交执行任务，线程启动时会调用NioEventLoop的run方法，执行I/O任务和非I/O任务：

I/O任务 即selectionKey中ready的事件，如accept、connect、read、write等，由processSelectedKeys方法触发。

非IO任务 添加到taskQueue中的任务，如register0、bind0等任务，由runAllTasks方法触发。

两种任务的执行时间比由变量ioRatio控制，默认为50，则表示允许非IO任务执行的时间与IO任务的执行时间相等。

NioEventLoopGroup

NioEventLoopGroup，主要管理eventLoop的生命周期，可以理解为一个线程池，内部维护了一组线程，每个线程(NioEventLoop)负责处理多个Channel上的事件，而一个Channel只对应于一个线程。

ChannelHandler

ChannelHandler是一个接口，处理I / O事件或拦截I / O操作，并将其转发到其ChannelPipeline(业务处理链)中的下一个处理程序。

ChannelHandler本身并没有提供很多方法，因为这个接口有许多的方法需要实现，方便使用期间，可以继承它的子类：

ChannelInboundHandler用于处理入站I / O事件

ChannelOutboundHandler用于处理出站I / O操作

或者使用以下适配器类：

ChannelInboundHandlerAdapter用于处理入站I / O事件

ChannelOutboundHandlerAdapter用于处理出站I / O操作

ChannelDuplexHandler用于处理入站和出站事件

ChannelHandlerContext

保存Channel相关的所有上下文信息，同时关联一个ChannelHandler对象

ChannelPipline

保存ChannelHandler的List，用于处理或拦截Channel的入站事件和出站操作。 ChannelPipeline实现了一种高级形式的拦截过滤器模式，使用户可以完全控制事件的处理方式，以及Channel中各个的ChannelHandler如何相互交互。

下图引用Netty的Javadoc4.1中ChannelPipline的说明，描述了ChannelPipeline中ChannelHandler通常如何处理I/O事件。 I/O事件由ChannelInboundHandler或ChannelOutboundHandler处理，并通过调用ChannelHandlerContext中定义的事件传播方法（例如ChannelHandlerContext.fireChannelRead（Object）和ChannelOutboundInvoker.write（Object））转发到其最近的处理程序。

入站事件由自下而上方向的入站处理程序处理，如图左侧所示。 入站Handler处理程序通常处理由图底部的I / O线程生成的入站数据。 通常通过实际输入操作（例如SocketChannel.read（ByteBuffer））从远程读取入站数据。

出站事件由上下方向处理，如图右侧所示。 出站Handler处理程序通常会生成或转换出站传输，例如write请求。 I/O线程通常执行实际的输出操作，例如SocketChannel.write（ByteBuffer）。

在 Netty 中每个 Channel 都有且仅有一个 ChannelPipeline 与之对应, 它们的组成关系如下:

img

一个 Channel 包含了一个 ChannelPipeline, 而 ChannelPipeline 中又维护了一个由 ChannelHandlerContext 组成的双向链表, 并且每个 ChannelHandlerContext 中又关联着一个 ChannelHandler。入站事件和出站事件在一个双向链表中，入站事件会从链表head往后传递到最后一个入站的handler，出站事件会从链表tail往前传递到最前一个出站的handler，两种类型的handler互不干扰。

工作原理架构

初始化并启动Netty服务端过程如下：

public static void main(String[] args) {

// 创建mainReactor

NioEventLoopGroup boosGroup = new NioEventLoopGroup();

// 创建工作线程组

NioEventLoopGroup workerGroup = new NioEventLoopGroup();

final ServerBootstrap serverBootstrap = new ServerBootstrap();

serverBootstrap

// 组装NioEventLoopGroup

.group(boosGroup, workerGroup)

// 设置channel类型为NIO类型

.channel(NioServerSocketChannel.class)

// 设置连接配置参数

.option(ChannelOption.SO\_BACKLOG, 1024)

.childOption(ChannelOption.SO\_KEEPALIVE, true)

.childOption(ChannelOption.TCP\_NODELAY, true)

// 配置入站、出站事件handler

.childHandler(new ChannelInitializer<NioSocketChannel>() {

@Override

protected void initChannel(NioSocketChannel ch) {

// 配置入站、出站事件channel

ch.pipeline().addLast(...);

ch.pipeline().addLast(...);

}

});

// 绑定端口

int port = 8080;

serverBootstrap.bind(port).addListener(future -> {

if (future.isSuccess()) {

System.out.println(new Date() + ": 端口[" + port + "]绑定成功!");

} else {

System.err.println("端口[" + port + "]绑定失败!");

}

});

}

基本过程如下：

1 初始化创建2个NioEventLoopGroup，其中boosGroup用于Accetpt连接建立事件并分发请求， workerGroup用于处理I/O读写事件和业务逻辑

2 基于ServerBootstrap(服务端启动引导类)，配置EventLoopGroup、Channel类型，连接参数、配置入站、出站事件handler

3 绑定端口，开始工作

结合上面的介绍的Netty Reactor模型，介绍服务端Netty的工作架构图：

服务端Netty Reactor工作架构图

server端包含1个Boss NioEventLoopGroup和1个Worker NioEventLoopGroup，NioEventLoopGroup相当于1个事件循环组，这个组里包含多个事件循环NioEventLoop，每个NioEventLoop包含1个selector和1个事件循环线程。

每个Boss NioEventLoop循环执行的任务包含3步：

1 轮询accept事件

2 处理accept I/O事件，与Client建立连接，生成NioSocketChannel，并将NioSocketChannel注册到某个Worker NioEventLoop的Selector上 \*3 处理任务队列中的任务，runAllTasks。任务队列中的任务包括用户调用eventloop.execute或schedule执行的任务，或者其它线程提交到该eventloop的任务。

每个Worker NioEventLoop循环执行的任务包含3步：

1 轮询read、write事件；

2 处I/O事件，即read、write事件，在NioSocketChannel可读、可写事件发生时进行处理

3 处理任务队列中的任务，runAllTasks。

其中任务队列中的task有3种典型使用场景

1 用户程序自定义的普通任务

ctx.channel().eventLoop().execute(new Runnable() {

@Override

public void run() {

//...

}

});

2 非当前reactor线程调用channel的各种方法 例如在推送系统的业务线程里面，根据用户的标识，找到对应的channel引用，然后调用write类方法向该用户推送消息，就会进入到这种场景。最终的write会提交到任务队列中后被异步消费。

3 用户自定义定时任务

ctx.channel().eventLoop().schedule(new Runnable() {

@Override

public void run() {

}

}, 60, TimeUnit.SECONDS);

总结

现在稳定推荐使用的主流版本还是Netty4，Netty5 中使用了 ForkJoinPool，增加了代码的复杂度，但是对性能的改善却不明显，所以这个版本不推荐使用，官网也没有提供下载链接。

Netty 入门门槛相对较高，其实是因为这方面的资料较少，并不是因为他有多难，大家其实都可以像搞透 Spring 一样搞透 Netty。在学习之前，建议先理解透整个框架原理结构，运行过程，可以少走很多弯路。

## RabbitMQ

### 为什么使用MQ？MQ的优点

简答

异步处理 - 相比于传统的串行、并行方式，提高了系统吞吐量。

应用解耦 - 系统间通过消息通信，不用关心其他系统的处理。

流量削锋 - 可以通过消息队列长度控制请求量；可以缓解短时间内的高并发请求。

日志处理 - 解决大量日志传输。

消息通讯 - 消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯。比如实现点对点消息队列，或者聊天室等。

详答

主要是：解耦、异步、削峰。

解耦：A 系统发送数据到 BCD 三个系统，通过接口调用发送。如果 E 系统也要这个数据呢？那如果 C 系统现在不需要了呢？A 系统负责人几乎崩溃…A 系统跟其它各种乱七八糟的系统严重耦合，A 系统产生一条比较关键的数据，很多系统都需要 A 系统将这个数据发送过来。如果使用 MQ，A 系统产生一条数据，发送到 MQ 里面去，哪个系统需要数据自己去 MQ 里面消费。如果新系统需要数据，直接从 MQ 里消费即可；如果某个系统不需要这条数据了，就取消对 MQ 消息的消费即可。这样下来，A 系统压根儿不需要去考虑要给谁发送数据，不需要维护这个代码，也不需要考虑人家是否调用成功、失败超时等情况。

就是一个系统或者一个模块，调用了多个系统或者模块，互相之间的调用很复杂，维护起来很麻烦。但是其实这个调用是不需要直接同步调用接口的，如果用 MQ 给它异步化解耦。

异步：A 系统接收一个请求，需要在自己本地写库，还需要在 BCD 三个系统写库，自己本地写库要 3ms，BCD 三个系统分别写库要 300ms、450ms、200ms。最终请求总延时是 3 + 300 + 450 + 200 = 953ms，接近 1s，用户感觉搞个什么东西，慢死了慢死了。用户通过浏览器发起请求。如果使用 MQ，那么 A 系统连续发送 3 条消息到 MQ 队列中，假如耗时 5ms，A 系统从接受一个请求到返回响应给用户，总时长是 3 + 5 = 8ms。

削峰：减少高峰时期对服务器压力。

### 消息队列有什么优缺点？RabbitMQ有什么优缺点？

优点上面已经说了，就是在特殊场景下有其对应的好处，解耦、异步、削峰。

缺点有以下几个：

系统可用性降低

本来系统运行好好的，现在你非要加入个消息队列进去，那消息队列挂了，你的系统不是呵呵了。因此，系统可用性会降低；

系统复杂度提高

加入了消息队列，要多考虑很多方面的问题，比如：一致性问题、如何保证消息不被重复消费、如何保证消息可靠性传输等。因此，需要考虑的东西更多，复杂性增大。

一致性问题

A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，咋整？你这数据就不一致了。

所以消息队列实际是一种非常复杂的架构，你引入它有很多好处，但是也得针对它带来的坏处做各种额外的技术方案和架构来规避掉，做好之后，你会发现，妈呀，系统复杂度提升了一个数量级，也许是复杂了 10 倍。但是关键时刻，用，还是得用的。

### 你们公司生产环境用的是什么消息中间件？

这个首先你可以说下你们公司选用的是什么消息中间件，比如用的是RabbitMQ，然后可以初步给一些你对不同MQ中间件技术的选型分析。

举个例子：比如说ActiveMQ是老牌的消息中间件，国内很多公司过去运用的还是非常广泛的，功能很强大。

但是问题在于没法确认ActiveMQ可以支撑互联网公司的高并发、高负载以及高吞吐的复杂场景，在国内互联网公司落地较少。而且使用较多的是一些传统企业，用ActiveMQ做异步调用和系统解耦。

然后你可以说说RabbitMQ，他的好处在于可以支撑高并发、高吞吐、性能很高，同时有非常完善便捷的后台管理界面可以使用。

另外，他还支持集群化、高可用部署架构、消息高可靠支持，功能较为完善。

而且经过调研，国内各大互联网公司落地大规模RabbitMQ集群支撑自身业务的case较多，国内各种中小型互联网公司使用RabbitMQ的实践也比较多。

除此之外，RabbitMQ的开源社区很活跃，较高频率的迭代版本，来修复发现的bug以及进行各种优化，因此综合考虑过后，公司采取了RabbitMQ。

但是RabbitMQ也有一点缺陷，就是他自身是基于erlang语言开发的，所以导致较为难以分析里面的源码，也较难进行深层次的源码定制和改造，毕竟需要较为扎实的erlang语言功底才可以。

然后可以聊聊RocketMQ，是阿里开源的，经过阿里的生产环境的超高并发、高吞吐的考验，性能卓越，同时还支持分布式事务等特殊场景。

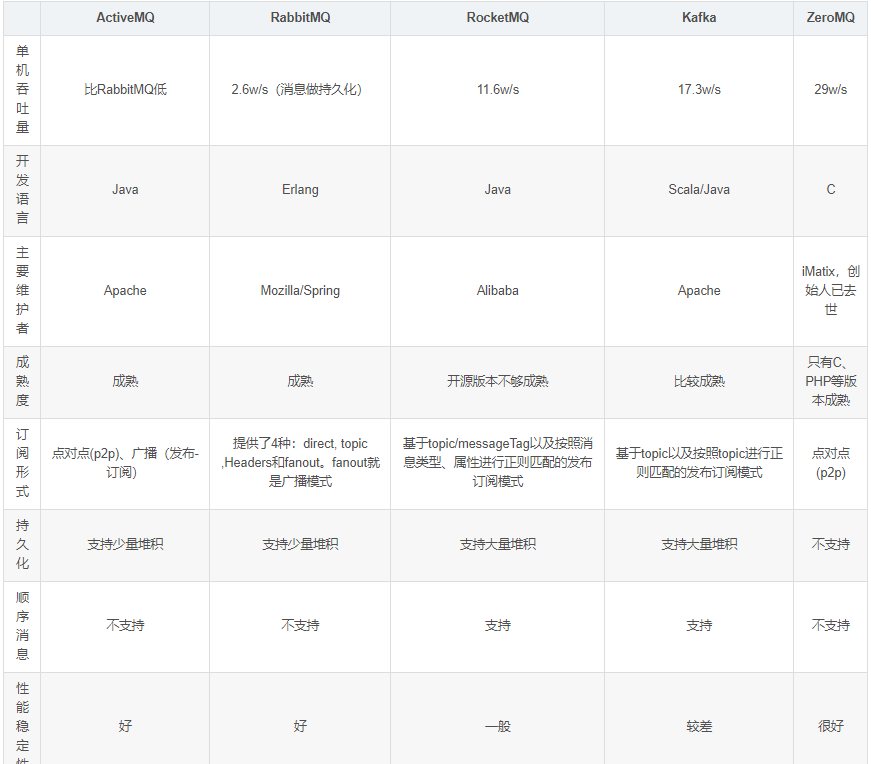
而且RocketMQ是基于Java语言开发的，适合深入阅读源码，有需要可以站在源码层面解决线上生产问题，包括源码的二次开发和改造。

另外就是Kafka。Kafka提供的消息中间件的功能明显较少一些，相对上述几款MQ中间件要少很多。

但是Kafka的优势在于专为超高吞吐量的实时日志采集、实时数据同步、实时数据计算等场景来设计。

因此Kafka在大数据领域中配合实时计算技术（比如Spark Streaming、Storm、Flink）使用的较多。但是在传统的MQ中间件使用场景中较少采用。

### Kafka、ActiveMQ、RabbitMQ、RocketMQ 有什么优缺点？



综上，各种对比之后，有如下建议：

一般的业务系统要引入 MQ，最早大家都用 ActiveMQ，但是现在确实大家用的不多了，没经过大规模吞吐量场景的验证，社区也不是很活跃，所以大家还是算了吧，我个人不推荐用这个了；

后来大家开始用 RabbitMQ，但是确实 erlang 语言阻止了大量的 Java 工程师去深入研究和掌控它，对公司而言，几乎处于不可控的状态，但是确实人家是开源的，比较稳定的支持，活跃度也高；

不过现在确实越来越多的公司会去用 RocketMQ，确实很不错，毕竟是阿里出品，但社区可能有突然黄掉的风险（目前 RocketMQ 已捐给 Apache，但 GitHub 上的活跃度其实不算高）对自己公司技术实力有绝对自信的，推荐用 RocketMQ，否则回去老老实实用 RabbitMQ 吧，人家有活跃的开源社区，绝对不会黄。

所以中小型公司，技术实力较为一般，技术挑战不是特别高，用 RabbitMQ 是不错的选择；大型公司，基础架构研发实力较强，用 RocketMQ 是很好的选择。

如果是大数据领域的实时计算、日志采集等场景，用 Kafka 是业内标准的，绝对没问题，社区活跃度很高，绝对不会黄，何况几乎是全世界这个领域的事实性规范。

### MQ 有哪些常见问题？如何解决这些问题？

MQ 的常见问题有：

消息的顺序问题

消息的重复问题

消息的顺序问题

消息有序指的是可以按照消息的发送顺序来消费。

假如生产者产生了 2 条消息：M1、M2，假定 M1 发送到 S1，M2 发送到 S2，如果要保证 M1 先于 M2 被消费，怎么做？

img

解决方案：

（1）保证生产者 - MQServer - 消费者是一对一对一的关系

img

缺陷：

并行度就会成为消息系统的瓶颈（吞吐量不够）

更多的异常处理，比如：只要消费端出现问题，就会导致整个处理流程阻塞，我们不得不花费更多的精力来解决阻塞的问题。 （2）通过合理的设计或者将问题分解来规避。

不关注乱序的应用实际大量存在

队列无序并不意味着消息无序 所以从业务层面来保证消息的顺序而不仅仅是依赖于消息系统，是一种更合理的方式。

消息的重复问题

造成消息重复的根本原因是：网络不可达。

所以解决这个问题的办法就是绕过这个问题。那么问题就变成了：如果消费端收到两条一样的消息，应该怎样处理？

消费端处理消息的业务逻辑保持幂等性。只要保持幂等性，不管来多少条重复消息，最后处理的结果都一样。保证每条消息都有唯一编号且保证消息处理成功与去重表的日志同时出现。利用一张日志表来记录已经处理成功的消息的 ID，如果新到的消息 ID 已经在日志表中，那么就不再处理这条消息。

### 什么是RabbitMQ？

RabbitMQ是一款开源的，Erlang编写的，基于AMQP协议的消息中间件

rabbitmq 的使用场景

（1）服务间异步通信

（2）顺序消费

（3）定时任务

（4）请求削峰

### RabbitMQ基本概念

Broker： 简单来说就是消息队列服务器实体

Exchange： 消息交换机，它指定消息按什么规则，路由到哪个队列

Queue： 消息队列载体，每个消息都会被投入到一个或多个队列

Binding： 绑定，它的作用就是把exchange和queue按照路由规则绑定起来

Routing Key： 路由关键字，exchange根据这个关键字进行消息投递

VHost： vhost 可以理解为虚拟 broker ，即 mini-RabbitMQ server。其内部均含有独立的 queue、exchange 和 binding 等，但最最重要的是，其拥有独立的权限系统，可以做到 vhost 范围的用户控制。当然，从 RabbitMQ 的全局角度，vhost 可以作为不同权限隔离的手段（一个典型的例子就是不同的应用可以跑在不同的 vhost 中）。

Producer： 消息生产者，就是投递消息的程序

Consumer： 消息消费者，就是接受消息的程序

Channel： 消息通道，在客户端的每个连接里，可建立多个channel，每个channel代表一个会话任务

由Exchange、Queue、RoutingKey三个才能决定一个从Exchange到Queue的唯一的线路。

### RabbitMQ的工作模式

一.simple模式（即最简单的收发模式）

1.消息产生消息，将消息放入队列

2.消息的消费者(consumer) 监听 消息队列,如果队列中有消息,就消费掉,消息被拿走后,自动从队列中删除(隐患 消息可能没有被消费者正确处理,已经从队列中消失了,造成消息的丢失，这里可以设置成手动的ack,但如果设置成手动ack，处理完后要及时发送ack消息给队列，否则会造成内存溢出)。

二.work工作模式(资源的竞争)

1.消息产生者将消息放入队列消费者可以有多个,消费者1,消费者2同时监听同一个队列,消息被消费。C1 C2共同争抢当前的消息队列内容,谁先拿到谁负责消费消息(隐患：高并发情况下,默认会产生某一个消息被多个消费者共同使用,可以设置一个开关(syncronize) 保证一条消息只能被一个消费者使用)。

三.publish/subscribe发布订阅(共享资源)

1、每个消费者监听自己的队列；

2、生产者将消息发给broker，由交换机将消息转发到绑定此交换机的每个队列，每个绑定交换机的队列都将接收到消息。

四.routing路由模式

1.消息生产者将消息发送给交换机按照路由判断,路由是字符串(info) 当前产生的消息携带路由字符(对象的方法),交换机根据路由的key,只能匹配上路由key对应的消息队列,对应的消费者才能消费消息;

2.根据业务功能定义路由字符串

3.从系统的代码逻辑中获取对应的功能字符串,将消息任务扔到对应的队列中。

4.业务场景:error 通知;EXCEPTION;错误通知的功能;传统意义的错误通知;客户通知;利用key路由,可以将程序中的错误封装成消息传入到消息队列中,开发者可以自定义消费者,实时接收错误;

五.topic 主题模式(路由模式的一种)

1.星号井号代表通配符

2.星号代表多个单词,井号代表一个单词

3.路由功能添加模糊匹配

4.消息产生者产生消息,把消息交给交换机

5.交换机根据key的规则模糊匹配到对应的队列,由队列的监听消费者接收消息消费

（在我的理解看来就是routing查询的一种模糊匹配，就类似sql的模糊查询方式）

### 如何保证RabbitMQ消息的顺序性？

拆分多个 queue，每个 queue 一个 consumer，就是多一些 queue 而已，确实是麻烦点；或者就一个 queue 但是对应一个 consumer，然后这个 consumer 内部用内存队列做排队，然后分发给底层不同的 worker 来处理。

### 消息如何分发？

若该队列至少有一个消费者订阅，消息将以循环（round-robin）的方式发送给消费者。每条消息只会分发给一个订阅的消费者（前提是消费者能够正常处理消息并进行确认）。通过路由可实现多消费的功能

### 消息怎么路由？

消息提供方->路由->一至多个队列消息发布到交换器时，消息将拥有一个路由键（routing key），在消息创建时设定。通过队列路由键，可以把队列绑定到交换器上。消息到达交换器后，RabbitMQ 会将消息的路由键与队列的路由键进行匹配（针对不同的交换器有不同的路由规则）；

常用的交换器主要分为一下三种：

fanout：如果交换器收到消息，将会广播到所有绑定的队列上

direct：如果路由键完全匹配，消息就被投递到相应的队列

topic：可以使来自不同源头的消息能够到达同一个队列。 使用 topic 交换器时，可以使用通配符

### 消息基于什么传输？

由于 TCP 连接的创建和销毁开销较大，且并发数受系统资源限制，会造成性能瓶颈。RabbitMQ 使用信道的方式来传输数据。信道是建立在真实的 TCP 连接内的虚拟连接，且每条 TCP 连接上的信道数量没有限制。

### 如何保证消息不被重复消费

先说为什么会重复消费：正常情况下，消费者在消费消息的时候，消费完毕后，会发送一个确认消息给消息队列，消息队列就知道该消息被消费了，就会将该消息从消息队列中删除；

但是因为网络传输等等故障，确认信息没有传送到消息队列，导致消息队列不知道自己已经消费过该消息了，再次将消息分发给其他的消费者。

针对以上问题，一个解决思路是：保证消息的唯一性，就算是多次传输，不要让消息的多次消费带来影响；保证消息等幂性；

比如：在写入消息队列的数据做唯一标示，消费消息时，根据唯一标识判断是否消费过；

假设你有个系统，消费一条消息就往数据库里插入一条数据，要是你一个消息重复两次，你不就插入了两条，这数据不就错了？但是你要是消费到第二次的时候，自己判断一下是否已经消费过了，若是就直接扔了，这样不就保留了一条数据，从而保证了数据的正确性。

### 如何确保消息正确地发送至 RabbitMQ？

发送方确认模式

将信道设置成 confirm 模式（发送方确认模式），则所有在信道上发布的消息都会被指派一个唯一的 ID。

一旦消息被投递到目的队列后，或者消息被写入磁盘后（可持久化的消息），信道会发送一个确认给生产者（包含消息唯一 ID）。

如果 RabbitMQ 发生内部错误从而导致消息丢失，会发送一条 nack（notacknowledged，未确认）消息。

发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者应用程序，生产者应用程序的回调方法就会被触发来处理确认消息。

接收方确认机制

消费者接收每一条消息后都必须进行确认（消息接收和消息确认是两个不同操作）。只有消费者确认了消息，RabbitMQ 才能安全地把消息从队列中删除。

这里并没有用到超时机制，RabbitMQ 仅通过 Consumer 的连接中断来确认是否需要重新发送消息。也就是说，只要连接不中断，RabbitMQ 给了 Consumer 足够长的时间来处理消息。保证数据的最终一致性；

下面罗列几种特殊情况

如果消费者接收到消息，在确认之前断开了连接或取消订阅，RabbitMQ 会认为消息没有被分发，然后重新分发给下一个订阅的消费者。（可能存在消息重复消费的隐患，需要去重）

如果消费者接收到消息却没有确认消息，连接也未断开，则 RabbitMQ 认为该消费者繁忙，将不会给该消费者分发更多的消息。

### 如何保证RabbitMQ消息的可靠传输？

消息不可靠的情况可能是消息丢失，劫持等原因；

丢失又分为：生产者丢失消息、消息列表丢失消息、消费者丢失消息；

生产者丢失消息：从生产者弄丢数据这个角度来看，RabbitMQ提供transaction和confirm模式来确保生产者不丢消息；

transaction机制就是说：发送消息前，开启事务（channel.txSelect()）,然后发送消息，如果发送过程中出现什么异常，事务就会回滚（channel.txRollback()）,如果发送成功则提交事务（channel.txCommit()）。然而，这种方式有个缺点：吞吐量下降；

confirm模式用的居多：一旦channel进入confirm模式，所有在该信道上发布的消息都将会被指派一个唯一的ID（从1开始），一旦消息被投递到所有匹配的队列之后；

rabbitMQ就会发送一个ACK给生产者（包含消息的唯一ID），这就使得生产者知道消息已经正确到达目的队列了；

如果rabbitMQ没能处理该消息，则会发送一个Nack消息给你，你可以进行重试操作。

消息队列丢数据：消息持久化。

处理消息队列丢数据的情况，一般是开启持久化磁盘的配置。

这个持久化配置可以和confirm机制配合使用，你可以在消息持久化磁盘后，再给生产者发送一个Ack信号。

这样，如果消息持久化磁盘之前，rabbitMQ阵亡了，那么生产者收不到Ack信号，生产者会自动重发。

那么如何持久化呢？

这里顺便说一下吧，其实也很容易，就下面两步

将queue的持久化标识durable设置为true,则代表是一个持久的队列

发送消息的时候将deliveryMode=2

这样设置以后，即使rabbitMQ挂了，重启后也能恢复数据

消费者丢失消息：消费者丢数据一般是因为采用了自动确认消息模式，改为手动确认消息即可！

消费者在收到消息之后，处理消息之前，会自动回复RabbitMQ已收到消息；

如果这时处理消息失败，就会丢失该消息；

解决方案：处理消息成功后，手动回复确认消息。

### 为什么不应该对所有的 message 都使用持久化机制？

首先，必然导致性能的下降，因为写磁盘比写 RAM 慢的多，message 的吞吐量可能有 10 倍的差距。

其次，message 的持久化机制用在 RabbitMQ 的内置 cluster 方案时会出现“坑爹”问题。矛盾点在于，若 message 设置了 persistent 属性，但 queue 未设置 durable 属性，那么当该 queue 的 owner node 出现异常后，在未重建该 queue 前，发往该 queue 的 message 将被 blackholed ；若 message 设置了 persistent 属性，同时 queue 也设置了 durable 属性，那么当 queue 的 owner node 异常且无法重启的情况下，则该 queue 无法在其他 node 上重建，只能等待其 owner node 重启后，才能恢复该 queue 的使用，而在这段时间内发送给该 queue 的 message 将被 blackholed 。

所以，是否要对 message 进行持久化，需要综合考虑性能需要，以及可能遇到的问题。若想达到 100,000 条/秒以上的消息吞吐量（单 RabbitMQ 服务器），则要么使用其他的方式来确保 message 的可靠 delivery ，要么使用非常快速的存储系统以支持全持久化（例如使用 SSD）。另外一种处理原则是：仅对关键消息作持久化处理（根据业务重要程度），且应该保证关键消息的量不会导致性能瓶颈。

### 如何保证高可用的？RabbitMQ 的集群

RabbitMQ 是比较有代表性的，因为是基于主从（非分布式）做高可用性的，我们就以 RabbitMQ 为例子讲解第一种 MQ 的高可用性怎么实现。RabbitMQ 有三种模式：单机模式、普通集群模式、镜像集群模式。

单机模式，就是 Demo 级别的，一般就是你本地启动了玩玩儿的?，没人生产用单机模式

普通集群模式，意思就是在多台机器上启动多个 RabbitMQ 实例，每个机器启动一个。你创建的 queue，只会放在一个 RabbitMQ 实例上，但是每个实例都同步 queue 的元数据（元数据可以认为是 queue 的一些配置信息，通过元数据，可以找到 queue 所在实例）。你消费的时候，实际上如果连接到了另外一个实例，那么那个实例会从 queue 所在实例上拉取数据过来。这方案主要是提高吞吐量的，就是说让集群中多个节点来服务某个 queue 的读写操作。

镜像集群模式：这种模式，才是所谓的 RabbitMQ 的高可用模式。跟普通集群模式不一样的是，在镜像集群模式下，你创建的 queue，无论元数据还是 queue 里的消息都会存在于多个实例上，就是说，每个 RabbitMQ 节点都有这个 queue 的一个完整镜像，包含 queue 的全部数据的意思。然后每次你写消息到 queue 的时候，都会自动把消息同步到多个实例的 queue 上。RabbitMQ 有很好的管理控制台，就是在后台新增一个策略，这个策略是镜像集群模式的策略，指定的时候是可以要求数据同步到所有节点的，也可以要求同步到指定数量的节点，再次创建 queue 的时候，应用这个策略，就会自动将数据同步到其他的节点上去了。这样的话，好处在于，你任何一个机器宕机了，没事儿，其它机器（节点）还包含了这个 queue 的完整数据，别的 consumer 都可以到其它节点上去消费数据。坏处在于，第一，这个性能开销也太大了吧，消息需要同步到所有机器上，导致网络带宽压力和消耗很重！RabbitMQ 一个 queue 的数据都是放在一个节点里的，镜像集群下，也是每个节点都放这个 queue 的完整数据。

### 如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决？

消息积压处理办法：临时紧急扩容：

先修复 consumer 的问题，确保其恢复消费速度，然后将现有 cnosumer 都停掉。

新建一个 topic，partition 是原来的 10 倍，临时建立好原先 10 倍的 queue 数量。

然后写一个临时的分发数据的 consumer 程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的 10 倍数量的 queue。

接着临时征用 10 倍的机器来部署 consumer，每一批 consumer 消费一个临时 queue 的数据。这种做法相当于是临时将 queue 资源和 consumer 资源扩大 10 倍，以正常的 10 倍速度来消费数据。

等快速消费完积压数据之后，得恢复原先部署的架构，重新用原先的 consumer 机器来消费消息。

MQ中消息失效：假设你用的是 RabbitMQ，RabbtiMQ 是可以设置过期时间的，也就是 TTL。如果消息在 queue 中积压超过一定的时间就会被 RabbitMQ 给清理掉，这个数据就没了。那这就是第二个坑了。这就不是说数据会大量积压在 mq 里，而是大量的数据会直接搞丢。我们可以采取一个方案，就是批量重导，这个我们之前线上也有类似的场景干过。就是大量积压的时候，我们当时就直接丢弃数据了，然后等过了高峰期以后，比如大家一起喝咖啡熬夜到晚上12点以后，用户都睡觉了。这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入 mq 里面去，把白天丢的数据给他补回来。也只能是这样了。假设 1 万个订单积压在 mq 里面，没有处理，其中 1000 个订单都丢了，你只能手动写程序把那 1000 个订单给查出来，手动发到 mq 里去再补一次。

mq消息队列块满了：如果消息积压在 mq 里，你很长时间都没有处理掉，此时导致 mq 都快写满了，咋办？这个还有别的办法吗？没有，谁让你第一个方案执行的太慢了，你临时写程序，接入数据来消费，消费一个丢弃一个，都不要了，快速消费掉所有的消息。然后走第二个方案，到了晚上再补数据吧。

### 设计MQ思路

比如说这个消息队列系统，我们从以下几个角度来考虑一下：

首先这个 mq 得支持可伸缩性吧，就是需要的时候快速扩容，就可以增加吞吐量和容量，那怎么搞？设计个分布式的系统呗，参照一下 kafka 的设计理念，broker -> topic -> partition，每个 partition 放一个机器，就存一部分数据。如果现在资源不够了，简单啊，给 topic 增加 partition，然后做数据迁移，增加机器，不就可以存放更多数据，提供更高的吞吐量了？

其次你得考虑一下这个 mq 的数据要不要落地磁盘吧？那肯定要了，落磁盘才能保证别进程挂了数据就丢了。那落磁盘的时候怎么落啊？顺序写，这样就没有磁盘随机读写的寻址开销，磁盘顺序读写的性能是很高的，这就是 kafka 的思路。

其次你考虑一下你的 mq 的可用性啊？这个事儿，具体参考之前可用性那个环节讲解的 kafka 的高可用保障机制。多副本 -> leader & follower -> broker 挂了重新选举 leader 即可对外服务。

能不能支持数据 0 丢失啊？可以的，参考我们之前说的那个 kafka 数据零丢失方案。

# WEB

## Cookie

cookie 是一个非常具体的东西，指的就是浏览器里面能永久存储的一种数据，仅仅是浏览器实现的一种数据存储功能。

cookie由服务器生成，发送给浏览器，浏览器把cookie以kv形式保存到某个目录下的文本文件内，下一次请求同一网站时会把该cookie发送给服务器。由于cookie是存在客户端上的，所以浏览器加入了一些限制确保cookie不会被恶意使用，同时不会占据太多磁盘空间，所以每个域的cookie数量是有限的。

## Session

session 从字面上讲，就是会话。这个就类似于你和一个人交谈，你怎么知道当前和你交谈的是张三而不是李四呢？对方肯定有某种特征（长相等）表明他就是张三。

session 也是类似的道理，服务器要知道当前发请求给自己的是谁。为了做这种区分，服务器就要给每个客户端分配不同的“身份标识”，然后客户端每次向服务器发请求的时候，都带上这个“身份标识”，服务器就知道这个请求来自于谁了。至于客户端怎么保存这个“身份标识”，可以有很多种方式，对于浏览器客户端，大家都默认采用 cookie 的方式。

服务器使用session把用户的信息临时保存在了服务器上，用户离开网站后session会被销毁。这种用户信息存储方式相对cookie来说更安全，可是session有一个缺陷：如果web服务器做了负载均衡，那么下一个操作请求到了另一台服务器的时候session会丢失。

## Token

### 几点特性

会让你在程序中使用基于Token的身份验证

无状态、可扩展

支持移动设备

跨程序调用

安全

那些使用基于Token的身份验证的大佬们

大部分你见到过的API和Web应用都使用tokens。例如Facebook, Twitter, Google+, GitHub等。

### 基于服务器的验证

我们都是知道HTTP协议是无状态的，这种无状态意味着程序需要验证每一次请求，从而辨别客户端的身份。

在这之前，程序都是通过在服务端存储的登录信息来辨别请求的。这种方式一般都是通过存储Session来完成。

随着Web，应用程序，已经移动端的兴起，这种验证的方式逐渐暴露出了问题。尤其是在可扩展性方面。

### 基于服务器验证方式暴露的一些问题

Seesion：每次认证用户发起请求时，服务器需要去创建一个记录来存储信息。当越来越多的用户发请求时，内存的开销也会不断增加。

可扩展性：在服务端的内存中使用Seesion存储登录信息，伴随而来的是可扩展性问题。

CORS(跨域资源共享)：当我们需要让数据跨多台移动设备上使用时，跨域资源的共享会是一个让人头疼的问题。在使用Ajax抓取另一个域的资源，就可以会出现禁止请求的情况。

CSRF(跨站请求伪造)：用户在访问银行网站时，他们很容易受到跨站请求伪造的攻击，并且能够被利用其访问其他的网站。

在这些问题中，可扩展行是最突出的。因此我们有必要去寻求一种更有行之有效的方法。

### 基于Token的验证原理

基于Token的身份验证是无状态的，我们不将用户信息存在服务器或Session中。

这种概念解决了在服务端存储信息时的许多问题

NoSession意味着你的程序可以根据需要去增减机器，而不用去担心用户是否登录。

### 基于Token的身份验证的过程如下:

用户通过用户名和密码发送请求。

程序验证。

程序返回一个签名的token 给客户端。

客户端储存token,并且每次用于每次发送请求。

服务端验证token并返回数据。

## JWT

### JSON Web Token是什么

JSON Web Token (JWT)是一个开放标准(RFC 7519)，它定义了一种紧凑的、自包含的方式，用于作为JSON对象在各方之间安全地传输信息。该信息可以被验证和信任，因为它是数字签名的。

### JSON Web Tokens是如何工作的

在认证的时候，当用户用他们的凭证成功登录以后，一个JSON Web Token将会被返回。此后，token就是用户凭证了，你必须非常小心以防止出现安全问题。一般而言，你保存令牌的时候不应该超过你所需要它的时间。

无论何时用户想要访问受保护的路由或者资源的时候，用户代理（通常是浏览器）都应该带上JWT，典型的，通常放在Authorization header中，用Bearer schema。

header应该看起来是这样的：

Authorization: Bearer

服务器上的受保护的路由将会检查Authorization header中的JWT是否有效，如果有效，则用户可以访问受保护的资源。如果JWT包含足够多的必需的数据，那么就可以减少对某些操作的数据库查询的需要，尽管可能并不总是如此。

如果token是在授权头（Authorization header）中发送的，那么跨源资源共享(CORS)将不会成为问题，因为它不使用cookie。

## Serlvet

Servlet其实就是一个遵循Servlet开发的java类。Serlvet是由服务器调用的，运行在服务器端。

### 为什么要用到Serlvet？

我们编写java程序想要在网上实现 聊天、发帖、这样一些的交互功能，普通的java技术是非常难完成的。sun公司就提供了Serlvet这种技术供我们使用。

### Serlvet生命周期

第一次访问Servlet，我们发现init()和service()都被调用了

第二次访问Servlet，service()被调用了

当我们关闭Tomcat服务器的时候，destroy()被调用了！

加载Servlet。当Tomcat第一次访问Servlet的时候，Tomcat会负责创建Servlet的实例

初始化。当Servlet被实例化后，Tomcat会调用init()方法初始化这个对象

处理服务。当浏览器访问Servlet的时候，Servlet 会调用service()方法处理请求

销毁。当Tomcat关闭时或者检测到Servlet要从Tomcat删除的时候会自动调用destroy()方法，让该实例释放掉所占的资源。一个Servlet如果长时间不被使用的话，也会被Tomcat自动销毁

卸载。当Servlet调用完destroy()方法后，等待垃圾回收。如果有需要再次使用这个Servlet，会重新调用init()方法进行初始化操作。

## Spring 过滤器

过滤器 Filter，是在 Servlet 规范中定义的，是 Servlet 容器支持的，该接口定义在 javax.servlet包下，主要是在客户端请求(HttpServletRequest)进行预处理，以及对服务器响应(HttpServletResponse)进行后处理。

对上面三个接口方法进行分析:

init(FilterConfig): 初始化接口，在用户自定义的 Filter 初始化时被调用，它与 Servlet 的 init 方法的作用是一样的。

doFilter(ServletRequest,ServletResponse,FilterChain): 在每个用户的请求进来时这个方法都会被调用，并在 Servlet 的 service 方法之前调用(如果我们是开发 Servlet 项目)，而 FilterChain 就代表当前的整个请求链，通过调用 FilterChain.doFilter可以将请求继续传递下去，如果想拦截这个请求，可以不调用 FilterChain.doFilter，那么这个请求就直接返回了，所以 Filter 是一种责任链设计模式，在spring security就大量使用了过滤器，有一条过滤器链。

destroy: 当 Filter 对象被销毁时，这个方法被调用，注意，当 Web 容器调用这个方法之后，容器会再调用一次 doFilter 方法。

### 如何实现自己的过滤器

首先我们需要创建一个类，让它实现 Filter 接口，然后重写接口中的方法：

当我们配置了多个 filter，且一个请求能够被多次拦截时，该请求将沿着 客户端 -> 过滤器1 -> 过滤器2 -> servlet -> 过滤器2 -> 过滤器1 -> 客户端 链式流转

启动项目，可以看到我们的过滤器已经随着程序的启动被成功初始化了。

分别对这两个接口发送请求, 看到结果:

可以看出，当请求同时满足多个过滤器的过滤条件时，filterChain.doFilter() 会将其按一定顺序（可以通过 @Order 指定）依次传递到下一个 filter，直到进入 servlet 进行接口的实际调用。调用完成后，响应结果将沿着原路返回，并在再一次经过各个 filter 后，最终抵达客户端。

## 拦截器

### 什么是拦截器

拦截器是 AOP 的一种实现策略，用于在某个方法或字段被访问前对它进行拦截，然后在其之前或之后加上某些操作。同 filter 一样，interceptor 也是链式调用。每个 interceptor 的调用会依据它的声明顺序依次执行。一般来说拦截器可以用于以下方面 ：

日志记录 ：几率请求信息的日志，以便进行信息监控、信息统计等等

权限检查 ：对用户的访问权限，认证，或授权等进行检查

性能监控 ：通过拦截器在进入处理器前后分别记录开始时间和结束时间，从而得到请求的处理时间

通用行为 ：读取 cookie 得到用户信息并将用户对象放入请求头中，从而方便后续流程使用

拦截器的接口方法

在 SpringMVC 中，DispatcherServlet 捕获每个请求，在到达对应的 Controller 之前，请求可以被拦截器处理，在拦截器中进行前置处理后，请求最终才到达 Controller。

拦截器的接口是 org.springframework.web.servlet.HandlerInterceptor接口，

接口方法解读:

preHandle 方法：对客户端发过来的请求进行前置处理，如果方法返回 true,继续执行后续操作，如果返回 false，执行中断请求处理，请求不会发送到 Controller

postHandler 方法：在请求进行处理后执行，也就是在 Controller 方法调用之后处理，当然前提是之前的 preHandle方法返回 true。具体来说，postHandler方法会在 DispatcherServlet 进行视图返回渲染前被调用，也就是说我们可以在这个方法中对 Controller 处理之后的ModelAndView对象进行操作

afterCompletion 方法: 该方法在整个请求结束之后执行，当然前提依然是 preHandle方法的返回值为 true 才行。该方法一般用于资源清理工作

如何实现一个自己的拦截器

同样，首先创建一个类，让它实现 HandlerInterceptor 接口，然后重写接口中的方法 ：

紧接着需要对拦截器进行注册，指明使用哪个拦截器，及该拦截器对应拦截的 URL ：

### 其他注意事项

在 Http 的请求执行过程中，要经过以下几个步骤  ：

由 DispatcherServlet 捕获请求

DispatcherServlet 将接收到的 URL 和对应的 Controller 进行映射

在请求到达相应的 Controller 之前，由拦截器对请求进行处理

处理完成之后，进行视图的解析

返回视图

所以，只有经过 DispatcherServlet 的请求才会被拦截器捕获，而我们自定义的 Servlet 请求则不会被拦截的。

## 过滤器与拦截器两者对比总结

过滤器是基于函数的回调，而拦截器是基于 Java 反射机制的

过滤器 Filter 依赖于 Servlet 容器。拦截器 Interceptor 依赖于框架容器，可以调用 IOC 容器中的各种依赖

拦截器可以 preHandle方法内返回 false 进行中断。过滤器就比较复杂，需要处理请求和响应对象来引发中断，需要额外的动作，比如将用户重定向到错误页面

过滤器只能在请求的前后使用，而拦截器可以详细到每个方法

tomcat 容器中执行顺序: Filter -> Servlet -> Interceptor -> Controller

## 转发和重定向区别

页面跳转的两种实现方式：请求转发和重定向；

请求转发：

客户首先发送一个请求到服务器端，服务器端发现匹配的servlet，并指定它去执行，当这个servlet执行完之后，它要调用getRequestDispacther()方法，把请求转发给指定的student\_list.jsp,整个流程都是在服务器端完成的，而且是在同一个请求里面完成的，因此servlet和jsp共享的是同一个request，在servlet里面放的所有东西，在student\_list中都能取出来，因此，student\_list能把结果getAttribute()出来，getAttribute()出来后执行完把结果返回给客户端。整个过程是一个请求，一个响应。

重定向：

客户发送一个请求到服务器，服务器匹配servlet，servlet处理完之后调用了sendRedirect()方法，立即向客户端返回这个响应，响应行告诉客户端你必须要再发送一个请求，去访问student\_list.jsp，紧接着客户端收到这个请求后，立刻发出一个新的请求，去请求student\_list.jsp,这里两个请求互不干扰，相互独立，在前面request里面setAttribute()的任何东西，在后面的request里面都获得不了。可见，在sendRedirect()里面是两个请求，两个响应。（服务器向浏览器发送一个302状态码以及一个location消息头，浏览器收到请求后会向再次根据重定向地址发出请求）

请求转发：request.getRequestDispatcher("/test.jsp").forword(request,response);

重定向：response.sendRedirect("/test.jsp");

区别：

1、请求次数：重定向是浏览器向服务器发送一个请求并收到响应后再次向一个新地址发出请求，转发是服务器收到请求后为了完成响应跳转到一个新的地址；重定向至少请求两次，转发请求一次；

2、地址栏不同：重定向地址栏会发生变化，转发地址栏不会发生变化；

3、是否共享数据：重定向两次请求不共享数据，转发一次请求共享数据（在request级别使用信息共享，使用重定向必然出错）；

4、跳转限制：重定向可以跳转到任意URL，转发只能跳转本站点资源；

5、发生行为不同：重定向是客户端行为，转发是服务器端行为；

使用：

1、可以利用request的域对象的特点，由源组件向其中存放写数据；

2、可以让用户访问到存放在WEB-INF目录中的目标资源；

3、重定向的速度比转发慢，因为浏览器还得发出一个新的请求，所以如果在使用转发和重定向都无所谓的时候建议使用转发；

4、因为转发只能访问当前WEB的应用程序，所以不同WEB应用程序之间的访问，特别是要访问到另外一个WEB站点上的资源的情况，这个时候就只能使用重定向了。

## Http

超文本传输协议可以进行文字分割：超文本（Hypertext）、传输（Transfer）、协议（Protocol）

## 什么是超文本

在互联网早期的时候，我们输入的信息只能保存在本地，无法和其他电脑进行交互。我们保存的信息通常都以文本即简单字符的形式存在，文本是一种能够被计算机解析的有意义的二进制数据包。而随着互联网的高速发展，两台电脑之间能够进行数据的传输后，人们不满足只能在两台电脑之间传输文字，还想要传输图片、音频、视频，甚至点击文字或图片能够进行超链接的跳转，那么文本的语义就被扩大了，这种语义扩大后的文本就被称为超文本(Hypertext)。

## 与 HTTP 有关的组件

随着网络世界演进，HTTP 协议已经几乎成为不可替代的一种协议，在了解了 HTTP 的基本组成后，下面再来带你进一步认识一下 HTTP 协议。

### 网络模型

网络是一个复杂的系统，不仅包括大量的应用程序、端系统、通信链路、分组交换机等，还有各种各样的协议组成，那么现在我们就来聊一下网络中的协议层次。

为了给网络协议的设计提供一个结构，网络设计者以分层(layer)的方式组织协议，每个协议属于层次模型之一。每一层都是向它的上一层提供服务(service)，即所谓的服务模型(service model)。每个分层中所有的协议称为 协议栈(protocol stack)。因特网的协议栈由五个部分组成：物理层、链路层、网络层、运输层和应用层。我们采用自上而下的方法研究其原理，也就是应用层 -> 物理层的方式。

### 应用层

应用层是网络应用程序和网络协议存放的分层，因特网的应用层包括许多协议，例如我们学 web 离不开的 HTTP，电子邮件传送协议 SMTP、端系统文件上传协议 FTP、还有为我们进行域名解析的 DNS 协议。应用层协议分布在多个端系统上，一个端系统应用程序与另外一个端系统应用程序交换信息分组，我们把位于应用层的信息分组称为 报文(message)。

### 运输层

因特网的运输层在应用程序断点之间传送应用程序报文，在这一层主要有两种传输协议 TCP和 UDP，利用这两者中的任何一个都能够传输报文，不过这两种协议有巨大的不同。

TCP 向它的应用程序提供了面向连接的服务，它能够控制并确认报文是否到达，并提供了拥塞机制来控制网络传输，因此当网络拥塞时，会抑制其传输速率。

UDP 协议向它的应用程序提供了无连接服务。它不具备可靠性的特征，没有流量控制，也没有拥塞控制。我们把运输层的分组称为 报文段(segment)

### 网络层

因特网的网络层负责将称为 数据报(datagram) 的网络分层从一台主机移动到另一台主机。网络层一个非常重要的协议是 IP 协议，所有具有网络层的因特网组件都必须运行 IP 协议，IP 协议是一种网际协议，除了 IP 协议外，网络层还包括一些其他网际协议和路由选择协议，一般把网络层就称为 IP 层，由此可知 IP 协议的重要性。

### 链路层

现在我们有应用程序通信的协议，有了给应用程序提供运输的协议，还有了用于约定发送位置的 IP 协议，那么如何才能真正的发送数据呢？为了将分组从一个节点（主机或路由器）运输到另一个节点，网络层必须依靠链路层提供服务。链路层的例子包括以太网、WiFi 和电缆接入的 DOCSIS 协议，因为数据从源目的地传送通常需要经过几条链路，一个数据包可能被沿途不同的链路层协议处理，我们把链路层的分组称为 帧(frame)

### 物理层

虽然链路层的作用是将帧从一个端系统运输到另一个端系统，而物理层的作用是将帧中的一个个 比特 从一个节点运输到另一个节点，物理层的协议仍然使用链路层协议，这些协议与实际的物理传输介质有关，例如，以太网有很多物理层协议：关于双绞铜线、关于同轴电缆、关于光纤等等

## 与 HTTP 有关的协议

在互联网中，任何协议都不会单独的完成信息交换，HTTP 也一样。虽然 HTTP 属于应用层的协议，但是它仍然需要其他层次协议的配合完成信息的交换，那么在完成一次 HTTP 请求和响应的过程中，需要哪些协议的配合呢？一起来看一下

### TCP/IP

TCP/IP 协议你一定听过，TCP/IP 我们一般称之为协议簇，什么意思呢？就是 TCP/IP 协议簇中不仅仅只有 TCP 协议和 IP 协议，它是一系列网络通信协议的统称。而其中最核心的两个协议就是 TCP / IP 协议，其他的还有 UDP、ICMP、ARP 等等，共同构成了一个复杂但有层次的协议栈。

TCP 协议的全称是 Transmission Control Protocol 的缩写，意思是传输控制协议，HTTP 使用 TCP 作为通信协议，这是因为 TCP 是一种可靠的协议，而可靠能保证数据不丢失。

IP 协议的全称是 Internet Protocol 的缩写，它主要解决的是通信双方寻址的问题。IP 协议使用 IP 地址 来标识互联网上的每一台计算机，可以把 IP 地址想象成为你手机的电话号码，你要与他人通话必须先要知道他人的手机号码，计算机网络中信息交换必须先要知道对方的 IP 地址。（关于 TCP 和 IP 更多的讨论我们会在后面详解）

### DNS

你有没有想过为什么你可以通过键入 www.google.com 就能够获取你想要的网站？我们上面说到，计算机网络中的每个端系统都有一个 IP 地址存在，而把 IP 地址转换为便于人类记忆的协议就是 DNS 协议。

DNS 的全称是域名系统（Domain Name System，缩写：DNS），它作为将域名和 IP 地址相互映射的一个分布式数据库，能够使人更方便地访问互联网。

### URI / URL

我们上面提到，你可以通过输入 www.google.com 地址来访问谷歌的官网，那么这个地址有什么规定吗？我怎么输都可以？AAA.BBB.CCC 是不是也行？当然不是的，你输入的地址格式必须要满足 URI 的规范。

URI的全称是（Uniform Resource Identifier），中文名称是统一资源标识符，使用它就能够唯一地标记互联网上资源。

URL的全称是（Uniform Resource Locator），中文名称是统一资源定位符，也就是我们俗称的网址，它实际上是 URI 的一个子集。

URI 不仅包括 URL，还包括 URN（统一资源名称），它们之间的关系如下

### HTTPS

HTTP 一般是明文传输，很容易被攻击者窃取重要信息，鉴于此，HTTPS 应运而生。HTTPS 的全称为 （Hyper Text Transfer Protocol over SecureSocket Layer），全称有点长，HTTPS 和 HTTP 有很大的不同在于 HTTPS 是以安全为目标的 HTTP 通道，在 HTTP 的基础上通过传输加密和身份认证保证了传输过程的安全性。HTTPS 在 HTTP 的基础上增加了 SSL 层，也就是说 HTTPS = HTTP + SSL。（这块我们后面也会详谈 HTTPS）

## HTTP 请求响应过程

当我们输入网址并点击回车时，浏览器内部会进行如下操作

DNS服务器会首先进行域名的映射，找到访问www.someSchool.edu所在的地址，然后HTTP 客户端进程在 80 端口发起一个到服务器 www.someSchool.edu 的 TCP 连接（80 端口是 HTTP 的默认端口）。在客户和服务器进程中都会有一个套接字与其相连。

HTTP 客户端通过它的套接字向服务器发送一个 HTTP 请求报文。该报文中包含了路径 someDepartment/home.index 的资源，我们后面会详细讨论 HTTP 请求报文。

HTTP 服务器通过它的套接字接受该报文，进行请求的解析工作，并从其存储器(RAM 或磁盘)中检索出对象 www.someSchool.edu/someDepartment/home.index，然后把检索出来的对象进行封装，封装到 HTTP 响应报文中，并通过套接字向客户进行发送。

HTTP 服务器随即通知 TCP 断开 TCP 连接，实际上是需要等到客户接受完响应报文后才会断开 TCP 连接。

HTTP 客户端接受完响应报文后，TCP 连接会关闭。HTTP 客户端从响应中提取出报文中是一个 HTML 响应文件，并检查该 HTML 文件，然后循环检查报文中其他内部对象。

检查完成后，HTTP 客户端会把对应的资源通过显示器呈现给用户。

至此，键入网址再按下回车的全过程就结束了。上述过程描述的是一种简单的请求-响应全过程，真实的请求-响应情况可能要比上面描述的过程复杂很多。

## HTTP 请求特征

从上面整个过程中我们可以总结出 HTTP 进行分组传输是具有以下特征

支持客户-服务器模式

简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有 GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于 HTTP 协议简单，使得 HTTP 服务器的程序规模小，因而通信速度很快。

灵活：HTTP 允许传输任意类型的数据对象。正在传输的类型由 Content-Type 加以标记。

无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

无状态：HTTP 协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

## 详解 HTTP 报文

### HTTP 协议主要由三大部分组成：

起始行（start line）：描述请求或响应的基本信息；

头部字段（header）：使用 key-value 形式更详细地说明报文；

消息正文（entity）：实际传输的数据，它不一定是纯文本，可以是图片、视频等二进制数据。

其中起始行和头部字段并成为 请求头 或者 响应头，统称为 Header；消息正文也叫做实体，称为 body。HTTP 协议规定每次发送的报文必须要有 Header，但是可以没有 body，也就是说头信息是必须的，实体信息可以没有。

### HTTP 请求方法

HTTP 请求方法一般分为 8 种，它们分别是

GET 获取资源，GET 方法用来请求访问已被 URI 识别的资源。指定的资源经服务器端解析后返回响应内容。也就是说，如果请求的资源是文本，那就保持原样返回；

POST 传输实体，虽然 GET 方法也可以传输主体信息，但是便于区分，我们一般不用 GET 传输实体信息，反而使用 POST 传输实体信息，

PUT 传输文件，PUT 方法用来传输文件。就像 FTP 协议的文件上传一样，要求在请求报文的主体中包含文件内容，然后保存到请求 URI 指定的位置。

但是，鉴于 HTTP 的 PUT 方法自身不带验证机制，任何人都可以上传文件 , 存在安全性问题，因此一般的 W eb 网站不使用该方法。若配合 W eb 应用程序的验证机制，或架构设计采用REST（REpresentational State Transfer，表征状态转移）标准的同类 Web 网站，就可能会开放使用 PUT 方法。

HEAD 获得响应首部，HEAD 方法和 GET 方法一样，只是不返回报文主体部分。用于确认 URI 的有效性及资源更新的日期时间等。

DELETE 删除文件，DELETE 方法用来删除文件，是与 PUT 相反的方法。DELETE 方法按请求 URI 删除指定的资源。

OPTIONS 询问支持的方法，OPTIONS 方法用来查询针对请求 URI 指定的资源支持的方法。

TRACE 追踪路径，TRACE 方法是让 Web 服务器端将之前的请求通信环回给客户端的方法。

CONNECT 要求用隧道协议连接代理，CONNECT 方法要求在与代理服务器通信时建立隧道，实现用隧道协议进行 TCP 通信。主要使用 SSL（Secure Sockets Layer，安全套接层）和 TLS（Transport Layer Security，传输层安全）协议把通信内容加 密后经网络隧道传输。

我们一般最常用的方法也就是 GET 方法和 POST 方法，其他方法暂时了解即可。

## HTTP 的优点和缺点

### HTTP 的优点

简单灵活易扩展

HTTP 最重要也是最突出的优点是 简单、灵活、易于扩展。

HTTP 的协议比较简单，它的主要组成就是 header + body，头部信息也是简单的文本格式，而且 HTTP 的请求报文根据英文也能猜出来个大概的意思，降低学习门槛，能够让更多的人研究和开发 HTTP 应用。

所以，在简单的基础上，HTTP 协议又多了灵活 和 易扩展 的优点。

HTTP 协议里的请求方法、URI、状态码、原因短语、头字段等每一个核心组成要素都没有被制定死，允许开发者任意定制、扩充或解释，给予了浏览器和服务器最大程度的信任和自由。

应用广泛、环境成熟

因为过于简单，普及，因此应用很广泛。因为 HTTP 协议本身不属于一种语言，它并不限定某种编程语言或者操作系统，所以天然具有跨语言、跨平台的优越性。而且，因为本身的简单特性很容易实现，所以几乎所有的编程语言都有 HTTP 调用库和外围的开发测试工具。

随着移动互联网的发展， HTTP 的触角已经延伸到了世界的每一个角落，从简单的 Web 页面到复杂的 JSON、XML 数据，从台式机上的浏览器到手机上的各种 APP、新闻、论坛、购物、手机游戏，你很难找到一个没有使用 HTTP 的地方。

无状态

无状态其实既是优点又是缺点。因为服务器没有记忆能力，所以就不需要额外的资源来记录状态信息，不仅实现上会简单一些，而且还能减轻服务器的负担，能够把更多的 CPU 和内存用来对外提供服务。

### HTTP 的缺点

无状态

既然服务器没有记忆能力，它就无法支持需要连续多个步骤的事务操作。每次都得问一遍身份信息，不仅麻烦，而且还增加了不必要的数据传输量。由此出现了 Cookie 技术。

明文

HTTP 协议里还有一把优缺点一体的双刃剑，就是明文传输。明文意思就是协议里的报文（准确地说是 header 部分）不使用二进制数据，而是用简单可阅读的文本形式。

对比 TCP、UDP 这样的二进制协议，它的优点显而易见，不需要借助任何外部工具，用浏览器、Wireshark 或者 tcpdump 抓包后，直接用肉眼就可以很容易地查看或者修改，为我们的开发调试工作带来极大的便利。

当然缺点也是显而易见的，就是不安全，可以被监听和被窥探。因为无法判断通信双方的身份，不能判断报文是否被更改过。

性能

HTTP 的性能不算差，但不完全适应现在的互联网，还有很大的提升空间。

# Spark

Apache Spark™是用于大规模数据处理的统一分析引擎。

中间结果输出：基于MapReduce的计算引擎通常会将中间结果输出到磁盘上，进行存储和容错。出于任务管道承接的，考虑，当一些查询翻译到MapReduce任务时，往往会产生多个Stage，而这些串联的Stage又依赖于底层文件系统（如HDFS）来存储每一个Stage的输出结果。

Spark是MapReduce的替代方案，而且兼容HDFS、Hive，可融入Hadoop的生态系统，以弥补MapReduce的不足。

Spark 的主要特点还包括:

- (1)提供 Cache 机制来支持需要反复迭代计算或者多次数据共享,减少数据读取的 IO 开销;

- (2)提供了一套支持 DAG 图的分布式并行计算的编程框架,减少多次计算之间中间结果写到 Hdfs 的开销;

- (3)使用多线程池模型减少 Task 启动开稍, shuffle 过程中避免不必要的 sort 操作并减少磁盘 IO 操作。(Hadoop 的 Map 和 reduce 之间的 shuffle 需要 sort)

## Spark的四大特性

高效性

运行速度提高100倍。

Apache Spark使用最先进的DAG调度程序，查询优化程序和物理执行引擎，实现批量和流式数据的高性能。

易用性

Spark支持Java、Python和Scala的API，还支持超过80种高级算法，使用户可以快速构建不同的应用。而且Spark支持交互式的Python和Scala的shell，可以非常方便地在这些shell中使用Spark集群来验证解决问题的方法。

通用性

Spark提供了统一的解决方案。Spark可以用于批处理、交互式查询（Spark SQL）、实时流处理（Spark Streaming）、机器学习（Spark MLlib）和图计算（GraphX）。这些不同类型的处理都可以在同一个应用中无缝使用。Spark统一的解决方案非常具有吸引力，毕竟任何公司都想用统一的平台去处理遇到的问题，减少开发和维护的人力成本和部署平台的物力成本。

兼容性

Spark可以非常方便地与其他的开源产品进行融合。比如，Spark可以使用Hadoop的YARN和Apache Mesos作为它的资源管理和调度器，器，并且可以处理所有Hadoop支持的数据，包括HDFS、HBase和Cassandra等。这对于已经部署Hadoop集群的用户特别重要，因为不需要做任何数据迁移就可以使用Spark的强大处理能力。Spark也可以不依赖于第三方的资源管理和调度器，它实现了Standalone作为其内置的资源管理和调度框架，这样进一步降低了Spark的使用门槛，使得所有人都可以非常容易地部署和使用Spark。此外，Spark还提供了在EC2上部署Standalone的Spark集群的工具。

## Spark 系统架构

首先明确相关术语[^demo\_shuyu]:

- 应用程序(Application): 基于Spark的用户程序，包含了一个Driver Program 和集群中多个的Executor；

- 驱动(Driver): 运行Application的main()函数并且创建SparkContext;

- 执行单元(Executor): 是为某Application运行在Worker Node上的一个进程，该进程负责运行Task，并且负责将数据存在内存或者磁盘上，每个Application都有各自独立的Executors;

- 集群管理程序(Cluster Manager): 在集群上获取资源的外部服务(例如：Local、Standalone、Mesos或Yarn等集群管理系统)；

- 操作(Operation): 作用于RDD的各种操作分为Transformation和Action.

整个 Spark 集群中,分为 Master 节点与 worker 节点,,其中 Master 节点上常驻 Master 守护进程和 Driver 进程, Master 负责将串行任务变成可并行执行的任务集Tasks, 同时还负责出错问题处理等,而 Worker 节点上常驻 Worker 守护进程, Master 节点与 Worker 节点分工不同, Master 负载管理全部的 Worker 节点,而 Worker 节点负责执行任务.

　　Driver 的功能是创建 SparkContext, 负责执行用户写的 Application 的 main 函数进程,Application 就是用户写的程序.

Spark 支持不同的运行模式,包括Local, Standalone,Mesoses,Yarn 模式.不同的模式可能会将 Driver 调度到不同的节点上执行.集群管理模式里, local 一般用于本地调试.

　　每个 Worker 上存在一个或多个 Executor 进程,该对象拥有一个线程池,每个线程负责一个 Task 任务的执行.根据 Executor 上 CPU-core 的数量,其每个时间可以并行多个 跟 core 一样数量的 Task[^demopingtai].Task 任务即为具体执行的 Spark 程序的任务.

## spark 运行原理

一开始看不懂的话可以看完第三和第四章再回来看.

底层详细细节介绍:

　　我们使用spark-submit提交一个Spark作业之后，这个作业就会启动一个对应的Driver进程。根据你使用的部署模式（deploy-mode）不同，Driver进程可能在本地启动，也可能在集群中某个工作节点上启动。而Driver进程要做的第一件事情，就是向集群管理器（可以是Spark Standalone集群，也可以是其他的资源管理集群，美团•大众点评使用的是YARN作为资源管理集群）申请运行Spark作业需要使用的资源，这里的资源指的就是Executor进程。YARN集群管理器会根据我们为Spark作业设置的资源参数，在各个工作节点上，启动一定数量的Executor进程，每个Executor进程都占有一定数量的内存和CPU core。

　　在申请到了作业执行所需的资源之后，Driver进程就会开始调度和执行我们编写的作业代码了。Driver进程会将我们编写的Spark作业代码分拆为多个stage，每个stage执行一部分代码片段，并为每个stage创建一批Task，然后将这些Task分配到各个Executor进程中执行。Task是最小的计算单元，负责执行一模一样的计算逻辑（也就是我们自己编写的某个代码片段），只是每个Task处理的数据不同而已。一个stage的所有Task都执行完毕之后，会在各个节点本地的磁盘文件中写入计算中间结果，然后Driver就会调度运行下一个stage。下一个stage的Task的输入数据就是上一个stage输出的中间结果。如此循环往复，直到将我们自己编写的代码逻辑全部执行完，并且计算完所有的数据，得到我们想要的结果为止。

　　Spark是根据shuffle类算子来进行stage的划分。如果我们的代码中执行了某个shuffle类算子（比如reduceByKey、join等），那么就会在该算子处，划分出一个stage界限来。可以大致理解为，shuffle算子执行之前的代码会被划分为一个stage，shuffle算子执行以及之后的代码会被划分为下一个stage。因此一个stage刚开始执行的时候，它的每个Task可能都会从上一个stage的Task所在的节点，去通过网络传输拉取需要自己处理的所有key，然后对拉取到的所有相同的key使用我们自己编写的算子函数执行聚合操作（比如reduceByKey()算子接收的函数）。这个过程就是shuffle。

　　当我们在代码中执行了cache/persist等持久化操作时，根据我们选择的持久化级别的不同，每个Task计算出来的数据也会保存到Executor进程的内存或者所在节点的磁盘文件中。

　　因此Executor的内存主要分为三块：第一块是让Task执行我们自己编写的代码时使用，默认是占Executor总内存的20%；第二块是让Task通过shuffle过程拉取了上一个stage的Task的输出后，进行聚合等操作时使用，默认也是占Executor总内存的20%；第三块是让RDD持久化时使用，默认占Executor总内存的60%。

　　Task的执行速度是跟每个Executor进程的CPU core数量有直接关系的。一个CPU core同一时间只能执行一个线程。而每个Executor进程上分配到的多个Task，都是以每个Task一条线程的方式，多线程并发运行的。如果CPU core数量比较充足，而且分配到的Task数量比较合理，那么通常来说，可以比较快速和高效地执行完这些Task线程。

## Spark的组成

Spark组成(BDAS)：全称伯克利数据分析栈，通过大规模集成算法、机器、人之间展现大数据应用的一个平台。也是处理大数据、云计算、通信的技术解决方案。

它的主要组件有：

SparkCore：将分布式数据抽象为弹性分布式数据集（RDD），实现了应用任务调度、RPC、序列化和压缩，并为运行在其上的上层组件提供API。

SparkSQL：Spark Sql 是Spark来操作结构化数据的程序包，可以让我使用SQL语句的方式来查询数据，Spark支持 多种数据源，包含Hive表，parquest以及JSON等内容。

SparkStreaming： 是Spark提供的实时数据进行流式计算的组件。

MLlib：提供常用机器学习算法的实现库。

GraphX：提供一个分布式图计算框架，能高效进行图计算。

BlinkDB：用于在海量数据上进行交互式SQL的近似查询引擎。

Tachyon：以内存为中心高容错的的分布式文件系统。

## 什么是RDD？

RDD（Resilient Distributed Dataset）叫做弹性分布式数据集，是Spark中最基本的数据抽象，它代表一个不可变、可分区、里面的元素可并行计算的集合。RDD具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。RDD允许用户在执行多个查询时显式地将工作集缓存在内存中，后续的查询能够重用工作集，这极大地提升了查询速度。

RDD(Resilent Distributed Datasets)俗称弹性分布式数据集,是 Spark 底层的分布式存储的数据结构,可以说是 Spark 的核心, Spark API 的所有操作都是基于 RDD 的. 数据不只存储在一台机器上,而是分布在多台机器上,实现数据计算的并行化.弹性表明数据丢失时,可以进行重建.在Spark 1.5版以后,新增了数据结构 Spark-DataFrame,仿造的 R 和 python 的类 SQL 结构-DataFrame, 底层为 RDD, 能够让数据从业人员更好的操作 RDD.

　　在Spark 的设计思想中,为了减少网络及磁盘 IO 开销,需要设计出一种新的容错方式,于是才诞生了新的数据结构 RDD. RDD 是一种只读的数据块,可以从外部数据转换而来,你可以对RDD 进行函数操作(Operation),包括 Transformation 和 Action. 在这里只读表示当你对一个 RDD 进行了操作,那么结果将会是一个新的 RDD, 这种情况放在代码里,假设变换前后都是使用同一个变量表示这一 RDD,RDD 里面的数据并不是真实的数据,而是一些元数据信息,记录了该 RDD 是通过哪些 Transformation 得到的,在计算机中使用 lineage 来表示这种血缘结构,lineage 形成一个有向无环图 DAG, 整个计算过程中,将不需要将中间结果落地到 HDFS 进行容错,加入某个节点出错,则只需要通过 lineage 关系重新计算即可.

1).RDD 主要具有如下特点:

- 1.它是在集群节点上的不可变的、已分区的集合对象;

- 2.通过并行转换的方式来创建(如 Map、 filter、join 等);

- 3.失败自动重建;

- 4.可以控制存储级别(内存、磁盘等)来进行重用;

- 5.必须是可序列化的;

- 6.是静态类型的(只读)。

2).RDD 的创建方式主要有2种:

- 并行化(Parallelizing)一个已经存在与驱动程序(Driver Program)中的集合如set、list;

- 读取外部存储系统上的一个数据集，比如HDFS、Hive、HBase,或者任何提供了Hadoop InputFormat的数据源.也可以从本地读取 txt、csv 等数据集

3).RDD 的操作函数(operation)主要分为2种类型 Transformation 和 Action.



Transformation 操作不是马上提交 Spark 集群执行的,Spark 在遇到 Transformation 操作时只会记录需要这样的操作,并不会去执行,需要等到有 Action 操作的时候才会真正启动计算过程进行计算.针对每个 Action,Spark 会生成一个 Job, 从数据的创建开始,经过 Transformation, 结尾是 Action 操作.这些操作对应形成一个有向无环图(DAG),形成 DAG 的先决条件是最后的函数操作是一个Action.

### RDD的属性

（1）一组分片（Partition），即数据集的基本组成单位。对于RDD来说，每个分片都会被一个计算任务处理，并决定并行计算的粒度。用户可以在创建RDD时指定RDD的分片个数，如果没有指定，那么就会采用默认值。默认值就是程序所分配到的CPU Core的数目。

（2）一个计算每个分区的函数。Spark中RDD的计算是以分片为单位的，每个RDD都会实现compute函数以达到这个目的。compute函数会对迭代器进行复合，不需要保存每次计算的结果。

（3）RDD之间的依赖关系。RDD的每次转换都会生成一个新的RDD，所以RDD之间就会形成类似于流水线一样的前后依赖关系。在部分分区数据丢失时，Spark可以通过这个依赖关系重新计算丢失的分区数据，而不是对RDD的所有分区进行重新计算。

（4）一个Partitioner，即RDD的分片函数。当前Spark中实现了两种类型的分片函数，一个是基于哈希的HashPartitioner，另外一个是基于范围的RangePartitioner。只有对于于key-value的RDD，才会有Partitioner，非key-value的RDD的Parititioner的值是None。Partitioner函数不但决定了RDD本身的分片数量，也决定了parent RDD Shuffle输出时的分片数量。

（5）一个列表，存储存取每个Partition的优先位置（preferred location）。对于一个HDFS文件来说，这个列表保存的就是每个Partition所在的块的位置。按照“移动数据不如移动计算”的理念，Spark在进行任务调度的时候，会尽可能地将计算任务分配到其所要处理数据块的存储位置。

### RDD的创建方式

2.1　通过读取文件生成的

由外部存储系统的数据集创建，包括本地的文件系统，还有所有Hadoop支持的数据集，比如HDFS、Cassandra、HBase等

通过并行化的方式创建RDD

由一个已经存在的Scala集合创建。

## RDD编程API

Spark支持两个类型（算子）操作：Transformation和Action

### Transformation

主要做的是就是将一个已有的RDD生成另外一个RDD。Transformation具有lazy特性(延迟加载)。Transformation算子的代码不会真正被执行。只有当我们的程序里面遇到一个action算子的时候，代码才会真正的被执行。这种设计让Spark更加有效率地运行。

常用的Transformation：

转换

含义

map(func)

返回一个新的RDD，该RDD由每一个输入元素经过func函数转换后组成

filter(func)

返回一个新的RDD，该RDD由经过func函数计算后返回值为true的输入元素组成

flatMap(func)

类似于map，但是每一个输入元素可以被映射为0或多个输出元素（所以func应该返回一个序列，而不是单一元素）

mapPartitions(func)

类似于map，但独立地在RDD的每一个分片上运行，因此在类型为T的RDD上运行时，func的函数类型必须是Iterator[T] => Iterator[U]

mapPartitionsWithIndex(func)

类似于mapPartitions，但func带有一个整数参数表示分片的索引值，因此在类型为T的RDD上运行时，func的函数类型必须是

(Int, Interator[T]) => Iterator[U]

sample(withReplacement, fraction, seed)

根据fraction指定的比例对数据进行采样，可以选择是否使用随机数进行替换，seed用于指定随机数生成器种子

union(otherDataset)

对源RDD和参数RDD求并集后返回一个新的RDD

intersection(otherDataset)

对源RDD和参数RDD求交集后返回一个新的RDD

distinct([numTasks]))

对源RDD进行去重后返回一个新的RDD

groupByKey([numTasks])

在一个(K,V)的RDD上调用，返回一个(K, Iterator[V])的RDD

reduceByKey(func, [numTasks])

在一个(K,V)的RDD上调用，返回一个(K,V)的RDD，使用指定的reduce函数，将相同key的值聚合到一起，与groupByKey类似，reduce任务的个数可以通过第二个可选的参数来设置

aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])

先按分区聚合 再总的聚合 每次要跟初始值交流 例如：aggregateByKey(0)(\_+\_,\_+\_) 对k/y的RDD进行操作

sortByKey([ascending], [numTasks])

在一个(K,V)的RDD上调用，K必须实现Ordered接口，返回一个按照key进行排序的(K,V)的RDD

sortBy(func,[ascending], [numTasks])

与sortByKey类似，但是更灵活 第一个参数是根据什么排序 第二个是怎么排序 false倒序 第三个排序后分区数 默认与原RDD一样

join(otherDataset, [numTasks])

在类型为(K,V)和(K,W)的RDD上调用，返回一个相同key对应的所有元素对在一起的(K,(V,W))的RDD 相当于内连接（求交集）

cogroup(otherDataset, [numTasks])

在类型为(K,V)和(K,W)的RDD上调用，返回一个(K,(Iterable<V>,Iterable<W>))类型的RDD

cartesian(otherDataset)

两个RDD的笛卡尔积 的成很多个K/V

pipe(command, [envVars])

调用外部程序

coalesce(numPartitions)

重新分区 第一个参数是要分多少区，第二个参数是否shuffle 默认false 少分区变多分区 true 多分区变少分区 false

repartition(numPartitions)

重新分区 必须shuffle 参数是要分多少区 少变多

repartitionAndSortWithinPartitions(partitioner)

重新分区+排序 比先分区再排序效率高 对K/V的RDD进行操作

foldByKey(zeroValue)(seqOp)

该函数用于K/V做折叠，合并处理 ，与aggregate类似 第一个括号的参数应用于每个V值 第二括号函数是聚合例如：\_+\_

combineByKey

合并相同的key的值 rdd1.combineByKey(x => x, (a: Int, b: Int) => a + b, (m: Int, n: Int) => m + n)

partitionBy（partitioner）

对RDD进行分区 partitioner是分区器 例如new HashPartition(2

cache

RDD缓存，可以避免重复计算从而减少时间，区别：cache内部调用了persist算子，cache默认就一个缓存级别MEMORY-ONLY ，而persist则可以选择缓存级别

persist

Subtract（rdd）

返回前rdd元素不在后rdd的rdd

leftOuterJoin

leftOuterJoin类似于SQL中的左外关联left outer join，返回结果以前面的RDD为主，关联不上的记录为空。只能用于两个RDD之间的关联，如果要多个RDD关联，多关联几次即可。

rightOuterJoin

rightOuterJoin类似于SQL中的有外关联right outer join，返回结果以参数中的RDD为主，关联不上的记录为空。只能用于两个RDD之间的关联，如果要多个RDD关联，多关联几次即可

subtractByKey

substractByKey和基本转换操作中的subtract类似只不过这里是针对K的，返回在主RDD中出现，并且不在otherRDD中出现的元素

### Action

触发代码的运行，我们一段spark代码里面至少需要有一个action操作。

常用的Action:

动作

含义

reduce(func)

通过func函数聚集RDD中的所有元素，这个功能必须是课交换且可并联的

collect()

在驱动程序中，以数组的形式返回数据集的所有元素

count()

返回RDD的元素个数

first()

返回RDD的第一个元素（类似于take(1)）

take(n)

返回一个由数据集的前n个元素组成的数组

takeSample(withReplacement,num, [seed])

返回一个数组，该数组由从数据集中随机采样的num个元素组成，可以选择是否用随机数替换不足的部分，seed用于指定随机数生成器种子

takeOrdered(n, [ordering])

saveAsTextFile(path)

将数据集的元素以textfile的形式保存到HDFS文件系统或者其他支持的文件系统，对于每个元素，Spark将会调用toString方法，将它装换为文件中的文本

saveAsSequenceFile(path)

将数据集中的元素以Hadoop sequencefile的格式保存到指定的目录下，可以使HDFS或者其他Hadoop支持的文件系统。

saveAsObjectFile(path)

countByKey()

针对(K,V)类型的RDD，返回一个(K,Int)的map，表示每一个key对应的元素个数。

foreach(func)

在数据集的每一个元素上，运行函数func进行更新。

aggregate

先对分区进行操作，在总体操作

reduceByKeyLocally

lookup

top

fold

foreachPartition

## shuffle 和 stage

shuffle 是划分 DAG 中 stage 的标识,同时影响 Spark 执行速度的关键步骤.

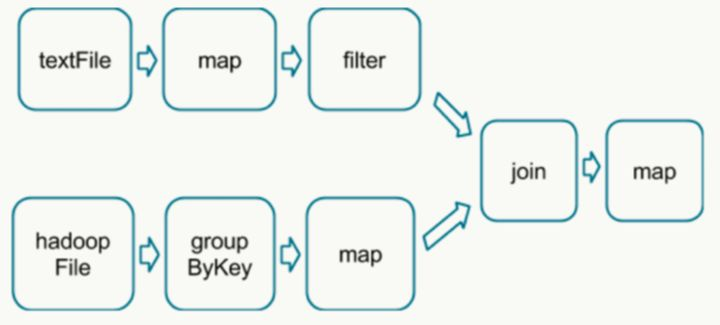
　　RDD 的 Transformation 函数中,又分为窄依赖(narrow dependency)和宽依赖(wide dependency)的操作.窄依赖跟宽依赖的区别是是否发生 shuffle(洗牌) 操作.宽依赖会发生 shuffle 操作. 窄依赖是子 RDD的各个分片(partition)不依赖于其他分片,能够独立计算得到结果,宽依赖指子 RDD 的各个分片会依赖于父RDD 的多个分片,所以会造成父 RDD 的各个分片在集群中重新分片, 看如下两个示例:

第一个 Map 操作将 RDD 里的各个元素进行映射, RDD 的各个数据元素之间不存在依赖,可以在集群的各个内存中独立计算,也就是并行化,第二个 groupby 之后的 Map 操作,为了计算相同 key 下的元素个数,需要把相同 key 的元素聚集到同一个 partition 下,所以造成了数据在内存中的重新分布,即 shuffle 操作.shuffle 操作是 spark 中最耗时的操作,应尽量避免不必要的 shuffle.

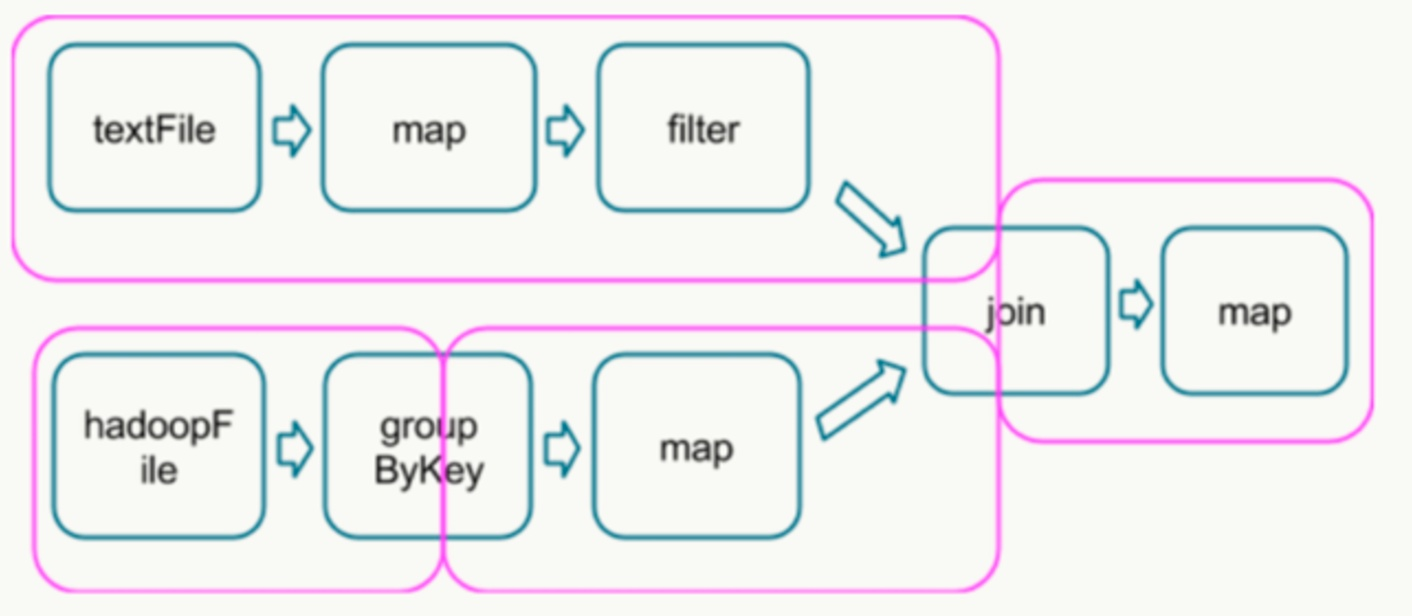
　　宽依赖主要有两个过程: shuffle write 和 shuffle fetch. 类似 Hadoop 的 Map 和 Reduce 阶段.shuffle write 将 ShuffleMapTask 任务产生的中间结果缓存到内存中, shuffle fetch 获得 ShuffleMapTask 缓存的中间结果进行 ShuffleReduceTask 计算,这个过程容易造成OutOfMemory.

　　shuffle 过程内存分配使用 ShuffleMemoryManager 类管理,会针对每个 Task 分配内存,Task 任务完成后通过 Executor 释放空间.这里可以把 Task 理解成不同 key 的数据对应一个 Task. 早期的内存分配机制使用公平分配,即不同 Task 分配的内存是一样的,但是这样容易造成内存需求过多的 Task 的 OutOfMemory, 从而造成多余的 磁盘 IO 过程,影响整体的效率.(例:某一个 key 下的数据明显偏多,但因为大家内存都一样,这一个 key 的数据就容易 OutOfMemory).1.5版以后 Task 共用一个内存池,内存池的大小默认为 JVM 最大运行时内存容量的16%,分配机制如下:假如有 N 个 Task,ShuffleMemoryManager 保证每个 Task 溢出之前至少可以申请到1/2N 内存,且至多申请到1/N,N 为当前活动的 shuffle Task 数,因为N 是一直变化的,所以 manager 会一直追踪 Task 数的变化,重新计算队列中的1/N 和1/2N.但是这样仍然容易造成内存需要多的 Task 任务溢出,所以最近有很多相关的研究是针对 shuffle 过程内存优化的.

如下 DAG 流程图中,分别读取数据,经过处理后 join 2个 RDD 得到结果



在这个图中,根据是否发生 shuffle 操作能够将其分成如下的 stage 类型:



运行到每个 stage 的边界时，数据在父 stage 中按照 Task 写到磁盘上，而在子 stage 中通过网络按照 Task 去读取数据。这些操作会导致很重的网络以及磁盘的I/O，所以 stage 的边界是非常占资源的，在编写 Spark 程序的时候需要尽量避免的 。父 stage 中 partition 个数与子 stage 的 partition 个数可能不同，所以那些产生 stage 边界的 Transformation 常常需要接受一个 numPartition 的参数来觉得子 stage 中的数据将被切分为多少个 partition[^demoa]。

PS:shuffle 操作的时候可以用 combiner 压缩数据,减少 IO 的消耗

## Spark的广播变量和累加器

概述

在spark程序中，当一个传递给Spark操作(例如map和reduce)的函数在远程节点上面运行时，Spark操作实际上操作的是这个函数所用变量的一个独立副本。这些变量会被复制到每台机器上，并且这些变量在远程机器上的所有更新都不会传递回驱动程序。通常跨任务的读写变量是低效的，但是，Spark还是为两种常见的使用模式提供了两种有限的共享变量：广播变（broadcast variable）和累加器（accumulator）

如果我们要在分布式计算里面分发大对象，例如：字典，集合，黑白名单等，这个都会由Driver端进行分发，一般来讲，如果这个变量不是广播变量，那么每个task就会分发一份，这在task数目十分多的情况下Driver的带宽会成为系统的瓶颈，而且会大量消耗task服务器上的资源，如果将这个变量声明为广播变量，那么知识每个executor拥有一份，这个executor启动的task会共享这个变量，节省了通信的成本和服务器的资源。

为什么要将一个变量定义为一个累加器？

在spark应用程序中，我们经常会有这样的需求，如异常监控，调试，记录符合某特性的数据的数目，这种需求都需要用到计数器，如果一个变量不被声明为一个累加器，那么它将在被改变时不会再driver端进行全局汇总，即在分布式运行时每个task运行的只是原始变量的一个副本，并不能改变原始变量的值，但是当这个变量被声明为累加器后，该变量就会有分布式计数的功能。

## 基本概念

（1）Application：表示你的应用程序

（2）Driver：表示main()函数，创建SparkContext。由SparkContext负责与ClusterManager通信，进行资源的申请，任务的分配和监控等。程序执行完毕后关闭SparkContext

（3）Executor：某个Application运行在Worker节点上的一个进程，该进程负责运行某些task，并且负责将数据存在内存或者磁盘上。在Spark on Yarn模式下，其进程名称为 CoarseGrainedExecutor Backend，一个CoarseGrainedExecutor Backend进程有且仅有一个executor对象，它负责将Task包装成taskRunner，并从线程池中抽取出一个空闲线程运行Task，这样，每个CoarseGrainedExecutorBackend能并行运行Task的数据就取决于分配给它的CPU的个数。

（4）Worker：集群中可以运行Application代码的节点。在Standalone模式中指的是通过slave文件配置的worker节点，在Spark on Yarn模式中指的就是NodeManager节点。

（5）Task：在Executor进程中执行任务的工作单元，多个Task组成一个Stage

（6）Job：包含多个Task组成的并行计算，是由Action行为触发的

（7）Stage：每个Job会被拆分很多组Task，作为一个TaskSet，其名称为Stage

（8）DAGScheduler：根据Job构建基于Stage的DAG，并提交Stage给TaskScheduler，其划分Stage的依据是RDD之间的依赖关系

（9）TaskScheduler：将TaskSet提交给Worker（集群）运行，每个Executor运行什么Task就是在此处分配的。

## Spark的基本运行流程

1、说明

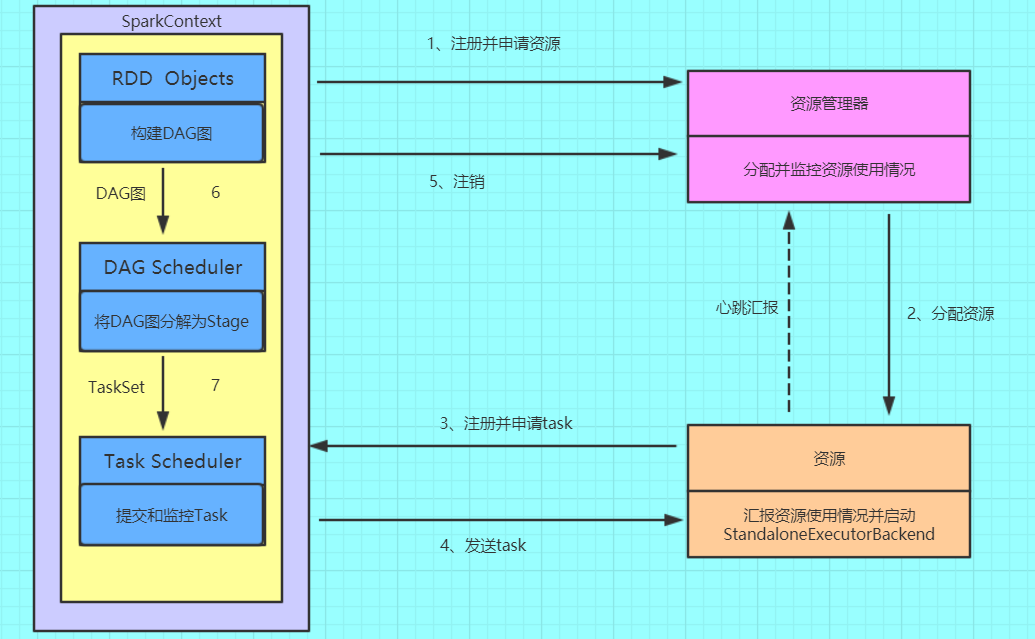
(1)构建Spark Application的运行环境（启动SparkContext），SparkContext向资源管理器（可以是Standalone、Mesos或YARN）注册并申请运行Executor资源；

(2)资源管理器分配Executor资源并启动StandaloneExecutorBackend，Executor运行情况将随着心跳发送到资源管理器上；

(3)SparkContext构建成DAG图，将DAG图分解成Stage，并把Taskset发送给Task Scheduler。Executor向SparkContext申请Task

(4)Task Scheduler将Task发放给Executor运行同时SparkContext将应用程序代码发放给Executor。

(5)Task在Executor上运行，运行完毕释放所有资源。



## Spark运行架构特点

（1）每个Application获取专属的executor进程，该进程在Application期间一直驻留，并以多线程方式运行tasks。这种Application隔离机制有其优势的，无论是从调度角度看（每个Driver调度它自己的任务），还是从运行角度看（来自不同Application的Task运行在不同的JVM中）。当然，这也意味着Spark Application不能跨应用程序共享数据，除非将数据写入到外部存储系统。

（2）Spark与资源管理器无关，只要能够获取executor进程，并能保持相互通信就可以了。

（3）提交SparkContext的Client应该靠近Worker节点（运行Executor的节点)，最好是在同一个Rack里，因为Spark Application运行过程中SparkContext和Executor之间有大量的信息交换；如果想在远程集群中运行，最好使用RPC将SparkContext提交给集群，不要远离Worker运行SparkContext。

（4）Task采用了数据本地性和推测执行的优化机制。

## DAGScheduler

Job=多个stage，Stage=多个同种task, Task分为ShuffleMapTask和ResultTask，Dependency分为ShuffleDependency和NarrowDependency

面向stage的切分，切分依据为宽依赖

维护waiting jobs和active jobs，维护waiting stages、active stages和failed stages，以及与jobs的映射关系

主要职能：

1、接收提交Job的主入口，submitJob(rdd, ...)或runJob(rdd, ...)。在SparkContext里会调用这两个方法。

生成一个Stage并提交，接着判断Stage是否有父Stage未完成，若有，提交并等待父Stage，以此类推。结果是：DAGScheduler里增加了一些waiting stage和一个running stage。

running stage提交后，分析stage里Task的类型，生成一个Task描述，即TaskSet。

调用TaskScheduler.submitTask(taskSet, ...)方法，把Task描述提交给TaskScheduler。TaskScheduler依据资源量和触发分配条件，会为这个TaskSet分配资源并触发执行。

DAGScheduler提交job后，异步返回JobWaiter对象，能够返回job运行状态，能够cancel job，执行成功后会处理并返回结果

2、处理TaskCompletionEvent

如果task执行成功，对应的stage里减去这个task，做一些计数工作：

如果task是ResultTask，计数器Accumulator加一，在job里为该task置true，job finish总数加一。加完后如果finish数目与partition数目相等，说明这个stage完成了，标记stage完成，从running stages里减去这个stage，做一些stage移除的清理工作

如果task是ShuffleMapTask，计数器Accumulator加一，在stage里加上一个output location，里面是一个MapStatus类。MapStatus是ShuffleMapTask执行完成的返回，包含location信息和block size(可以选择压缩或未压缩)。同时检查该stage完成，向MapOutputTracker注册本stage里的shuffleId和location信息。然后检查stage的output location里是否存在空，若存在空，说明一些task失败了，整个stage重新提交；否则，继续从waiting stages里提交下一个需要做的stage

如果task是重提交，对应的stage里增加这个task

如果task是fetch失败，马上标记对应的stage完成，从running stages里减去。如果不允许retry，abort整个stage；否则，重新提交整个stage。另外，把这个fetch相关的location和map任务信息，从stage里剔除，从MapOutputTracker注销掉。最后，如果这次fetch的blockManagerId对象不为空，做一次ExecutorLost处理，下次shuffle会换在另一个executor上去执行。

其他task状态会由TaskScheduler处理，如Exception, TaskResultLost, commitDenied等。

3、其他与job相关的操作还包括：cancel job， cancel stage, resubmit failed stage等

其他职能：

cacheLocations 和 preferLocation

## TaskScheduler

维护task和executor对应关系，executor和物理资源对应关系，在排队的task和正在跑的task。

内部维护一个任务队列，根据FIFO或Fair策略，调度任务。

TaskScheduler本身是个接口，spark里只实现了一个TaskSchedulerImpl，理论上任务调度可以定制。

主要功能：

1、submitTasks(taskSet)，接收DAGScheduler提交来的tasks

为tasks创建一个TaskSetManager，添加到任务队列里。TaskSetManager跟踪每个task的执行状况，维护了task的许多具体信息。

触发一次资源的索要。

首先，TaskScheduler对照手头的可用资源和Task队列，进行executor分配(考虑优先级、本地化等策略)，符合条件的executor会被分配给TaskSetManager。

然后，得到的Task描述交给SchedulerBackend，调用launchTask(tasks)，触发executor上task的执行。task描述被序列化后发给executor，executor提取task信息，调用task的run()方法执行计算。

2、cancelTasks(stageId)，取消一个stage的tasks

调用SchedulerBackend的killTask(taskId, executorId, ...)方法。taskId和executorId在TaskScheduler里一直维护着。

3、resourceOffer(offers: Seq[Workers])，这是非常重要的一个方法，调用者是SchedulerBacnend，用途是底层资源SchedulerBackend把空余的workers资源交给TaskScheduler，让其根据调度策略为排队的任务分配合理的cpu和内存资源，然后把任务描述列表传回给SchedulerBackend

从worker offers里，搜集executor和host的对应关系、active executors、机架信息等等

worker offers资源列表进行随机洗牌，任务队列里的任务列表依据调度策略进行一次排序

遍历每个taskSet，按照进程本地化、worker本地化、机器本地化、机架本地化的优先级顺序，为每个taskSet提供可用的cpu核数，看是否满足

默认一个task需要一个cpu，设置参数为"spark.task.cpus=1"

为taskSet分配资源，校验是否满足的逻辑，最终在TaskSetManager的resourceOffer(execId, host, maxLocality)方法里

满足的话，会生成最终的任务描述，并且调用DAGScheduler的taskStarted(task, info)方法，通知DAGScheduler，这时候每次会触发DAGScheduler做一次submitMissingStage的尝试，即stage的tasks都分配到了资源的话，马上会被提交执行

4、statusUpdate(taskId, taskState, data),另一个非常重要的方法，调用者是SchedulerBacnend，用途是SchedulerBacnend会将task执行的状态汇报给TaskScheduler做一些决定

若TaskLost，找到该task对应的executor，从active executor里移除，避免这个executor被分配到其他task继续失败下去。

task finish包括四种状态：finished, killed, failed, lost。只有finished是成功执行完成了。其他三种是失败。

task成功执行完，调用TaskResultGetter.enqueueSuccessfulTask(taskSet, tid, data)，否则调用TaskResultGetter.enqueueFailedTask(taskSet, tid, state, data)。TaskResultGetter内部维护了一个线程池，负责异步fetch task执行结果并反序列化。默认开四个线程做这件事，可配参数"spark.resultGetter.threads"=4。

TaskResultGetter取task result的逻辑

1、对于success task，如果taskResult里的数据是直接结果数据，直接把data反序列出来得到结果；如果不是，会调用blockManager.getRemoteBytes(blockId)从远程获取。如果远程取回的数据是空的，那么会调用TaskScheduler.handleFailedTask，告诉它这个任务是完成了的但是数据是丢失的。否则，取到数据之后会通知BlockManagerMaster移除这个block信息，调用TaskScheduler.handleSuccessfulTask，告诉它这个任务是执行成功的，并且把result data传回去。

2、对于failed task，从data里解析出fail的理由，调用TaskScheduler.handleFailedTask，告诉它这个任务失败了，理由是什么。

## SchedulerBackend

在TaskScheduler下层，用于对接不同的资源管理系统，维护executor相关信息(包括executor的地址、通信端口、host、总核数，剩余核数)，手头上executor有多少被注册使用了，有多少剩余，总共还有多少核是空的等等。

## Executor

Executor是spark里的进程模型，可以套用到不同的资源管理系统上，与SchedulerBackend配合使用。

内部有个线程池，有个running tasks map，有个actor，接收上面提到的由SchedulerBackend发来的事件。

事件处理

launchTask。根据task描述，生成一个TaskRunner线程，丢尽running tasks map里，用线程池执行这个TaskRunner

killTask。从running tasks map里拿出线程对象，调它的kill方法。

## Spark on Standalone运行过程

Standalone模式是Spark实现的资源调度框架，其主要的节点有Client节点、Master节点和Worker节点。其中Driver既可以运行在Master节点上中，也可以运行在本地Client端。当用spark-shell交互式工具提交Spark的Job时，Driver在Master节点上运行；当使用spark-submit工具提交Job或者在Eclips、IDEA等开发平台上使用”new SparkConf().setMaster(“spark://master:7077”)”方式运行Spark任务时，Driver是运行在本地Client端上的。

运行过程文字说明

1、我们提交一个任务，任务就叫Application

2、初始化程序的入口SparkContext，

　　2.1 初始化DAG Scheduler

　　2.2 初始化Task Scheduler

3、Task Scheduler向master去进行注册并申请资源（CPU Core和Memory）

4、Master根据SparkContext的资源申请要求和Worker心跳周期内报告的信息决定在哪个Worker上分配资源，然后在该Worker上获取资源，然后启动StandaloneExecutorBackend；顺便初

始化好了一个线程池

5、StandaloneExecutorBackend向Driver(SparkContext)注册,这样Driver就知道哪些Executor为他进行服务了。

　 到这个时候其实我们的初始化过程基本完成了，我们开始执行transformation的代码，但是代码并不会真正的运行，直到我们遇到一个action操作。生产一个job任务，进行stage的划分

6、SparkContext将Applicaiton代码发送给StandaloneExecutorBackend；并且SparkContext解析Applicaiton代码，构建DAG图，并提交给DAG Scheduler分解成Stage（当碰到Action操作 时，就会催生Job；每个Job中含有1个或多个Stage，Stage一般在获取外部数据和shuffle之前产生）。

7、将Stage（或者称为TaskSet）提交给Task Scheduler。Task Scheduler负责将Task分配到相应的Worker，最后提交给StandaloneExecutorBackend执行；

8、对task进行序列化，并根据task的分配算法，分配task

9、对接收过来的task进行反序列化，把task封装成一个线程

10、开始执行Task，并向SparkContext报告，直至Task完成。

11、资源注销

## Spark on YARN运行过程

YARN是一种统一资源管理机制，在其上面可以运行多套计算框架。目前的大数据技术世界，大多数公司除了使用Spark来进行数据计算，由于历史原因或者单方面业务处理的性能考虑而使用着其他的计算框架，比如MapReduce、Storm等计算框架。Spark基于此种情况开发了Spark on YARN的运行模式，由于借助了YARN良好的弹性资源管理机制，不仅部署Application更加方便，而且用户在YARN集群中运行的服务和Application的资源也完全隔离，更具实践应用价值的是YARN可以通过队列的方式，管理同时运行在集群中的多个服务。

Spark on YARN模式根据Driver在集群中的位置分为两种模式：一种是YARN-Client模式，另一种是YARN-Cluster（或称为YARN-Standalone模式）。

## YARN-Client

Yarn-Client模式中，Driver在客户端本地运行，这种模式可以使得Spark Application和客户端进行交互，因为Driver在客户端，所以可以通过webUI访问Driver的状态，默认是http://hadoop1:4040访问，而YARN通过http:// hadoop1:8088访问。

YARN-client的工作流程分为以下几个步骤：

文字说明

1.Spark Yarn Client向YARN的ResourceManager申请启动Application Master。同时在SparkContent初始化中将创建DAGScheduler和TASKScheduler等，由于我们选择的是Yarn-Client模式，程序会选择YarnClientClusterScheduler和YarnClientSchedulerBackend；

2.ResourceManager收到请求后，在集群中选择一个NodeManager，为该应用程序分配第一个Container，要求它在这个Container中启动应用程序的ApplicationMaster，与YARN-Cluster区别的是在该ApplicationMaster不运行SparkContext，只与SparkContext进行联系进行资源的分派；

3.Client中的SparkContext初始化完毕后，与ApplicationMaster建立通讯，向ResourceManager注册，根据任务信息向ResourceManager申请资源（Container）；

4.一旦ApplicationMaster申请到资源（也就是Container）后，便与对应的NodeManager通信，要求它在获得的Container中启动启动CoarseGrainedExecutorBackend，CoarseGrainedExecutorBackend启动后会向Client中的SparkContext注册并申请Task；

5.Client中的SparkContext分配Task给CoarseGrainedExecutorBackend执行，CoarseGrainedExecutorBackend运行Task并向Driver汇报运行的状态和进度，以让Client随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务；

6.应用程序运行完成后，Client的SparkContext向ResourceManager申请注销并关闭自己。

## YARN-Cluster

在YARN-Cluster模式中，当用户向YARN中提交一个应用程序后，YARN将分两个阶段运行该应用程序：第一个阶段是把Spark的Driver作为一个ApplicationMaster在YARN集群中先启动；第二个阶段是由ApplicationMaster创建应用程序，然后为它向ResourceManager申请资源，并启动Executor来运行Task，同时监控它的整个运行过程，直到运行完成。

YARN-cluster的工作流程分为以下几个步骤：

文字说明

1. Spark Yarn Client向YARN中提交应用程序，包括ApplicationMaster程序、启动ApplicationMaster的命令、需要在Executor中运行的程序等；

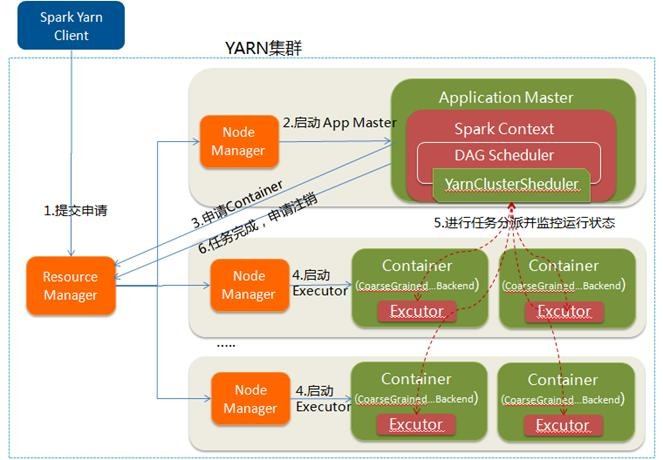
2. ResourceManager收到请求后，在集群中选择一个NodeManager，为该应用程序分配第一个Container，要求它在这个Container中启动应用程序的ApplicationMaster，其中ApplicationMaster进行SparkContext等的初始化；

3. ApplicationMaster向ResourceManager注册，这样用户可以直接通过ResourceManage查看应用程序的运行状态，然后它将采用轮询的方式通过RPC协议为各个任务申请资源，并监控它们的运行状态直到运行结束；

4. 一旦ApplicationMaster申请到资源（也就是Container）后，便与对应的NodeManager通信，要求它在获得的Container中启动启动CoarseGrainedExecutorBackend，CoarseGrainedExecutorBackend启动后会向ApplicationMaster中的SparkContext注册并申请Task。这一点和Standalone模式一样，只不过SparkContext在Spark Application中初始化时，使用CoarseGrainedSchedulerBackend配合YarnClusterScheduler进行任务的调度，其中YarnClusterScheduler只是对TaskSchedulerImpl的一个简单包装，增加了对Executor的等待逻辑等；

5. ApplicationMaster中的SparkContext分配Task给CoarseGrainedExecutorBackend执行，CoarseGrainedExecutorBackend运行Task并向ApplicationMaster汇报运行的状态和进度，以让ApplicationMaster随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务；

6. 应用程序运行完成后，ApplicationMaster向ResourceManager申请注销并关闭自己。



## YARN-Client 与 YARN-Cluster 区别

理解YARN-Client和YARN-Cluster深层次的区别之前先清楚一个概念：Application Master。在YARN中，每个Application实例都有一个ApplicationMaster进程，它是Application启动的第一个容器。它负责和ResourceManager打交道并请求资源，获取资源之后告诉NodeManager为其启动Container。从深层次的含义讲YARN-Cluster和YARN-Client模式的区别其实就是ApplicationMaster进程的区别。

1、YARN-Cluster模式下，Driver运行在AM(Application Master)中，它负责向YARN申请资源，并监督作业的运行状况。当用户提交了作业之后，就可以关掉Client，作业会继续在YARN上运行，因而YARN-Cluster模式不适合运行交互类型的作业；

2、YARN-Client模式下，Application Master仅仅向YARN请求Executor，Client会和请求的Container通信来调度他们工作，也就是说Client不能离开。

# 常用框架源码

## Mybatis

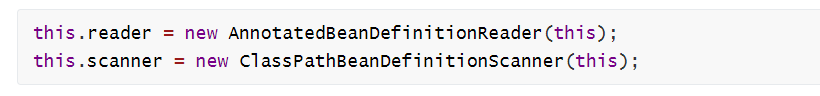
## Spring

### 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);

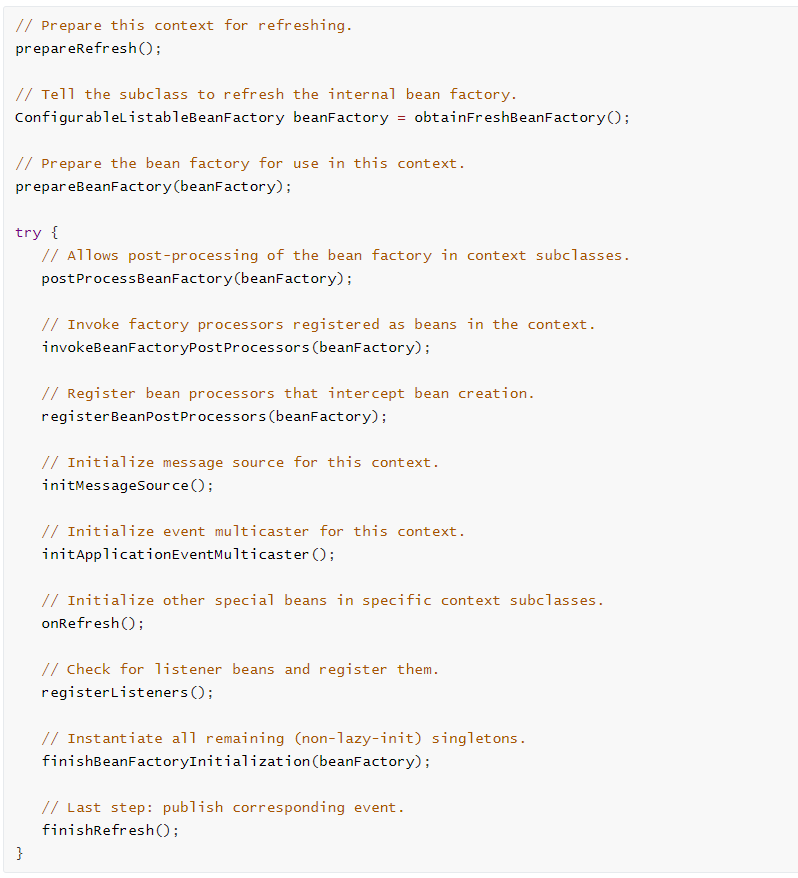
设置bean定义的属性：是否懒加载，Bean初始化销毁的顺序、是否首选、设置依赖、描述等等

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

面试官，您好！我叫张鹏飞。我17年从南京理工大学毕业，目前有参加正式工作有三年多的工作经验，而且大四的时候也在亚信实习了一年时间，毕业后在亚信工作一年时间后，加入华为工作。在我的工作生涯中一直使用java来完成自己的项目。 我了解过Java的常用集合的实现原理，掌握多线程编程和JVM相关体系知识。熟练使用Spring、MyBatis等框架。而且我对python也有一定的了解，经常使用python写一些常用的工具。而且因为自己最近一直在搞一个大数据计算平台，对spark框架有一定的经验，也看过spark的源码，这个时候就需要去了解一下scala语言的相关知识，对Scala语言有入门的了解。平时的学习方式就是在github上找一些优秀的项目去阅读学习。在github上也有自己的账号，开源了一个多线程的python下载工具，目前有40个start4

想离开华为是因为我想在技术上得到更多的锻炼。

在华为我主要参与的是规划设计工具和宙斯盾平台系统，负责搭建了整个项目的基础架构，属于工具的owner，带领合作方的开发人员一起保证工具的需求迭代和上线维护。

规划设计工具是根据用户设计的ip路由，连线等等规则，自动输出LLD和ZTP等相关部署调测的设计表格和图纸。项目构建使用的springboot，持久层框架使用的是Mybatis。前台使用VUE构建是一个前后端分离的项目；

今年我的主要工作就聚焦在宙斯盾平台上了，这是一个进行现网数据的分析，导出风险并推送的一个平台。平时的工作主要是构建Spark计算任务去HDFS上的文件中匹配风险进行推送，HDFS上的文件数目大约有近10w，其中包含的网元数据有超1000w台，最近对整个宙斯盾平台的数据进行改造，建立了每个文件的索引，记录了文件中所有有效网元的设备类型，版本补丁等信息，以后我们每次进行风险扫描的时候就不需要全量遍历所有的文件了，根据设备类型版本等信息进行匹配，是整个宙斯盾风险分析的效率有了很大的提升。因为这件事也得到了领导的好评，获得了华为的金网络奖。

我一直都非常想加入贵公司，我觉得贵公司的文化和技术氛围我都非常喜欢，期待能与你共事！

## 你有什么问题需要问我

\*\*团队现在面临的最大挑战是什么？\*\*

\*\*这个岗位为什么还在招人？\*\*

希望能加入贵公司，拿到贵公司的offer，感觉您这边抽时间来对我进行面试。

## 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

## 宙斯盾项目分析：

### 一、索引目录的创建

对集群上的所有文件进行扫描，根据HDFS\_PATH（集群上的文件目录），LINE\_NUM(每一个文件对应的行数)作为key值扫描并统计

site\_Id(局点id)、

DEV\_FORM（设备形态）、

PRODUCT\_VERSION（版本）、

PROBSOLVED\_PATCH（补丁）等信息

作为规则执行的索引结构进行索引分析。

索引结构图如下：

目前集群上的log文件已经扫描出相关索引入库

### 二、规则执行的过程：

1、获取需要执行的所有规则列表

2、根据规则属性（设备形态，版本和补丁），获取每个规则需要处理的log文件的索引，得到需要扫描的文件列表

3、将获取到的所有文件加载到spark容器中，在集群上依次执行规则匹配。

4、得到结果，结果去重。

### 出现的问题

出现“org.apache.spark.SparkException: Task not serializable"这个错误，一般是因为在map、filter等的参数使用了外部的变量，但是这个变量不能序列化。特别是当引用了某个类（经常是当前类）的成员函数或变量时，会导致这个类的所有成员（整个类）都需要支持序列化。解决这个问题最常用的方法有：

1. 如果可以，将依赖的变量放到map、filter等的参数内部定义。这样就可以使用不支持序列化的类；

2. 如果可以，将依赖的变量独立放到一个小的class中，让这个class支持序列化；这样做可以减少网络传输量，提高效率；

3. 如果可以，将被依赖的类中不能序列化的部分使用transient关键字修饰，告诉编译器它不需要序列化。

4. 将引用的类做成可序列化的。

5. 以下这两个没试过。。

redis中存放了最新网元的filehash值

创建一个spark任务去读取集群上的文件

### 日志问题：

使用spark-submit启动任务，会在spark的依赖环境下运行，spark使用的是log4j日志实现，springboot默认使用的是logback日志框架会出现冲突，所以我把springboot的日志框架改成的log4j项目就能正常启动了。

### 序列化问题:

使用spring注入的方式有很多问题解决不了，redistemplate通过注入的方式在集群上跑任务的时候一直会报对象没有序列化的问题

首先我就想着对redistemplate进行序列化，构造自己的redisTemplate，构造自己的连接池并将其放到spring容器管理中去

redistemplate里面存在很多没有序列化的组件都需要自己实现序列化。

### 空指针问题：

封装一个工具类来进行redis操作，都是使用静态方法。

使用静态属性通过构造器注入的方式去调用，结果总是会有空指针的问题

中间试了很多种方案，最后通过自己创建redis连接的方式构造redistemplate

# 算法

## # 各种类型初始化

```java

String[] strarr = new String[]{"cat", "bat", "hat", "tree"};

int[] intarr = new int[]{1,0,0,1,0,0,1,0};

char[] chararr=new char[]{'A','A','B','B','C','C','D','D'};

int[][] board = new int[][]{

{8,0},{4,4},{8,1},{5,0},{6,1},{5,2}

};

String str = "this problem is an easy problem";

List<String> list=new ArrayList();

list.add("xxx");

list.add("yyy");

list.add("zzz");

List<String> name = Arrays.asList("xxx","yyy","zzz");

List<String> list = new ArrayList<String>() {{

add("a");

add("b");

}};

Map<String, String> map = new HashMap<>() {{

put("key", "value");

put("key2", "value2");

}};

```

## # 排序

### ## 冒泡排序

算法描述

- 比较相邻的元素。如果第一个比第二个大，就交换它们两个；

- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；

- 针对所有的元素重复以上的步骤，除了最后一个；

- 重复步骤1~3，直到排序完成。

最佳情况：T(n) = O(n) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(n2)

```java

public static void select(int[] arr){

for (int i=0; i<arr.length-1; i++)

{

for (int j=0; j<arr.length-1-i;j++)

{

if (arr[j]>arr[j+1])

{

int tmp=arr[j];

arr[j]=arr[j+1];

arr[j+1]=tmp;

}

}

}

}

```

### ## 插入排序

```java

public static void insertSort(int[] a) {

int i, j, insertNote;// 要插入的数据

for (i = 1; i < a.length; i++) {// 从数组的第二个元素开始循环将数组中的元素插入

insertNote = a[i];// 设置数组中的第2个元素为第一次循环要插入的数据

j = i - 1;

while (j >= 0 && insertNote < a[j]) {

a[j + 1] = a[j];// 如果要插入的元素小于第j个元素,就将第j个元素向后移动

j--;

}

a[j + 1] = insertNote;// 直到要插入的元素不小于第j个元素,将insertNote插入到数组中

}

}

```

### ## 选择排序

表现最稳定的排序算法之一，因为无论什么数据进去都是O(n2)的时间复杂度，所以用到它的时候，数据规模越小越好。唯一的好处可能就是不占用额外的内存空间了吧。理论上讲，选择排序可能也是平时排序一般人想到的最多的排序方法了吧。

选择排序(Selection-sort)是一种简单直观的排序算法。它的工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

```java

public static void selectSort(int[] arr){

for (int i = 0; i < arr.length; i++) {

int tmp = arr[i];

int index=i;

for (int j = i+1; j < arr.length; j++) {

if (arr[j]<tmp) {

tmp=arr[j];

index=j;

}

}

if (index!=i) {

//交换

arr[i]=arr[index];

arr[index]=tmp;

}

}

}

```

### ## 快速排序

通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

最佳情况：T(n) = O(nlogn) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(nlogn)

```java

public static void quickSort(int[] array) {

quickSort(array, 0, array.length - 1);

}

private static void quickSort(int[] array, int left, int right) {

if (array == null || left >= right || array.length <= 1) {

return;

}

int mid = partition(array, left, right);

quickSort(array, left, mid);

quickSort(array, mid + 1, right);

}

private static int partition(int[] array, int left, int right) {

int temp = array[left];

while (right > left) {

// 先判断基准数和后面的数依次比较

while (temp <= array[right] && left < right) {

--right;

}

// 当基准数大于了 arr[left]，则填坑

if (left < right) {

array[left] = array[right];

++left;

}

// 现在是 arr[right] 需要填坑了

while (temp >= array[left] && left < right) {

++left;

}

if (left < right) {

array[right] = array[left];

--right;

}

}

array[left] = temp;

return left;

}

```

### 归并排序（Merge Sort）

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为2-路归并。

算法描述

* 把长度为n的输入序列分成两个长度为n/2的子序列；
* 对这两个子序列分别采用归并排序；
* 将两个排序好的子序列合并成一个最终的排序序列。

function mergeSort(arr) {

    var len = arr.length;

    if(len < 2) {

        return arr;

    }

    Var middle = Math.floor(len / 2),

        left = arr.slice(0, middle),

        right = arr.slice(middle);

    return merge(mergeSort(left), mergeSort(right));

}

function merge(left, right) {

    varresult = [];

    while(left.length>0 && right.length>0) {

        if(left[0] <= right[0]) {

            result.push(left.shift());

        }else{

            result.push(right.shift());

        }

    }

    while(left.length)

        result.push(left.shift());

    while(right.length)

        result.push(right.shift());

    returnresult;

}

## # 查找

### ## 二分法查找

二分查找也称折半查找（Binary Search），它是一种效率较高的查找方法，前提是数据结构必须先排好序，时间复杂度可以表示O(h)=O(log2n)，以2为底，n的对数。其缺点是要求待查表为有序表，且插入删除困难。

```java

public static int search(int[] arr, int key){

int start=0;

int end=arr.length-1;

while (start<=end)

{

int mid = (end+start)/2;

if (key<arr[mid])

{

end=mid-1;

}else if (key>arr[mid]){

start=mid+1;

}else {

return mid;

}

}

return -1;

}

```

### ## 深度优先

```java

public void depthOrderTraversal(TreeNode root) {

if (root == null) {

System.out.println("empty tree");

return;

}

ArrayDeque<TreeNode> stack = new ArrayDeque<TreeNode>();

stack.push(root);

while (!stack.isEmpty()) {

TreeNode node = stack.pop();

System.out.print(node.val + " ");

if (node.right != null) {

stack.push(node.right);

}

if (node.left != null) {

stack.push(node.left);

}

}

System.out.print("\n");

}

```

### ## 广度优先

```java

/\*\*

\* 广度优先遍历

\* 采用非递归实现

\* 需要辅助数据结构：队列

\*/

public void levelOrderTraversal(TreeNode root) {

if (root == null) {

System.out.println("empty tree");

return;

}

ArrayDeque<TreeNode> queue = new ArrayDeque<TreeNode>();

queue.add(root);

while (!queue.isEmpty()) {

TreeNode node = queue.remove();

System.out.print(node.val + " ");

if (node.left != null) {

queue.add(node.left);

}

if (node.right != null) {

queue.add(node.right);

}

}

System.out.print("\n");

}

```

## # 去重

```java

public static void searchRepeatIndex(int[] arr){

for (int i = 0; i < arr.length-1; i++) {

for (int j = i+1; j < arr.length; j++) {

if (arr[i]==arr[j]) {

System.out.println("重复元素下标:"+i);

break;//去掉这句可以查找重复次数

}

}

}

}

```

## # 二叉树的遍历

### ## 先序遍历

```java

/\*\*

\* 递归先序遍历

\* \*/

public void preOrderRecursion(TreeNode node){

if(node==null) //如果结点为空则返回

return;

visit(node);//访问根节点

preOrderRecursion(node.left);//访问左孩子

preOrderRecursion(node.right);//访问右孩子

}

/\*\*

\* 非递归先序遍历二叉树

\* \*/

public List<Integer> preorderTraversal(TreeNode root) {

List<Integer> resultList=new ArrayList<>();

Stack<TreeNode> treeStack=new Stack<>();

if(root==null) //如果为空树则返回

return resultList;

treeStack.push(root);

while(!treeStack.isEmpty()){

TreeNode tempNode=treeStack.pop();

if(tempNode!=null){

resultList.add(tempNode.val);//访问根节点

treeStack.push(tempNode.right); //入栈右孩子

treeStack.push(tempNode.left);//入栈左孩子

}

}

return resultList;

}

```

### ## 中序遍历

```java

/\*\*

\* 递归中序遍历

\* \*/

public void preOrderRecursion(TreeNode node){

if(node==null) //如果结点为空则返回

return;

preOrderRecursion(node.left);//访问左孩子

visit(node);//访问根节点

preOrderRecursion(node.right);//访问右孩子

}

/\*\*

\* 非递归中序遍历

\* \*/

public List<Integer> inorderTraversalNonCur(TreeNode root) {

List<Integer> visitedList=new ArrayList<>();

Map<TreeNode,Integer> visitedNodeMap=new HashMap<>();//保存已访问的节点

Stack<TreeNode> toBeVisitedNodes=new Stack<>();//待访问的节点

if(root==null)

return visitedList;

toBeVisitedNodes.push(root);

while(!toBeVisitedNodes.isEmpty()){

TreeNode tempNode=toBeVisitedNodes.peek(); //注意这里是peek而不是pop

while(tempNode.left!=null){ //如果该节点的左节点还未被访问，则需先访问其左节点

if(visitedNodeMap.get(tempNode.left)!=null) //该节点已经被访问（不存在某个节点已被访问但其左节点还未被访问的情况）

break;

toBeVisitedNodes.push(tempNode.left);

tempNode=tempNode.left;

}

tempNode=toBeVisitedNodes.pop();//访问节点

visitedList.add(tempNode.val);

visitedNodeMap.put(tempNode, 1);//将节点加入已访问map

if(tempNode.right!=null) //将右结点入栈

toBeVisitedNodes.push(tempNode.right);

}

return visitedList;

}

```

### ## 后序遍历

```java

/\*\*

\* 递归中序遍历

\* \*/

public void preOrderRecursion(TreeNode node){

if(node==null) //如果结点为空则返回

return;

preOrderRecursion(node.left);//访问左孩子

preOrderRecursion(node.right);//访问右孩子

visit(node);//访问根节点

}

/\*\*

\* 非递归后序遍历

\* \*/

public List<Integer> postOrderNonCur(TreeNode root){

List<Integer> resultList=new ArrayList<>();

if(root==null)

return resultList;

Map<TreeNode,Integer> visitedMap=new HashMap<>();

Stack<TreeNode> toBeVisitedStack=new Stack<>();

toBeVisitedStack.push(root);

while(!toBeVisitedStack.isEmpty()){

TreeNode tempNode=toBeVisitedStack.peek(); //注意这里是peek而不是pop

if(tempNode.left==null && tempNode.right==null){ //如果没有左右孩子则访问

resultList.add(tempNode.val);

visitedMap.put(tempNode, 1);

toBeVisitedStack.pop();

continue;

}else if(!((tempNode.left!=null&&visitedMap.get(tempNode.left)==null )|| (tempNode.right!=null && visitedMap.get(tempNode.right)==null))){

//如果节点的左右孩子均已被访问

resultList.add(tempNode.val);

toBeVisitedStack.pop();

visitedMap.put(tempNode, 1);

continue;

}

if(tempNode.left!=null){

while(tempNode.left!=null && visitedMap.get(tempNode.left)==null){//左孩子没有被访问

toBeVisitedStack.push(tempNode.left);

tempNode=tempNode.left;

}

}

if(tempNode.right!=null){

if(visitedMap.get(tempNode.right)==null){//右孩子没有被访问

toBeVisitedStack.push(tempNode.right);

}

}

}

return resultList;

}

```

## 链表

### 删除链表倒数第k个节点

import java.util.\*;

public class ListNode {

int val;

ListNode next = null;

}

public class Solution {

public ListNode removeNthFromEnd (ListNode head, int n) {

// write code here

if(head == null){

return null;

}

ListNode p = head;

ListNode q = head;

ListNode pre = null;

while(n>0){

if(p!=null){

p = p.next;

}else{

return null;

}

n--;

}

if(p==null){

return head.next;

}

while(p != null){

p = p.next;

pre = q ;

q = q.next;

if(p ==null){

pre.next = q.next;

}

}

return head;

}

}

### 反转链表

public class Solution {

public ListNode ReverseList(ListNode head) {

//初始化pre指针，用于记录当前结点的前一个结点地址

ListNode pre = null;

//初始化p指针，用于记录当前结点的下一个结点地址

ListNode p = null;

//head指向null时，循环终止。

while(head != null){

//先用p指针记录当前结点的下一个结点地址。

p = head.next;

//让被当前结点与链表断开并指向前一个结点pre。

head.next = pre;

//pre指针指向当前结点

pre = head;

//head指向p(保存着原链表中head的下一个结点地址)

head = p;

}

return pre;//当循环结束时,pre所指的就是反转链表的头结点

}

}

### 链表中环的入口节点

public class Solution{

public ListNode detectCycle(ListNode head){

ListNode slow=head;

ListNode fast=head;

while(fast!=null&&fast.nest!=null){

fast=fast.next.next

slow=slow.next;

if(fast==slow){ //利用快慢指针找相遇点

ListNode slow2=head; //设置以相同速度的新指针从起始位置出发

while(slow2!=slow){ //未相遇循环。

slow=slow.next;

slow2=slow2.next;

}

return slow;

}

}

return null;

}

}

## 二叉树

### 两个节点的公共祖先

public int lowestCommonAncestor (TreeNode root, int o1, int o2) {

return CommonAncestor(root, o1, o2).val;

}

public TreeNode CommonAncestor (TreeNode root, int o1, int o2) {

if (root == null || root.val == o1 || root.val == o2) { // 超过叶子节点，或者root为p、q中的一个直接返回

return root;

}

TreeNode left = CommonAncestor(root.left,o1,o2); // 返回左侧的p\q节点

TreeNode right = CommonAncestor(root.right,o1,o2); // 返回右侧的p\q节点

if (left == null) { // 都在右侧

return right;

}

if (right == null) { // 都在左侧

return left;

}

return root; // 在左右两侧

}

## 字符串

### 最长无重复子串

public int maxLength (int[] arr) {

HashMap<Integer,Integer> map = new HashMap<>();

int max = 1;

for(int start = 0, end = 0; end<arr.length ; end++){

if(map.containsKey(arr[end])){

//重复了

start = Math.max(start, map.get(arr[end])+1);

//注意：这里一定要取最大的start，不然就错误了

//为什么？ 因为重复数字的索引很可能比start小

}

max = Math.max(max , end-start+1);

map.put(arr[end],end);

}

return max;

}

### 最长公共字串

public String LCS(String str1, String str2){

if(str1.length() == 0 || str2.length() == 0){

return "-1";

}

char[] s1 = str1.toCharArray();

char[] s2 = str2.toCharArray();

int start1 = -1;

int start2 = -1;

int[][] dp = new int[s1.length][s2.length];

int max = 0;

for(int i=0; i<s1.length; i++){

dp[i][0] = (s1[i] == s2[0] ? 1 : 0);

for(int j=0; j<s2.length; j++){

dp[0][j] = (s1[0] == s2[j] ? 1 : 0);

if(i > 0 && j > 0){

if(s1[i] == s2[j]){

dp[i][j] = dp[i-1][j-1] + 1;

}

}

if(max < dp[i][j]){

max = dp[i][j];

start1 = i + 1 - max;

start2 = j + 1 - max;

}

}

}

if(max == 0) return "-1";

return str1.substring(start1, max+start1);

}

## LRU缓存实现

LinkedList储存key值，实现最近最少使用。

get：

如果存在该key，则先把LinkedList中原来的key值删除，再把key添加到LinkedList末尾，越最近使用的key越靠近LinkedList末尾。

put：

如果存在该key，也要像get一样，先把LinkedList中原来的key值删除，再把key添加到LinkedList末尾，直接调用HashMap的put方法，新的value值就会覆盖旧的value值。

如果put之前元素个数已经达到了容量，则把LinkedList中第一个元素删除，越是最近最少使用的key越靠近LinkedList头部。然后调用HashMap的put方法。

不存在该key，也没有到达容量，直接调用HashMap的put方法。

class LRUCache {

private int capacity;

private HashMap<Integer,Integer> map;

private LinkedList<Integer> list;

public LRUCache(int capacity) {

this.capacity=capacity;

map=new HashMap<>();

list=new LinkedList<>();

}

public int get(int key) {

if(map.containsKey(key)){

list.remove((Integer)key);

list.addLast(key);

return map.get(key);

}

return -1;

}

public void put(int key, int value) {

if(map.containsKey(key)){

list.remove((Integer)key);

list.addLast(key);

map.put(key,value);

return;

}

if(list.size()==capacity){

map.remove(list.removeFirst());

map.put(key,value);

list.addLast(key);

}

else{

map.put(key,value);

list.addLast(key);

}

}

}

## SQL

### 组合两个表

select FirstName, LastName, City, State

from Person left join Address

on Person.PersonId = Address.PersonId

;

### 第二高的薪水

方法一：使用子查询和 LIMIT 子句

算法

将不同的薪资按降序排序，然后使用 LIMIT 子句获得第二高的薪资。

MySQL

SELECT DISTINCT

Salary AS SecondHighestSalary

FROM

Employee

ORDER BY Salary DESC

LIMIT 1 OFFSET 1

然而，如果没有这样的第二最高工资，这个解决方案将被判断为 “错误答案”，因为本表可能只有一项记录。为了克服这个问题，我们可以将其作为临时表。

MySQL

SELECT

(SELECT DISTINCT

Salary

FROM

Employee

ORDER BY Salary DESC

LIMIT 1 OFFSET 1) AS SecondHighestSalary

;

方法二：使用 IFNULL 和 LIMIT 子句

解决 “NULL” 问题的另一种方法是使用 “IFNULL” 函数，如下所示。

MySQL

SELECT

IFNULL(

(SELECT DISTINCT Salary

FROM Employee

ORDER BY Salary DESC

LIMIT 1 OFFSET 1),

NULL) AS SecondHighestSalary

### 超过经理收入的员工

SELECT

a.Name AS 'Employee'

FROM

Employee AS a,

Employee AS b

WHERE

a.ManagerId = b.Id

AND a.Salary > b.Salary

;

算法

实际上， JOIN 是一个更常用也更有效的将表连起来的办法，我们使用 ON 来指明条件。

SELECT

a.NAME AS Employee

FROM Employee AS a JOIN Employee AS b

ON a.ManagerId = b.Id

AND a.Salary > b.Salary

;

### 查找重复的电子邮箱

方法一：使用 GROUP BY 和临时表

算法

重复的电子邮箱存在多次。要计算每封电子邮件的存在次数，我们可以使用以下代码。

MySQL

select Email, count(Email) as num

from Person

group by Email;

| Email | num |

|---------|-----|

| a@b.com | 2 |

| c@d.com | 1 |

以此作为临时表，我们可以得到下面的解决方案。

MySQL

select Email from

(

select Email, count(Email) as num

from Person

group by Email

) as statistic

where num > 1

;

方法二：使用 GROUP BY 和 HAVING 条件

向 GROUP BY 添加条件的一种更常用的方法是使用 HAVING 子句，该子句更为简单高效。

所以我们可以将上面的解决方案重写为：

MySQL

select Email

from Person

group by Email

having count(Email) > 1;

## 数据结构

红黑树（英语：Red–black tree）是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。

每个节点要么是黑的，要么是红的

根节点是黑的

叶节点是黑的

如果一个节点是红的，他的两个儿子节点都是黑的

对于任一节点而言，其到叶节点树尾端NIL指针的每一条路径都包含相同数目的黑节点