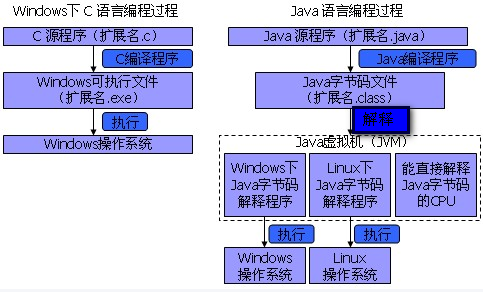
# 第一章 基础篇

## Java基础

### 1.1.1 Java的平台无关性如何体现出来的



1. C/C++语言都直接编译成针对特定平台的机器码。如果要跨平台，需要使用相应的编译器重新编译。
2. Java源程序（.java）要先编译成与平台无关的字节码文件(.class)，然后字节码文件再解释成机器码运行。解释是通过Java虚拟机来执行的。
3. 字节码文件不面向任何具体平台，只面向虚拟机。
4. Java虚拟机是可运行Java字节码文件的虚拟计算机。不同平台的虚拟机是不同的，但它们都提供了相同的接口。
5. Java语言具有一次编译，到处运行的特点。就是说编译后的.class可以跨平台运行，前提是该平台具有相应的Java虚拟机。但是性能比C/C++要低。
6. Java的跨平台原理决定了其性能没有C/C++高

#### 1.1.1.1 JDK和JRE的区别

JRE：Java Runtime Environment（java运行时环境）。即java程序的运行时环境，包含了java虚拟机，java基础类库。

JDK：Java Development Kit（java开发工具包）。即java语言编写的程序所需的开发工具包。

JDK包含了JRE，同时还包括java源码的编译器javac、监控工具jconsole、分析工具jvisualvm等。

### 1.1.2面向对象的特征：继承、封装和多态

**封装：** 原则：隐藏对象的属性和实现细节，仅对外提供公共访问方式。

好处：①将变化隔离。②便于使用。③提高重用性。 ④提高安全性。

**继承：**概述： 多个类中存在相同属性和行为时，将这些内容抽取到单独一个类中，那么多个类无需再定义这些属性和行为，只要继承那个类即可。父类又称基类，超类；子类又称派生类。子类可以直接访问父类中的非私有的属性和行为。关键字为 extends

好处：提高了代码的复用性。让类与类之间产生了关系，提供了多态的前提。

特点：a.Java只支持单继承，不支持多继承。因为如果可以多继承，就会出现调用不明确的问题。 b.Java支持多重继承（继承体系）

**多态：**概述： 在同一个方法中，由于参数不同而导致执行效果各异的现象就是多态。

前提条件：A:要有继承关系。B:要有方法重写。C:要有父类引用指向子类对象。

多态间的成员特点： 方法有重写，而变量没有；成员变量：编译看左边，运行看左边。

成员方法：编译看左边，运行看右边。

优点：提高代码的扩展性和可维护性。

弊端：父类引用不能使用子类特有功能。

注意：多态有三种体现形式： 类多态；抽象类多态；接口多态

基本类型：隐式转换(小到大)，强制转换(大到小)。

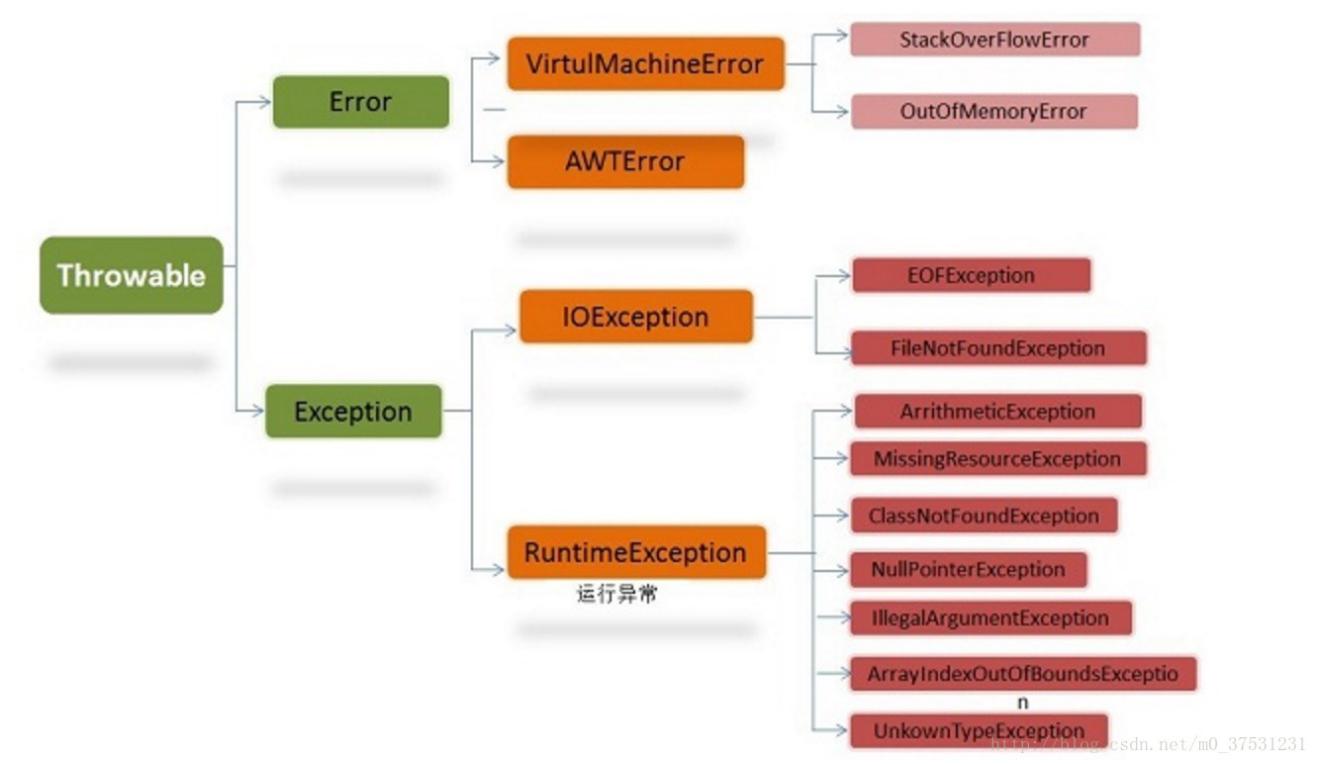
         引用类型：向上转型(小到大)，向下转型(大到小)。

### 1.1.3 final, finally, finalize 的区别

1. final修饰符（关键字）如果一个类被声明为final，意味着它不能再派生出新的子类，不能作为父类被继承。将变量或方法声明为final，可以保证它们在使用中不被改变。被声明为final的变量必须在声明时给定初值，而在以后的引用中只能读取，不可修改。被声明为final的方法也同样只能使用，不能重载。
2. finally在异常处理时提供 finally 块来执行任何清除操作。如果有finally的话，则不管是否发生异常，finally语句都会被执行。

finalize方法名。Java 技术允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。finalize() 方法是在垃圾收集器删除对象之前被调用的。它是在 Object 类中定义的，因此所有的类都继承了它。子类覆盖 finalize() 方法以整理系统资源或者执行其他清理工作。

### 1.1.4 Exception、Error、运行时异常与一般异常有何异同



所有的异常都是由Throwable类，下一层分解为两个分支：Error和Exceprion。

**Error：**层次结构描述了java运行时系统的内部错误和资源耗尽错误。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。应用程序不应该抛出这种类型的对象。如：内存溢出、动态链接异常、虚拟机错误

**Exceprion：**这个层次结构又分解为连个分支：一个分支派生于RuntimeException；另一个分支包含其他异常。划分两个分支的规则是：由程序错误导致的异常属于RuntimeException；而程序本身没有没有问题，但由于像I/O错误这类异常导致的异常属于其他异常。如：所需文件没有找到、零作除数，数组下标越界

**unchecked exception（非检查异常）**：包括运行时异常（RuntimeException）和派生于Error类的异常。对于运行时异常，java编译器不要求必须进行异常捕获处理或者抛出声明，由程序员自行决定。

**checked exception（检查异常，编译异常，必须要处理的异常）**

也：称非运行时异常（运行时异常以外的异常就是非运行时异常），java编译器强制程序员必须进行捕获处理，比如常见的IOExeption和SQLException。对于非运行时异常如果不进行捕获或者抛出声明处理，编译都不会通过。

#### 1.1.4.1 请写出5种常见到的runtime exception

**常见的RuntimeException（运行时异常）**：IndexOutOfBoundsException(下标越界异常)；NullPointerException(空指针异常)；NumberFormatException （String转换为指定的数字类型异常）；ArithmeticException -（算术运算异常 如除数为0）；ArrayStoreException - （向数组中存放与声明类型不兼容对象异常）；SecurityException -（安全异常）；

**IOException（其他异常）**：FileNotFoundException（文件未找到异常。）；IOException（操作输入流和输出流时可能出现的异常。）；EOFException （文件已结束异常）

#### 1.1.4.2 异常处理中throws和throw的区别

1. 作用不同：throw用于程序员自行产生并抛出异常；throws用于声明在该方法内抛出了异常
2. 使用的位置不同：throw位于方法体内部，可以作为单独语句使用；throws必须跟在方法参数列表的后面，不能单独使用。
3. 内容不同：throw抛出一个异常对象，且只能是一个；throws后面跟异常类，而且可以有多个。

#### 1.1.4.3 Java异常处理try-catch-finally的执行过程

try-catch-finally程序块的执行流程以及执行结果比较复杂。基本执行过程如下：

程序首先执行可能发生异常的try语句块。如果try语句没有出现异常则执行完后跳至finally语句块执行；如果try语句出现异常，则中断执行并根据发生的异常类型跳至相应的catch语句块执行处理。catch语句块可以有多个，分别捕获不同类型的异常。catch语句块执行完后程序会继续执行finally语句块。finally语句是可选的，如果有的话，则不管是否发生异常，finally语句都会被执行。

需要注意的是即使try和catch块中存在return语句，finally语句也会执行。是在执行完finally语句后再通过return退出。

#### 1.1.4.4 Checked异常和Runtime异常的区别

1. 运行时异常：包括RuntimeaException及其所有子类。不要求程序必须对它们作出处理，比如InputMismatchException、ArithmeticException、NullPointerException等。即使没有使用try-catch或throws进行处理，仍旧可以进行编译和运行。如果运行时发生异常，会输出异常的堆栈信息并中止程序执行。
2. Checked异常（非运行时异常）：除了运行时异常外的其他异常类都是Checked异常。程序必须捕获或者声明抛出这种异常，否则出现编译错误，无法通过编译。处理方式包括两种：通过try-catch捕获异常，通过throws声明抛出异常从而交给上一级调用方法处理。

### 1.1.5包装类，装箱和拆箱



装箱及拆箱

1）将基本数据类型变成包装类称为装箱。

2）将包装类的类型变为基本数据类型称为拆箱。

#### 1.1.5.1 基本数据类型、引用类型

引用类型（reference type）指向一个对象，不是原始值，指向对象的变量是引用变量

引用类型常见的有：**String，StringBuffer，ArrayList，HashSet，HashMap**等。

|  |  |
| --- | --- |
| 基本数据类型 | 引用数据类型 |
| 在栈中进行分配 | 在堆中进行分配,堆的读写速度远不及栈 |
| 变量名指向具体的数值 | 变量名指向存数据对象的内存地址,即变量名指向hash值 |
| 变量在声明之后java就会立刻分配给他内存空间 | 它以特殊的方式(类似C指针)指向对象实体（具体的值），这类变量声明时不会分配内存，只是存储了一个内存地址 |
| 基本类型之间的赋值是创建新的拷贝 | 对象之间的赋值只是传递引用 |
| “==”和“!=”是在比较值 | “==”和“!=”是在比较两个引用是否相同，需要自己实现equals()方法 |
| 基本类型变量创建和销毁很快 | 类对象需要JVM去销毁 |

#### 1.1.5.2 int 和 Integer 有什么区别，Integer的值缓存范围

**区别：**

（1）Integer是int的包装类；int是基本数据类型；

（2）Integer变量必须实例化后才能使用；int变量不需要；

（3）Integer实际是对象的引用，指向此new的Integer对象；int是直接存储数据值 ；

（4）Integer的默认值是null；int的默认值是0。

**Integer的值缓存范围**：【-128，127】

### 1.1.7 String 类

#### 1.1.7.1 String 类的常用方法

1、length() 字符串的长度

2、charAt() 截取一个字符

3、getChars() 截取多个字符

4、getBytes()替代getChars()的一种方法是将字符存储在字节数组中，该方法即getBytes()。

5、toCharArray()

6、equals()和equalsIgnoreCase() 比较两个字符串

7、regionMatches() 用于比较一个字符串中特定区域与另一特定区域，它有一个重载的形式允许在比较中忽略大小写。

8、startsWith()和endsWith()　　startsWith()方法决定是否以特定字符串开始，endWith()方法决定是否以特定字符串结束

9、equals()和==

equals()方法比较字符串对象中的字符，==运算符比较两个对象是否引用同一实例。

1. compareTo()和compareToIgnoreCase() 比较字符串
2. indexOf()和lastIndexOf()

indexOf() 查找字符或者子串第一次出现的地方。

lastIndexOf() 查找字符或者子串是后一次出现的地方。

1. concat() 连接两个字符串
2. substring()　它有两种形式，第一种是：String substring(int startIndex)

　　　　　　　 第二种是：String substring(int startIndex,int endIndex)

14 、replace() 替换

它有两种形式，第一种形式用一个字符在调用字符串中所有出现某个字符的地方进行替换，第二种形式是用一个字符序列替换另一个字符序列，

15、trim() 去掉起始和结尾的空格

16、valueOf() 转换为字符串

17、toLowerCase() 转换为小写

18、toUpperCase() 转换为大写

19、StringBuffer构造函数

　　StringBuffer定义了三个构造函数：

　　StringBuffer()

　　StringBuffer(int size)

　　StringBuffer(String str)

　　StringBuffer(CharSequence chars)

　　(1)、length()和capacity()　　　　一个StringBuffer当前长度可通过length()方法得到,而整个可分配空间通过capacity()方法得到。

　　(2)、ensureCapacity() 设置缓冲区的大小

　　(3)、setLength() 设置缓冲区的长度

　　(4)、charAt()和setCharAt()

　　(5)、getChars()

　　(6)、append() 可把任何类型数据的字符串表示连接到调用的StringBuffer对象的末尾。

　　(7)、insert() 插入字符串

　　(8)、reverse() 颠倒StringBuffer对象中的字符

　　(9)、delete()和deleteCharAt() 删除字符

　　(10)、replace() 替换

　　(11)、substring() 截取子串

#### 1.1.7.2 String、StringBuilder、StringBuffer（\*\*\*）

1. String类是不可变类，即一旦一个String对象被创建后，包含在这个对象中的字符序列是不可改变的，直至这个对象销毁。
2. StringBuffer类则代表一个字符序列可变的字符串，可以通过append、insert、reverse、setChartAt、setLength等方法改变其内容。一旦生成了最终的字符串，调用toString方法将其转变为String
3. JDK1.5新增了一个StringBuilder类，与StringBuffer相似，构造方法和方法基本相同。不同是StringBuffer是线程安全的，而StringBuilder是线程不安全的，所以性能略高。通常情况下，创建一个内容可变的字符串，应该优先考虑使用StringBuilder

### 1.1.8 重载和重写的区别

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 英文 | 位置不同 | 修饰符 | 返回值 | 方法名 | 参数 | 抛出异常 | 方法体 |
| 重载 | overload | 同一个类中 | 无关 | 无关 | 相同 | 不同 | 无关 | 不同 |
| 重写 | override | 子类和父类间 | 大于等于 | 小于等于 | 相同 | 相同 | 小于等于 | 不同 |

### 1.1.9 抽象类和接口有什么区别

相同点

* 抽象类和接口均包含抽象方法，类必须实现所有的抽象方法，否则是抽象类
* 抽象类和接口都不能实例化，他们位于继承树的顶端，用来被其他类继承和实现

两者的区别主要体现在两方面：语法方面和设计理念方面

语法方面的区别是比较低层次的，非本质的，主要表现在：

* 接口中只能定义全局静态常量，不能定义变量。抽象类中可以定义常量和变量。
* 接口中所有的方法都是全局抽象方法。抽象类中可以有0个、1个或多个，甚至全部都是抽象方法。
* 抽象类中可以有构造方法，但不能用来实例化，而在子类实例化是执行，完成属于抽象类的初始化操作。接口中不能定义构造方法。
* 一个类只能有一个直接父类（可以是抽象类），但可以充实实现多个接口。一个类使用extends来继承抽象类，使用implements来实现接口。

二者的主要区别还是在设计理念上，其决定了某些情况下到底使用抽象类还是接口。

* 抽象类体现了一种继承关系，目的是复用代码，抽象类中定义了各个子类的相同代码，可以认为父类是一个实现了部分功能的“中间产品”，而子类是“最终产品”。父类和子类之间必须存在“is-a”的关系，即父类和子类在概念本质上应该是相同的。
* 接口并不要求实现类和接口在概念本质上一致的，仅仅是实现了接口定义的约定或者能力而已。接口定义了“做什么”，而实现类负责完成“怎么做”，体现了功能（规范）和实现分离的原则。接口和实现之间可以认为是一种“has-a的关系”

### 1.1.10 java的java.util.concurrent包

**1.Callable<V>**

Callable<V>与Runnable类似，理解Callable<V>可以从比较其与Runnable的区别开始：、

**2.Semaphore**

一个计数信号量，主要用于控制多线程对共同资源库访问的限制。

**3.ReentrantLock与Condition**

1).ReentrantLock：可重入互斥锁。2).Condition：此类是同步的条件对象，每个Condition实例绑定到一个ReetrantLock中，以便争用同一个锁的多线程之间可以通过Condition的状态来获取通知。

**4.BlockingQueue**

简单介绍。这是一个阻塞的队列超类接口，concurrent包下很多架构都基于这个队列。

**5.CompletionService**

1).CompletionService是一个接口，用来保存一组异步求解的任务结果集。api的解释是：将新生产的异步任务与已完成的任务结果集分离开来。

2).CompletionService依赖于一个特定的Executor来执行任务。实际就是此接口需要多线程处理一个共同的任务，这些多线程由一个指定的线程池来管理。CompletionService的实现类ExecutorCompletionService。

3).api的官方代码示例参考ExecutorCompletionService类的api（一个通用分制概念的函数）。

4).使用示例：如有时我们需要一次插入大批量数据，那么可能我们需要将1w条数据分开插，异步执行。

5).CompletionService与Callable<V>+Future的对比：

在上面的Callable中说过，Callable+Future能实现任务的分治，但是有个问题就是：不知道call()什么时候完成，需要人为控制等待。

而jdk通过CompetionService已经将此麻烦简化，通过CompletionService将异步任务完成的与未完成的区分开来（正如api的描述），我们只用去取即可。

**6.CountDownLatch**

1).CountDownLatch:api解释：一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。

**7.CyclicBarrier**

1）.一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点。也就是说，这一组线程的执行分几个节点，每个节点往下执行，都需等待其他线程，这就需要这种等待具有循环性。CyclicBarrier在这样的情况下就很有用。

2）.CyclicBarrier与CountDownLacth的区别：

(1).CountDownLacth用于一个线程与一组线程之间的相互等待。常用的就是一个主线程与一组分治线程之间的等待：主线程发号令，一组线程同时执行；一组线程依次执行完，再唤醒主线程继续执行；

CyclicBarrier用于一组线程执行时，每个线程执行有多个节点，每个节点的处理需要相互等待。如：对5个文件进行处理，按行将各个文件数字挑出来合并成一行，排序，并输出到另一个文件，那每次处理都需要等待5个线程读入下一行。（api示例可供参考）

(2).CountDownLacth的处理机制是：初始化一个值N（相当于一组线程有N个），每个线程调用一次countDown()，那么cdLatch减1，等所有线程都调用过countDown()，那么cdLatch值达到0，那么线程从await()处接着玩下执行。

CyclicBarrier的处理机制是：初始化一个值N（相当于一组线程有N个），每个线程调用一次await()，那么barrier加1，等所有线程都调用过await()，那么barrier值达到初始值N，所有线程接着往下执行，并将barrier值重置为0，再次循环下一个屏障；

1. 由（2）可以知道，CountDownLatch只可以使用一次，而CyclicBarrier是可以循环使用的。

**8.Exchanger**

Exchanger可以在对中对元素进行配对和交换的线程的同步点。

参考博客：<https://blog.csdn.net/summer_yuxia/article/details/75005964>

## 1.2 说说反射的用途及实现

反射的核心是JVM在运行时才动态加载类或调用方法/访问属性，它不需要事先（写代码的时候或编译期）知道运行对象是谁。

一、Java反射框架主要提供以下功能：

1.在运行时判断任意一个对象所属的类；

2.在运行时构造任意一个类的对象；

3.在运行时判断任意一个类所具有的成员变量和方法（通过反射甚至可以调用private方法）；

4.在运行时调用任意一个对象的方法

二、主要用途 ：

1、反射最重要的用途就是开发各种通用框架。

三、基本反射功能的实现(反射相关的类一般都在java.lang.relfect包里)：

1、获得Class对象：使用Class类的forName静态方法；直接获取某一个对象的class；调用某个对象的getClass()方法

2、判断是否为某个类的实例： 用instanceof关键字来判断是否为某个类的实例

3、创建实例：使用Class对象的newInstance()方法来创建Class对象对应类的实例。

 先通过Class对象获取指定的Constructor对象，再调用Constructor对象的newInstance()方法来创建实例。

4、获取方法： getDeclaredMethods()

5、获取构造器信息： getDeclaredMethods()；  getMethods()；  getMethod()

6、获取类的成员变量（字段）信息： getFiled: 访问公有的成员变量

     getDeclaredField：所有已声明的成员变量。但不能得到其父类的成员变量

     getFileds和getDeclaredFields用法

7、调用方法： invoke()

8、利用反射创建数组：  Array.newInstance()

四、注意： 由于反射会额外消耗一定的系统资源，因此如果不需要动态地创建一个对象，那么就不需要用反射。

另外，反射调用方法时可以忽略权限检查，因此可能会破坏封装性而导致安全问题。

#### 1.2.0.1 反射的使用场合和作用、及其优缺点

1. 反射的使用场合和作用

使用场合：在编译时根本无法知道该对象或类可能属于哪些类，程序只依靠运行时信息来发现该对象和类的真实信息。

主要作用：通过反射可以使程序代码访问装载到JVM 中的类的内部信息

1. 获取已装载类的属性信息
2. 获取已装载类的方法
3. 获取已装载类的构造方法信息
4. 反射的优点

反射提高了Java程序的灵活性和扩展性，降低耦合性，提高自适应能力。它允许程序创建和控制任何类的对象，无需提前硬编码目标类；反射是其它一些常用语言，如C、C++、Fortran 或者Pascal等都不具备的

Java反射技术应用领域很广，如软件测试、 EJB、JavaBean等；许多流行的开源框架例如Struts、Hibernate、Spring在实现过程中都采用了该技术

1. .反射的缺点

性能问题：使用反射基本上是一种解释操作，用于字段和方法接入时要远慢于直接代码。因此Java反射机制主要应用在对灵活性和扩展性要求很高的系统框架上,普通程序不建议使用。

使用反射会模糊程序内部逻辑：程序人员希望在源代码中看到程序的逻辑，反射等绕过了源代码的技术，因而会带来维护问题。反射代码比相应的直接代码更复杂。

#### 1.2.0.2 JDK动态代理的用法是什么？

使用Proxy和InvocationHandler两个类，并实现invoke方法

### 1.2.1说说自定义注解的场景及实现

**自定义注解使用场景**

1.类属性自动赋值。

2.验证对象属性完整性。

3.代替配置文件功能，像spring基于注解的配置。

4.可以生成文档，像java代码注释中的@see,@param等

**自定义注解的实现：**public @interface MyAnnotation{}

#### 1.2.1.1 Java中元注解有哪些，并分别说一下作用

元注解是指注解的注解，Java中有：

① @Retention: 定义注解的保留策略

② @Target：定义注解的作用目标

③ @Document：说明该注解将被包含在javadoc中

④ @Inherited：说明子类可以继承父类中的该注解

### 1.2.2 HTTP请求

#### 1.2.2.1GET与POST方式的区别

GET在浏览器回退时是无害的，而POST会再次提交请求。

GET产生的URL地址可以被Bookmark，而POST不可以。

GET请求会被浏览器主动cache，而POST不会，除非手动设置。

GET请求只能进行url编码，而POST支持多种编码方式。

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

GET请求在URL中传送的参数是有长度限制的，而POST么有。

对参数的数据类型，GET只接受ASCII字符，而POST没有限制。

GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。

GET参数通过URL传递，POST放在Request body中。

#### 1.2.2.2 TCP协议和UDP协议的比较

TCP和UDP是TCP/IP协议栈中传输层的两个协议，它们使用IP路由功能把数据包发送到目的地，从而为应用程序及应用层协议（包括：HTTP、SMTP、SNMP、FTP和Telnet）提供网络服务。

TCP的server和client之间通信就好比两个人打电话，需要互相知道对方的电话号码，然后开始对话。所以在两者的连接过程中间需要指定端口和地址。

UDP的server和client之间的通信就像两个人互相发信。我只需要知道对方的地址，然后就发信过去。对方是否收到我不知道，也不需要专门对口令似的来建立连接。具体区别如下：

1.TCP是面向连接的传输。UDP是无连接的传输

2.TCP有流量控制、拥塞控制，检验数据数据按序到达，而UDP则相反。

3.TCP的路由选择只发生在建立连接的时候，而UDP的每个报文都要进行路由选择

4.TCP是可靠性传输，他的可靠性是由超时重发机制实现的，而UDP则是不可靠传输

5.UDP因为少了很多控制信息，所以传输速度比TCP速度快

6.TCP适合用于传输大量数据，UDP适合用于传输小量数据

#### 1.2.2.3 Session与Cookie区别

1、Cookie和Session都是会话技术，Cookie是运行在客户端，Session是运行在服务器端。

2、Cookie有大小限制以及浏览器在存cookie的个数也有限制，Session是没有大小限制和服务器的内存大小有关。

3、Cookie有安全隐患，通过拦截或本地文件找得到你的cookie后可以进行攻击。

4、Session是保存在服务器端上会存在一段时间才会消失，如果session过多会增加服务器的压力。

#### 1.2.2.4 什么是Socket编程

**什么是Socket编程**

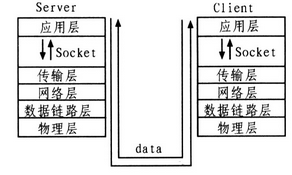
所谓socket通常也称作"套接字"，用于描述IP地址和端口，是一个通信链的句柄。应用程序通常通过"套接字"向网络发出请求或者应答网络请求。

我们开发的网络应用程序位于应用层，TCP和UDP属于传输层协议，在应用层如何使用传输层的服务呢？在应用层和传输层之间，则是使用套接字来进行分离。

套接字就像是传输层为应用层开的一个小口，应用程序通过这个小口向远程发送数据，或者接收远程发来的数据；而这个小口以内，也就是数据进入这个口之后，或者数据从这个口出来之前，是不知道也不需要知道的，也不会关心它如何传输，这属于网络其它层次的工作。

Socket实际是传输层供给应用层的编程接口。传输层则在网络层的基础上提供进程到进程问的逻辑通道，而应用层的进程则利用传输层向另一台主机的某一进程通信。Socket就是应用层与传输层之间的桥梁

使用Socket编程可以开发客户机和服务器应用程序，可以在本地网络上进行通信，也可通过Internet在全球范围内通信。



生活案例1如果你想写封邮件发给远方的朋友，如何写信、将信打包，属于应用层。信怎么写，怎么打包完全由我们做主；而当我们将信投入邮筒时，邮筒的那个口就是套接字，在进入套接字之后，就是传输层、网络层等（邮局、公路交管或者航线等）其它层次的工作了。我们从来不会去关心信是如何从西安发往北京的，我们只知道写好了投入邮筒就OK了。

生活案例2：可以把Socket比作是一个港口码头，应用程序只要将数据交给Socket，就算完成了数据的发送，具体细节由Socket来完成，细节不必了解。同理，对于接收方，应用程序也要创建一个码头，等待数据的到达，并获取数据。

### 1.2.3 java 的浅克隆和深克隆

在Object基类中，有一个方法叫clone，产生一个前期对象的克隆，克隆对象是原对象的拷贝，由于引用类型的存在，有深克隆和浅克隆之分，若克隆对象中存在引用类型的属性，**深克隆**会将此属性完全拷贝一份，而**浅克隆**仅仅是拷贝一份此属性的引用。

·clone方法是Object类的，并不是Cloneable接口的，Cloneable只是一个标记接口，标记接口是用用户标记实现该接口的类具有某种该接口标记的功能，常见的标记接口有三个：Serializable、Cloneable、RandomAccess，没有实现Cloneable接口，那么调用clone方法就会爆出CloneNotSupportedException异常。

·Object类中的clone方法是protected修饰的，这就表明我们在子类中不重写此方法，就在子类外无法访问，因为这个protected权限是仅仅能在Object所在的包和子类能访问的，这也验证了子类重写父类方法权限修饰符可以变大但不能变小的说法。

·重写clone方法，内部仅仅是调用了父类的clone方法，其实是为了扩大访问权限，当然你可以把protected改为public，以后再继承就不用重写了。当然只是浅克隆的clone函数，深克隆就需要修改了。

·属性是String的情况，String也是一个类，那String引用类型吗？String的表现有的像基本类型，归根到底就是因为String不可改变，克隆之后俩个引用指向同一个String，但当修改其中的一个，改的不是String的值，却是新生成一个字符串，让被修改的引用指向新的字符串。外表看起来就像基本类型一样。

### 1.2.4列出自己常用的JDK包

1:java.lang

2:java.sql

3:java.io

4:java.math

5:java.text

6:java.util

7:java.net

8:java.awt

9:java.applet

10:java.nio

1.2.4.1

### 1.2.7 equals与==的区别

==：

a) 基本类型，比较的是值

b) 引用类型，比较的是地址

c) 不能比较没有父子关系的两个对象

equals()

a) 系统类一般已经覆盖了equals()，比较的是内容。

b) 用户自定义类如果没有覆盖equals()，将调用父类的equals（比如是Object），而Object的equals的比较是地址（return (this == obj);）

c) 用户自定义类需要覆盖父类的equals()

注意：Object的==和equals比较的都是地址，作用相同

#### 1.2.7.1 hashCode和equals方法的区别与联系

hashCode()方法和equal()方法的作用其实一样，在Java里都是用来对比两个对象是否相等；

（1）equal()相等的两个对象他们的hashCode()肯定相等，也就是用equal()对比是绝对可靠的；

（2）hashCode()相等的两个对象他们的equal()不一定相等，也就是hashCode()不是绝对可靠的。

对于需要大量并且快速的对比的话如果都用equal()去做显然效率太低，所以解决方式是，每当需要对比的时候，首先用hashCode()去对比，如果hashCode()不一样，则表示这两个对象肯定不相等（也就是不必再用equal()去再对比了）,如果hashCode()相同，此时再对比他们的equal()，如果equal()也相同，则表示这两个对象是真的相同了，这样既能大大提高了效率也保证了对比的绝对正确性！

然而hashCode()和equal()一样都是基本类Object里的方法，而和equal()一样，Object里hashCode()里面只是返回当前对象的地址，如果是这样的话，那么我们相同的一个类，new两个对象，由于他们在内存里的地址不同，则他们的hashCode（）不同，所以这显然不是我们想要的，所以我们必须重写我们类的hashCode()方法.

### 1.2.8什么是Java序列化和反序列化，如何实现Java序列化？或者请解释Serializable 接口的作用

1,序列化和反序列化的概念

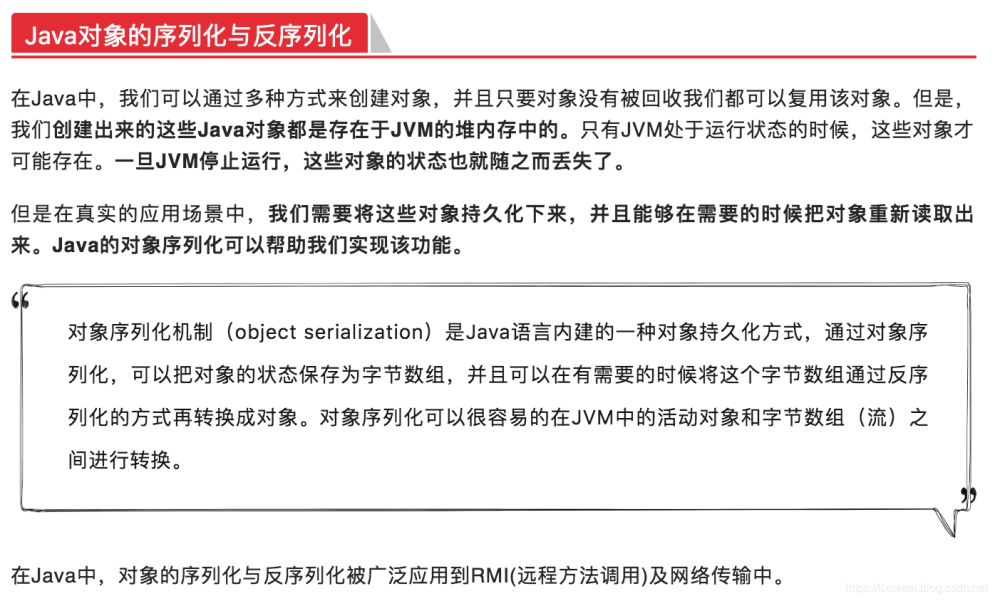
序列化：把对象转换为字节序列的过程称为对象的序列化。

反序列化：把字节序列恢复为对象的过程称为对象的反序列化。

2.如何实现java序列化

实现Serializable接口即可

1. Serializable 接口的作用



### 1.2.9 Object类中常见的方法，为什么wait notify会放在Object里边？

1. public boolean **equals**(java.lang.Object) 比较内容
2. public native int hashCode() 哈希码
3. public java.lang.String toString() 变成字符串
4. public final native java.lang.Class getClass() 获取类结构信息
5. protected void finalize() throws java.lang.Throwable 垃圾回收前执行的方法
6. protected native Object clone() throws java.lang.CloneNotSupportedException 克隆
7. public final void wait() throws java.lang.InterruptedException 多线程中等待功能
8. public final native void notify() 多线程中唤醒功能
9. public final native void notifyAll() 多线程中唤醒所有等待线程的功能

**为什么wait notify会放在Object里边？**

简单说：因为synchronized中的这把锁可以是任意对象，所以任意对象都可以调用wait()和notify()；所以wait和notify属于Object。

专业说：因为这些方法在操作同步线程时，都必须要标识它们操作线程的锁，只有同一个锁上的被等待线程，可以被同一个锁上的notify唤醒，不可以对不同锁中的线程进行唤醒。

也就是说，等待和唤醒必须是同一个锁。而锁可以是任意对象，所以可以被任意对象调用的方法是定义在object类中。

### 1.3.1 Java 8有哪些新特性

(1)Lambda 表达式，也可称为闭包，它是推动 Java 8 发布的最重要新特性。Lambda 允许把函数作为一个方法的参数（函数作为参数传递进方法中）。使用Lambda 表达式可以使代码变的更加简洁紧凑。

(2) 方法引用通过方法的名字来指向一个方法。方法引用可以使语言的构造更紧凑简洁，减少冗余代码。方法引用使用一对冒号 :: 。

(3) 函数式接口(FunctionalInterface)就是一个有且仅有一个抽象方法，但是可以有多个非抽象方法的接口。函数式接口可以被隐式转换为lambda表达式。函数式接口可以现有的函数友好地支持 lambda。

(4) Java 8 新增了接口的默认方法。简单说，默认方法就是接口可以有实现方法，而且不需要实现类去实现其方法。我们只需在方法名前面加个default关键字即可实现默认方法。

(5) Java 8 API添加了一个新的抽象称为流Stream，可以让你以一种声明的方式处理数据。Stream使用一种类似用SQL语句从数据库查询数据的直观方式来提供一种对Java集合运算和表达的高阶抽象。Stream API可以极大提高Java程序员的生产力，让程序员写出高效率、干净、简洁的代码。

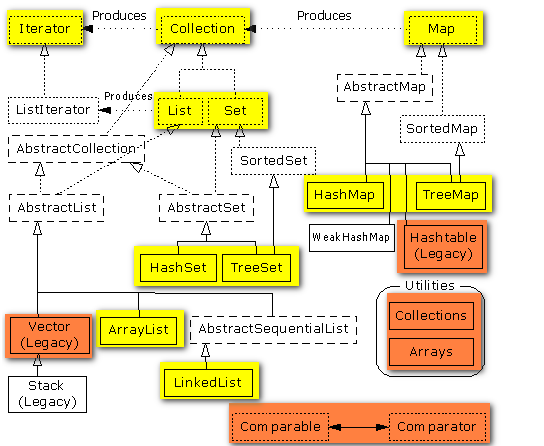
(6) Optional 类是一个可以为null的容器对象。如果值存在则isPresent()方法会返回true，调用get()方法会返回该对象。Optional 是个容器：它可以保存类型T的值，或者仅仅保存null。Optional提供很多有用的方法，这样我们就不用显式进行空值检测。Optional 类的引入很好的解决空指针异常。

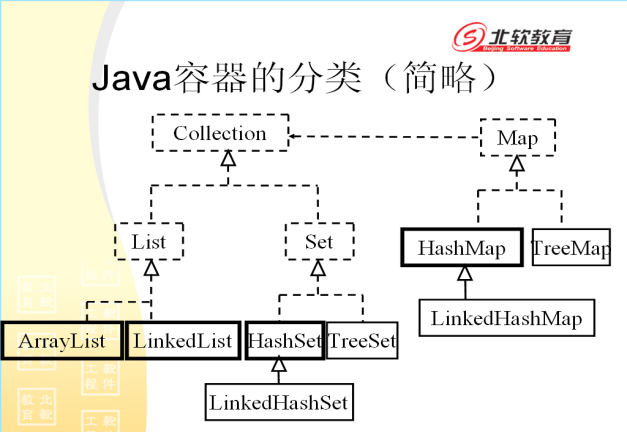
(7) jjs是个基于Nashorn引擎的命令行工具。它接受一些JavaScript源代码为参数，并且执行这些源代码。

(8) Java 8通过发布新的Date-Time API (JSR 310)来进一步加强对日期与时间的处理。

(9) 在Java8中，Base64编码已经成为Java类库的标准。Java 8 内置了 Base64 编码的编码器和解码器。

## 1.3 Java常见集合





Collection 接口存储一组不唯一，无序的对象

List 接口存储一组不唯一，有序（插入顺序）的对象

Set 接口存储一组唯一，无序的对象

Map接口存储一组键值对象，提供key到value的映射。Key无序，唯一。value不要求有序，允许重复。（如果只使用key存储，而不使用value，那就是Set）

### 1.3.1 List 和 Set 区别

1、List,Set都是继承自Collection接口

2、List特点：元素有放入顺序，元素可重复 ，Set特点：元素无放入顺序，元素不可重复，重复元素会覆盖掉，（元素虽然无放入顺序，但是元素在set中的位置是有该元素的HashCode决定的，其位置其实是固定的，加入Set 的Object必须定义equals()方法 ，另外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。）

3.Set和List对比：

Set：检索元素效率低下，删除和插入效率高，插入和删除不会引起元素位置改变。

List：和数组类似，List可以动态增长，查找元素效率高，插入删除元素效率低，因为会引起其他元素位置改变。

### 1.3.2 HashMap和TreeMap区别

HashMap：①数组加链表方式存储key/value, ②线程非安全，③允许null作为key和value，④key不可以重复，value允许重复，⑤不保证元素迭代顺序是按照插入时的顺序；

TreeMap：①基于红黑树，②线程非安全，③不允许null作为key，④key不可以重复，value允许重复，⑤存入TreeMap的元素应当实现Comparable接口或者实现Comparator接口才能按照排序后的顺序遍历元素；hashMap扩容（元素个数>阈值（即当前数组的长度） 乘以 加载因子（HashMap默认的加载因子是0.75））

### 1.3.3 Arraylist 与 LinkedList 区别

**ArrayList和LinkedList的区别和联系**

ArrayList实现了长度可变的数组，在内存中分配连续空间。遍历元素和随机访问元素的效率比较高；

Snap4

LinkedList采用链表存储方式。插入、删除元素时效率比较高

Snap7

### 1.3.4 ArrayList 与 Vector 区别

**Vector和ArrayList的区别和联系**

实现原理相同，功能相同，都是长度可变的数组结构，很多情况下可以互用

两者的主要区别如下

1. Vector是早期JDK接口，ArrayList是替代Vector的新接口
2. Vector线程安全，ArrayList重速度轻安全，线程非安全
3. 长度需增长时，Vector默认增长一倍，ArrayList增长50%

### 1.3.5 HashMap 和 Hashtable 的区别

实现原理相同，功能相同，底层都是哈希表结构，查询速度快，在很多情况下可以互用

两者的主要区别如下

1. Hashtable是早期JDK提供的接口，HashMap是新版JDK提供的接口
2. Hashtable继承Dictionary类，HashMap实现Map接口
3. Hashtable线程安全，HashMap线程非安全
4. Hashtable不允许null值，HashMap允许null值

### 1.3.6 HashSet 和 HashMap 区别

|  |  |
| --- | --- |
| HashMap | HashSet |
| 实现了Map接口 | 实现Set接口 |
| 存储键值对 | 仅存储对象 |
| 调用put（）向map中添加元素 | 调用add（）方法向Set中添加元素 |
| HashMap使用键（Key）计算Hashcode | HashSet使用成员对象来计算hashcode值，  对于两个对象来说hashcode可能相同，  所以equals()方法用来判断对象的相等性，  如果两个对象不同的话，那么返回false |
| HashMap相对于HashSet较快，因为它是使用唯一的键获取对象 | HashSet较HashMap来说比较慢 |

### 1.3.7 HashMap \HashTable\ConcurrentHashMap 的区别

HashTable

底层数组+链表实现，无论key还是value都不能为null，线程安全，实现线程安全的方式是在修改数据时锁住整个HashTable，效率低，ConcurrentHashMap做了相关优化

初始size为11，扩容：newsize = olesize\*2+1

计算index的方法：index = (hash & 0x7FFFFFFF) % tab.length

HashMap

底层数组+链表实现，可以存储null键和null值，线程不安全

初始size为16，扩容：newsize = oldsize\*2，size一定为2的n次幂

扩容针对整个Map，每次扩容时，原来数组中的元素依次重新计算存放位置，并重新插入

插入元素后才判断该不该扩容，有可能无效扩容（插入后如果扩容，如果没有再次插入，就会产生无效扩容）

当Map中元素总数超过Entry数组的75%，触发扩容操作，为了减少链表长度，元素分配更均匀

计算index方法：index = hash & (tab.length – 1)

ConcurrentHashMap

底层采用分段的数组+链表实现，线程安全

通过把整个Map分为N个Segment，可以提供相同的线程安全，但是效率提升N倍，默认提升16倍。(读操作不加锁，由于HashEntry的value变量是 volatile的，也能保证读取到最新的值。)

Hashtable的synchronized是针对整张Hash表的，即每次锁住整张表让线程独占，ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术

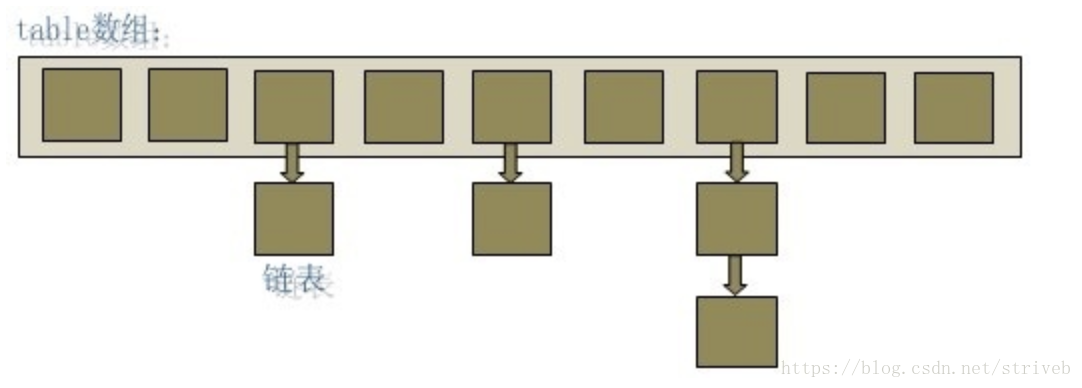
有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁

扩容：段内扩容（段内元素超过该段对应Entry数组长度的75%触发扩容，不会对整个Map进行扩容），插入前检测需不需要扩容，有效避免无效扩容

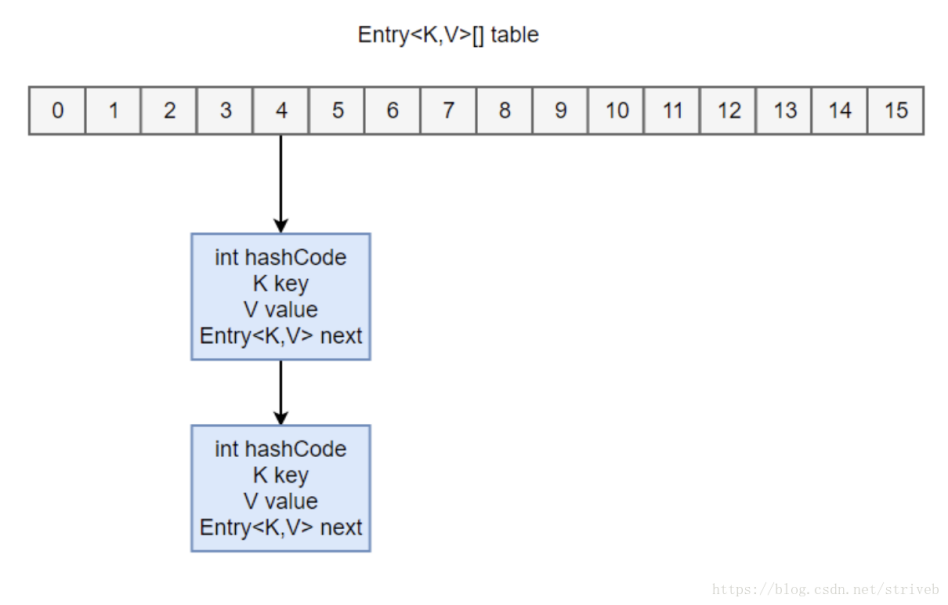
### 1.3.8 HashMap 的工作原理及代码实现，什么时候用到红黑树

hashmap是无序的，因为每次根据 key 的 hashcode 映射到 Entry 数组上，所以遍历出来的顺序并不是写入的顺序

HashMap 底层是基于数组和链表实现的，如图所示，其中两个重要的参数：容量和负载因子；容量的默认大小是 16，负载因子是 0.75，当 HashMap 的 size > 16\*0.75 时就会发生扩容(容量和负载因子都可以自由调整)。

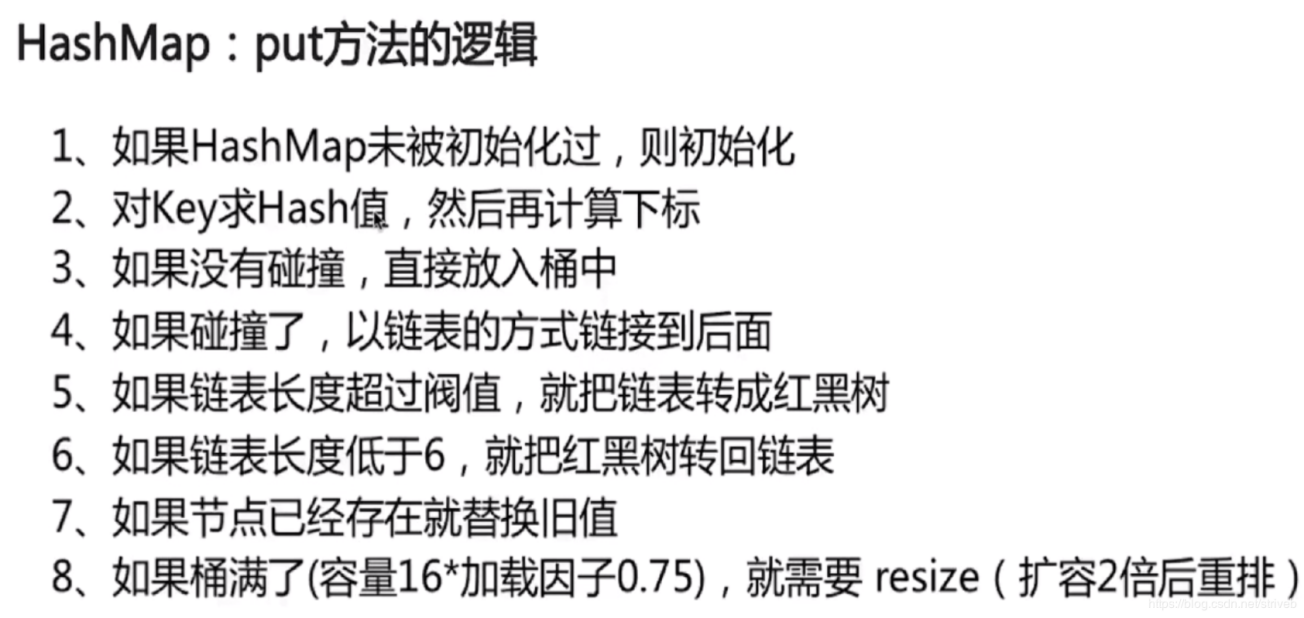


内部包含了一个 Entry 类型的数组 table。HashMap的主干是一个Entry数组。Entry是HashMap的基本组成单元，每一个Entry包含一个key-value键值对。



简单来说，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，如果定位到的数组位置不含链表（当前entry的next指向null）,那么对于查找，添加等操作很快，仅需一次寻址即可；如果定位到的数组包含链表，对于添加操作，其时间复杂度为O(n)，首先遍历链表，存在即覆盖，否则新增；对于查找操作来讲，仍需遍历链表，然后通过key对象的equals方法逐一比对查找。所以，性能考虑，HashMap中的链表出现越少，性能才会越好。

JDK 1.8



### 1.3.9 多线程情况下HashMap死循环的问题

当table[]数组容量较小，容易产生哈希碰撞，所以，Hash表的尺寸和容量非常的重要。一般来说，Hash表这个容器当有数据要插入时，都会检查容量有没有超过设定的thredhold，如果超过，需要增大Hash表的尺寸，这个过程称为resize。

多个线程同时往HashMap添加新元素时，多次resize会有一定概率出现死循环，因为每次resize需要把旧的数据映射到新的哈希表，这一部分代码在HashMap#transfer() 方法，如下：

|  |
| --- |
| void transfer(Entry[] newTable)  {  Entry[] src = table;  int newCapacity = newTable.length;  //下面这段代码的意思是：  // 从OldTable里摘一个元素出来，然后放到NewTable中  for (int j = 0; j < src.length; j++) {  Entry<K,V> e = src[j];  if (e != null) {  src[j] = null;  do {  Entry<K,V> next = e.next;  int i = indexFor(e.hash, newCapacity);  e.next = newTable[i];  newTable[i] = e;  e = next;  } while (e != null);  }  }  } |

### 1.3.10 HashMap出现Hash DOS攻击的问题

我们拿到json格式的数据之后都需要做jsonDecode(),将json串转换为json对象，而对象默认会存储于Hash Table，而Hash Table很容易被碰撞攻击。我只要将攻击数据放在json中，服务端程序在做jsonDecode()时必定中招

要想防御Hash Collision Dos攻击，行业内已经有很多成熟的方案了，不过都是建议换语言或者重写HashTable。这里只说当前json格式解析的问题。首先我们需要增加权限验证，最大可能的在jsonDecode()之前把非法用户拒绝。其次在jsonDecode()之前做数据大小与参数白名单验证。旧项目的改造与维护成本如果很高，建议自己重写jsonDecode()方法。

ConcurrentHashMap 的工作原理及代码实现，如何统计所有的元素个数

### 1.3.11 Concurrent并发HashMap并发能力为什么好于Hashtable？

①Hashtable是通过对hash表整体进行锁定，是阻塞式的，当一个线程占有这个锁时，其他线程必须阻塞等待其释放锁

而ConcurrentHashMap是如下实现：

②jdk1.6的实现：ConcurrentHashMap是采用Segment分段锁的方式，它并没有对整个数据结构进行锁定，而是局部锁定，

③jdk1.8的实现： 采用一种乐观锁CAS算法来实现同步问题，但其底层还是“数组+链表->红黑树”的实现

## 1.4 进程和线程

### 1.4.1线程和进程的概念、并行和并发的概念

1.定义：

1. 进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，是系统进行资源分配和调度的一个独立单位。
2. 线程是进程的一个实体，是CPU调度和分派的基本单位，他是比进程更小的能独立运行的基本单位，线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源（如程序计数器，一组寄存器和栈），一个线程可以创建和撤销另一个线程；

2.进程和线程的关系：

（1）一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。

（2）资源分配给进程，同一进程的所有线程共享该进程的所有资源。

（3）线程在执行过程中，需要协作同步。不同进程的线程间要利用消息通信的办法实现同步。

（4）处理机分给线程，即真正在处理机上运行的是线程。

（5）线程是指进程内的一个执行单元，也是进程内的可调度实体。

3.线程与进程的区别：

（1）调度：线程作为调度和分配的基本单位，进程作为拥有资源的基本单位。

（2）并发性：不仅进程之间可以并发执行，同一个进程的多个线程之间也可以并发执行。

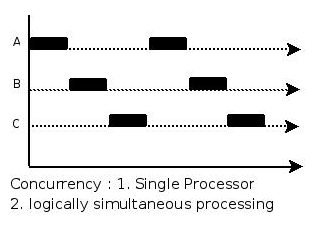
（3）拥有资源：进程是拥有资源的一个独立单位，线程不拥有系统资源，但可以访问隶属于进程的资源。

（4）系统开销：在创建或撤销进程的时候，由于系统都要为之分配和回收资源，导致系统的明显大于创建或撤销线程时的开销。但进程有独立的地址空间，进程崩溃后，在保护模式下不会对其他的进程产生影响，而线程只是一个进程中的不同的执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但是在进程切换时，耗费的资源较大，效率要差些

并行：多个线程可以同时执行，每一个时间段，可以有多个线程同时执行。



并发：多个线程同时竞争一个位置，竞争到的才可以执行，每一个时间段只有一个线程在执行。



#### 1.4.1.1什么是用户线程和守护线程，两者有什么区别？举例说出一个jvm中守护线程的例子

① 用户线程即运行在前台的线程，而守护线程是运行在后台的线程。 ②守护线程作用是为其他前台线程的运行提供便利服务，而且仅在普通、非守护线程仍然运行时才需要，③比如垃圾回收线程就是一个守护线程

### 1.4.2创建线程的方式及实现

#### 1.4.2.1 方式及实现

方式1：继承Java.lang.Thread类，并覆盖run() 方法。优势：编写简单；劣势：无法继承其它父类

public class ThreadDemo1 {

public static void main(String args[]) {

MyThread1 t = new MyThread1();

t.start();

while (true) {

System.out.println("兔子领先了，别骄傲");

}

}

}

class MyThread1 extends Thread {

public void run() {

while (true) {

System.out.println("乌龟领先了，加油");

}

}

}

方式2：实现Java.lang.Runnable接口，并实现run()方法。优势：可继承其它类，多线程可共享同一个Thread对象；劣势：编程方式稍微复杂，如需访问当前线程，需调用Thread.currentThread()方法

public class ThreadDemo2 {

public static void main(String args[]) {

MyThread2 mt = new MyThread2();

Thread t = new Thread(mt);

t.start();

while (true) {

System.out.println("兔子领先了，加油");

}

}

}

class MyThread2 implements Runnable {

public void run() {

while (true) {

System.out.println("乌龟超过了，再接再厉");

}

}

}

#### 1.4.2.2 如何实现线程同步

当多个线程访问同一个数据时，容易出现线程安全问题，需要某种方式来确保资源在某一时刻只被一个线程使用。需要让线程同步，保证数据安全

线程同步的实现方案：**同步代码块和同步方法，均需要使用synchronized关键字**

同步代码块：public void makeWithdrawal(int amt) {

synchronized (acct) { }

}

同步方法：public synchronized void makeWithdrawal(int amt) { }

线程同步的好处：解决了线程安全问题

线程同步的缺点：性能下降，可能会带来死锁

### 1.4.3进程间通信的方式

一、管道

管道，通常指无名管道，是 UNIX 系统IPC最古老的形式。

1、特点：

它是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端。

它只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间）。

它可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

二、FIFO

FIFO，也称为命名管道，它是一种文件类型。

1、特点

FIFO可以在无关的进程之间交换数据，与无名管道不同。

FIFO有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。

三、消息队列

消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列ID）来标识。

1、特点

消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级。

消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除。

消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取。

四、信号量

信号量（semaphore）与已经介绍过的 IPC 结构不同，它是一个计数器。信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据。

1、特点

信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。

信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。

每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

支持信号量组。

五、共享内存

共享内存（Shared Memory），指两个或多个进程共享一个给定的存储区。

1、特点

共享内存是最快的一种 IPC，因为进程是直接对内存进行存取。

因为多个进程可以同时操作，所以需要进行同步。

信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问。

### 1.4.4说说 CountDownLatch、CyclicBarrier 原理和区别

1、概念

  CountDownLatch:具有计数器的功能，等待其他线程执行完毕，主线程在继续执行，用于监听某些初始化操作，并且线程进行阻塞，等初始化执行完毕后，通知主线程继续工作执行。值得注意的是CountDownLatch计数的次数一定要与构造器传入的数字一致，比如构造器传入的是3，则countDown()一定要执行3次，否则线程将一直阻塞。CountDownLatch通常用来控制线程等待，它可以让线程等待倒计时结束，再开始执行。

  CyclicBrrier：翻译过来就是循环栅栏的意思，其作用就是多线程的进行阻塞，等待某一个临界值条件满足后，同时执行！假设有一个场景：每个线程代表一个跑步运动员，当运动员都准备好后，才一起出发，只要有一个人没有准备好，大家都等待！

2、区别

 CountDownLatch: 一个线程(或者多个)， 等待另外N个线程完成某个事情之后才能执行。

 CyclicBrrier: N个线程相互等待，任何一个线程完成之前，所有的线程都必须等待。

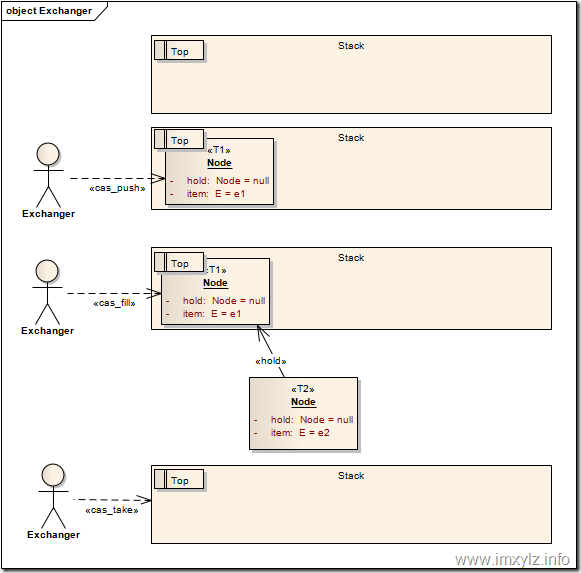
### 1.4.5说说 Semaphore 原理

Semaphore 直译为信号。实际上 Semaphore 可以看做是一个信号的集合。不同的线程能够从 Semaphore 中获取若干个信号量。当 Semaphore 对象持有的信号量不足时，尝试从 Semaphore 中获取信号的线程将会阻塞。直到其他线程将信号量释放以后，阻塞的线程会被唤醒，重新尝试获取信号量。

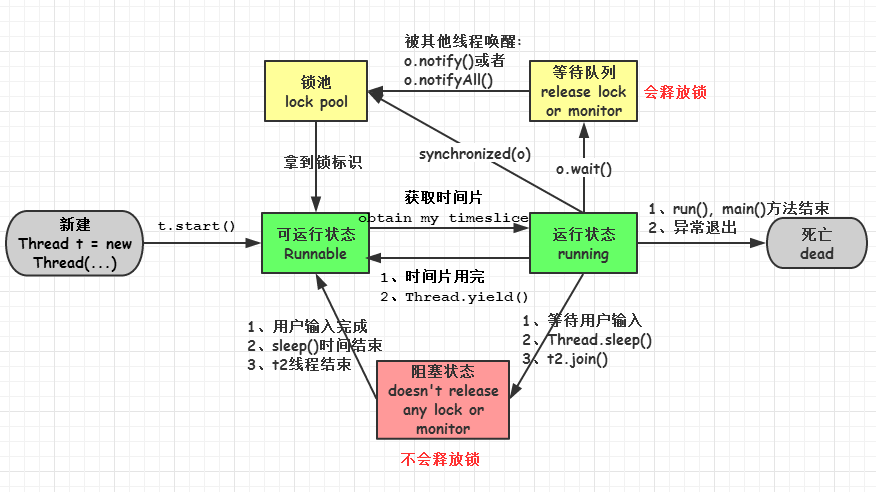
### 1.4.6说说 Exchanger 原理

Exchanger是自jdk1.5起开始提供的工具套件，一般用于两个工作线程之间交换数据。

Exchanger提供的是一个交换服务，允许原子性的交换两个（多个）对象，但同时只有一对才会成功



### 1.4.7 线程的生命周期，状态是如何转移的



1.新建(new)：新创建了一个线程对象。

2.可运行(runnable)：线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取cpu的使用权。

3.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。

4.阻塞(block)：阻塞状态是指线程因为某种原因放弃了cpu使用权，也即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有 机会再次获得cpu timeslice转到运行(running)状态。

阻塞的情况分三种：

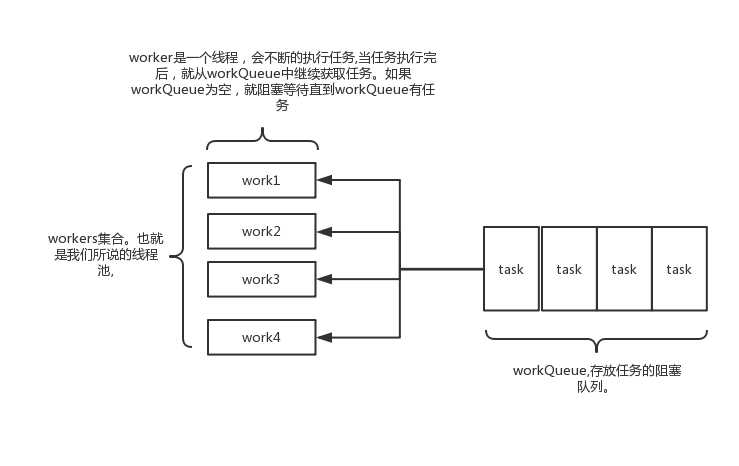
(一).等待阻塞：运行(running)的线程执行o.wait()方法，JVM会把该线程放 入等待队列(waitting queue)中。

(二).同步阻塞：运行(running)的线程在获取对象的同步锁时，若该同步锁 被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。

(三).其他阻塞:运行(running)的线程执行Thread.sleep(long ms)或t.join()方法，或者发出了I/O请求时JVM会把该线程置为阻塞状态。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，线程重新转入可运行(runnable)状态。

5.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。

### 1.4.8讲讲线程池的实现原理以及实现方式



**原理：** 多线程技术主要解决处理器单元内多个线程执行的问题，它可以显著减少处理器单元的闲置时间，增加处理器单元的吞吐能力.

就是一个线程集合workerSet和一个阻塞队列workQueue。当用户向线程池提交一个任务(也就是线程)时，线程池会先将任务放入workQueue中。workerSet中的线程会不断的从workQueue中获取线程然后执行。当workQueue中没有任务的时候，worker就会阻塞，直到队列中有任务了就取出来继续执行。

**实现方式：**

|  |
| --- |
| 线程池的参数如下： ① corePoolSize，线程池里最小线程数 ② maximumPoolSize，线程池里最大线程数量，超过最大线程时候会使用RejectedExecutionHandler ③ keepAliveTime，线程最大的存活时间 ④ workerQueue，缓存异步任务的队列 ⑤ threadFactory，用来构造线程池里的worker线程 |

public static ExecutorService newFixedThreadPool(int nThreads) 创建固定数目线程的线程池。

public static ExecutorService newCachedThreadPool() 创建一个可缓存的线程池，调用execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。

public static ExecutorService newSingleThreadExecutor() 创建一个单线程化的Executor。

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) 创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。

#### 1.4.8.1 JDK线程池的工作流程

工作流程如下：

① 如果正在运行的线程数量小于 corePoolSize，那么马上创建线程运行这个任务

② 如果正在运行的线程数量大于或等于 corePoolSize，那么将这个任务放入队列

③ 如果这时候队列满了，而且正在运行的线程数量小于 maximumPoolSize，那么要创建非核心线程立刻运行这个任务

④ 如果队列满了，而且正在运行的线程数量大于或等于 maximumPoolSize，那么线程池会抛出异常RejectExecutionException

### 1.4.9 线程池的同步和异步，实现机制（未完成）

同步，要求返回结果；

异步，不关心返回结果；

### 1.4.9锁机制

#### 1.4.9.1说说线程安全问题，什么是线程安全，如何保证线程安全

线程安全：就是多线程访问同一代码，不会产生不确定结果。（比如死锁）

如何保证呢：1使用线程安全的类

2使用synchronized同步代码块，或者用Lock锁

3多线程并发情况下，线程共享的变量改为方法局部级变量

**线程安全的三种方式：**（https://blog.csdn.net/q669239799/article/details/90614077）

第一种 : 互斥同步

第二种方法就是：非阻塞同步

第三种：无同步方案

#### 1.4.9.2重入锁的概念，重入锁为什么可以防止死锁

**概念：**自己可以获得自己的内部锁，比如一条线程获得某个对象的锁，此时这个对象锁还没有释放，当期再次想要获得这个对象锁的时候，还是可以获得的，如果不可重入锁则很有可能死锁。

**重入锁为什么可以防止死锁：**

如果一个线程有子类对象的引用loggingWidget，然后调用loggingWidget.doSomething方法的时候，会请求子类对象loggingWidget 的对象锁；又因为loggingWidget 的doSomething方法中调用的父类的doSomething方法，实际上还是要请求子类对象loggingWidget 的对象锁，那么如果**synchronized** 关键字不是个可重入锁的话，就会在子类对象持有的父类doSomething方法上产生死锁了。正因为**synchronized** 关键字的可重入锁，当前线程因为已经持有了子类对象loggingWidget 的对象锁，后面再遇到请求loggingWidget 的对象锁就可以畅通无阻地执行同步方法了。

**重入锁的种类：**

①ReentrantLock

②Synchronized

#### 1.4.9.3产生死锁的四个条件（互斥、请求与保持、不剥夺、循环等待）

**互斥条件**：资源是独占的且排他使用，进程互斥使用资源，即任意时刻一个资源只能给一个进程使用，其他进程若申请一个资源，而该资源被另一进程占有时，则申请者等待直到资源被占有者释放。

**不可剥夺条件**：进程所获得的资源在未使用完毕之前，不被其他进程强行剥夺，而只能由获得该资源的进程资源释放。

**请求和保持条件**：进程每次申请它所需要的一部分资源，在申请新的资源的同时，继续占用已分配到的资源。

**循环等待条件**：在发生死锁时必然存在一个进程等待队列{P1,P2,…,Pn},其中P1等待P2占有的资源，P2等待P3占有的资源，…，Pn等待P1占有的资源，形成一个进程等待环路，环路中每一个进程所占有的资源同时被另一个申请，也就是前一个进程占有后一个进程所深情地资源。

#### 1.4.9.4 如何检查死锁（通过jConsole检查死锁）

Jconsole是JDK自带的图形化界面工具，使用JDK给我们的的工具JConsole，可以通过打开cmd然后输入jconsole打开。基于图形化界面进行线程信息的查看与编辑，从而释放死锁。

（https://www.cnblogs.com/flyingeagle/articles/6853167.html）

#### 1.4.9.5 volatile 实现原理（禁止指令重排、刷新内存）

volatile具有①可见性（**指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。**）

②有序性（禁止指令重排）（**即程序执行的顺序按照代码的先后顺序执行。**），

③ 但不具备原子性（**即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。**）

java编程语言允许线程访问共享变量，为了确保共享变量能被准确和一致的更新，线程应该确保通过排他锁单独获得这个变量。Java语言提供了volatile，在某些情况下比锁更加方便。如果一个字段被声明成volatile，**java线程内存模型确保所有线程看到这个变量的值是一致的。**

#### 1.4.9.6 synchronized 实现原理（对象监视器）

一、synchronized是java中同步的一个关键字它是一个重量级锁，它可以修饰方法和代码块

1. synchronized修饰非静态方法时，它锁住的是当前实例。

2. synchronized修饰静态方法时，它锁住的是整个类（.class）。

3. synchronized修饰代码块时，它锁住的是synchronized后面的括号中的对象，这个对象可以是某个对象，也可以是某个类(.class)。

注意：一个线程访问一个类的static synchronized方法时，其他线程可以访问该类的非static synchronized的方法。原因是两个线程所获得的锁不同，前者是锁住的是这个类，而后者锁着的是这个实例对象，因此不存在互斥关系。

二、synchronize的底层实现原理

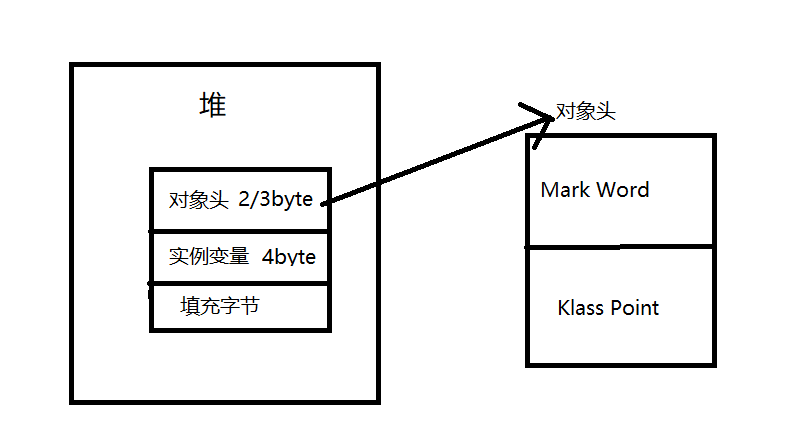
在说synchronized实现原理之前必须介绍一下，对象头和monitor(监视器)

对象是存储在堆中的，存储的内容分为三个部分：对象头、实例变量、填充字节

1. 对象头：对象头主要是由Mark Word 和 Klass Point(类元指针)，类元指针是用来指向类元数据的指针，JVM通过这个指针可以知道这个对象是哪个类的实例；而Mark Word中存储的则是用于自身的运行时数据，和synchronized的实现原理关系很大。如果对象是数组，则对象头占3个字节，如果是非数组则占2个字节，原因：如果是数组要记录数组的长度。

2. 先说实例变量：所谓实例变量就是存储对象的属性信息的，包括从父类继承来的属性。并且按照4字节对齐

3. 填充字节：顾名思义它的作用就是用来填充字节的。由于JVM规定对象的起始地址必须是8字节的整数倍。



Monitor：可以理解为一个监视器，在java中每个对象和类的内部都是和一个监视器monitor相关联的，为了实现监视器的排他性监视能力，JVM为每个对象和类都关联了一个监视器。并且任何对象都只有一个monitor与之对应，当这个对象的monitor被线程持有之后，他将进入锁定状态，其他线程无法访问。换句话来说就是：锁住了一个对象，就是获得了该对象的monitor。monitor对象实际是由底层ObjectMonitor(C++)对象实现的。主要介绍他的来历，在JVM的顶级基类oopDesc中有markOop类的子对象\_mark 通过调用\_mark 对象的monitor()这个方法，通过这个方法可以获得该对象的ObjectMonitor监视器对象。

下面举例介绍上面字段的含义，对一个synchronized修饰的方法（代码块）来说：

a、当多个线程访问该区域时，由于synchronized修饰只有一个会进入运行状态，其他线程都被阻塞此时其他线程进入blocking状态，这些线程都会被加入到\_EntryList列表中去

b、获得该锁的线程即就是获得该对象的monitor的线程，进入运行状态，并且将monitor对象中\_owner存储当前线程的地址。

c、当处于运行状态的线程调用了wait方法之后，那么当前线程释放monitor对象，进入等待状态。此时ObjectMonitor对象内\_owner字段赋值为null，同时将该线程加入到\_WaitSet列表中。直到有某个得到monitor对象的线程调用了notify或者notifyAll方法，则该线程才有机会从\_WaitSet列表中移除，加入到\_EntryList列表中等待获取monitor监视器锁。（注意：notify方法并不会释放对象锁，它会在方法结束时释放对象锁）

d、如果当前线程执行完毕，也会释放掉monitor监视器锁，此时ObjectMonitor对象的\_owner字段重新赋null。

A、synchronized修饰代码块时，通过在需要同步的代码块开始的位置插入一条monitorentry指令（底层实际是调用monitor()方法来获得的ObjectMonitor对象也就是monitor监视器锁）表示即将进入监视区域，需要获取monitor监视器锁，只有成功获取才可以执行代码块中的内容。在代码块的结束位置或者异常出现位置插入一条monitorexit指令表示退出监视区域，此时释放monitor监视器锁。为什么要插入两条monitorexit指令呢？原因是JVM要保证每一个monitorentry都要有一个monitorexit与它对应，所以就必须在两种退出监视区域的位置都要插入monitorexit，以保证无论哪种情况退出监视区域都可以释放monitor监视器锁。

B、当synchronized修饰同步方法时，获取monitor监视器的方式与代码块有所差异。不再是通过插入monitorentry和monitorexit指令来获取和释放monitor监视器锁。而是在该方法表结构中设置ACC\_SYNCHRONIZED标志。如果有线程执行到有ACC\_SYNCHRONIZED标志的方法时，它会在执行方法之前去尝试获取monitor监视器锁。如果成功获取则方法正常执行，获取失败则说明该monitor监视器锁已经被其他线程获取，则当前线程进入阻塞状态等待monitor监视器锁被释放。

#### 1.4.9.7 synchronized与lock的区别

|  |  |  |
| --- | --- | --- |
| **类别** | **synchronized** | Lock |
| 存在层次 | Java的关键字，在jvm层面上 | 是一个类 |
| 锁的释放 | 1. 以获取锁的线程执行完同步代码，释放锁 2. 线程执行发生异常，jvm会让线程释放锁 | 在finally中必须释放锁，不然容易造成线程死锁 |
| 锁的获取 | 假设A线程获得锁，B线程等待。如果A线程阻塞，B线程会一直等待 | 分情况而定，Lock有多个锁获取的方式，具体下面会说道，大致就是可以尝试获得锁，线程可以不用一直等待 |
| 锁状态 | 无法判断 | 可以判断 |
| 锁类型 | 可重入 不可中断 非公平 | 可重入 可判断 可公平（两者皆可） |
| 性能 | 少量同步 | 大量同步 |

#### 1.3.4.8 AQS同步队列

队列同步器（AbstractQueuedSynchronizer）简称AQS,是J.U.C同步构件的基础，包括ReentrantLock、ReentrantReadWriteLock、CountDownLatch、Semaphor都是基于AQS实现的。

**AQS原理**

用一个int类型的状态变量(volatile)state记录同步状态，默认值是0

用一个双向链表实现的队列对线程进程进行排队和调度

A线程使用compareAndSet(state,0,1)原子设置state的值，设置成功说明state当前无其他线程争用，A线程取锁的使用权。

设置不成功，说明B线程对state的值进行了设置，并且没有复位(state!=0)，B线程持有锁的使用权(B线程还没有释放锁)。A线程会构造成一个Node节点加入队列尾部并挂起。

当B线程执行完同步操作后，对state进行复位(state==0)，即释放锁，然后从队列头开始寻找，发现正在沉睡的A线程，将其唤醒。

#### 1.4.9.9 CAS无锁的概念、乐观锁和悲观锁

**CAS概念（无锁，是乐观锁的一种表现）**

①CAS是英文单词CompareAndSwap的缩写，中文意思是：比较并替换。

②CAS指令执行时，当且仅当旧值与预期值A相等时，才可以把值修改为B，否则就什么都不做。

②整个比较并替换的操作是一个原子操作，通过硬件指令支持

**乐观锁：**乐观并发控制（又名”乐观锁”，Optimistic Concurrency Control，缩写”OCC”）是一种并发控制的方法。它假设多用户并发的事务在处理时不会彼此互相影响，各事务能够在不产生锁的情况下处理各自影响的 那部分数据。在提交数据更新之前，每个事务会先检查在该事务读取数据后，有没有其他事务又修改了该数据。如果其他事务有更新的话，正在提交的事务会进行回滚。（数据库中实现乐观锁的方式就是添加版本号）

**悲观锁：**悲观并发控制（又名”悲观锁”，Pessimistic Concurrency Control，缩写”PCC”）是一种并发控制的方法。它可以阻止一个事务以影响其他用户的方式来修改数据。如果一个事务执行的操作读某行数据应用了 锁，那只有当这个事务把锁释放，其他事务才能够执行与该锁冲突的操作。

#### 1.4.9.10常见的原子操作类（未完成）

#### 1.4.9.11什么是ABA问题，出现ABA问题JDK是如何解决的（未完成）

#### 1.4.9.12 乐观锁的业务场景及实现方式（未完成）（未完成）

#### 1.4.9.13 Java 8并法包下常见的并发类

1、**ConcurrentHashMap** ：ConcurrentHashMap其实就是线程安全版本的hashMap。在每一个HashEntry上加一把锁，对于hash冲突的数据，因为采用链表存储，公用一把锁。**数据一致性**：对于数据元素的大小，ConcurrentHashMap将对应数组（HashEntry的长度）的变量为voliate类型的，也就是任何HashEntry发生变更，所有的地方都会知道数据的大小。

2、**CopyOnWriteArrayList**：CopyOnWriteArrayList是线程安全版本的ArrayList。和ArrayList不同的是，CopyOnWriteArrayList默认是创建了一个大小为0的容器。通过ReentrantLock来保证线程安全。CopyOnWriteArrayList其实每次增加的时候，需要新创建一个比原来容量+1大小的数组，然后拷贝原来的元素到新的数组中，同时将新插入的元素放在最末端。然后切换引用。

3、**CopyOnWriteArraySet**：CopyOnWriteArraySet是基于CopyOnWriteArrayList实现的。但是CopyOnWriteArraySet鉴于不能插入重复数据，因此每次add的时候都要遍历数据，性能略低于CopyOnWriteArrayList。

4、**ArrayBlockingQueue**：ArrayBlockingQueue是基于数组实现的一个线程安全的队列服务

5、**Atomic**类，如AtomicInteger、AtomicBoolean

#### 1.4.9.14偏向锁、轻量级锁、重量级锁、自旋锁的概念（未完成）

可参考：《Java多线程编程核心技术》

#### 1.4.9.15 volatile具有什么特性以及是否具有原子性？

volatile具有①可见性、②有序性（禁止指令重排），③ 但不具备原子性

### 1.4.10 jdk的有界阻塞队列在什么情况下发生阻塞

①队列满 ② 队列空时

## 1.5、JVM

### 1.5.1 JVM运行时内存区域划分

JVM运行时的内存区域分为：**程序计数器**，**虚拟机栈**，**本地方法栈**，**堆**，**方法区**，**运行时常量池**以及**直接内存区域。**

**1.程序计数器**

程序计数器是线程私有的，当前线程所执行的字节码行号指示器。字节码解析器通过改变这个计数器来选取下一条需要执行的字节码指令。

  执行java方法的时候，保存的是正在执行的虚拟机字节码指令地址，执行本地方法的时候保存的是空

  在jvm内存模型中唯一一个没有OOM异常的区域

**2.虚拟机栈**

也是线程私有的，生命周期与线程相同。此块区域描述的是方法执行的内存模型。每一个方法从调用到执行完成在虚拟机栈中描述的都是栈帧的入栈和出栈，虚拟机栈中的保存的是局部常量表，操作数栈，动态链接和方法出口等

注意两个知识点：局部变量表中保存有基本数据类型，对象引用和returnAddress类型；long和double占用2个局部变量空间，其他的占用一个局部变量空间

**3.本地方法栈**

  本地方法栈与虚拟机栈类似。差别在于，本地方法栈为虚拟机的本地方法服务。在jvm规范中，本地方法栈中的方法实现和数据类型没有做严格的规定，所以本地方法栈可以自由的实现，包括不使用java语言。在Hotspot中，本地方法栈被合并到了虚拟机栈中。本地方法栈也会抛出StackOverFlowError异常和OutOfMemoryError异常

**4.java堆**

  java堆是jvm中最大的一块区域，此区域是共享的，在虚拟机初始化的时候创建的。java堆中保存着具体的java对象实例，几乎所有的对象实例都保存在这个区域中。java虚拟机规范中规定，所有的数组和对象实例都保存在java堆中，但是随着JIT编译器的发展和逃逸分析技术的逐渐成熟，这个也成了并非绝对的。

  jvm规范中，java堆在物理内存上并不一定要是连续的，只要保证在逻辑上是连续的即可

  java堆的大小可以通过-Xmx和-Xmn来设定最大最小值，当然也可以设置成固定大小的

  在这个区域中，当没有足够内存分配对象的时候(GC之后)，则会抛出OutOfMemoryError异常。

**5.方法区**

  方法区与java堆一样，是线程共享的区域，此区域中保存着类信息，变量，静态变量，即时编译器编译后的代码等数据。在java虚拟机规范中，也将方法区描述为java堆的一部分，但是方法区还有另一个名字叫Non-Heap，目的也是为了与java堆区分开

  hotSpot上，习惯将这个区域叫永久带，因为hotspot团队将GC分代收集扩展到这个区域，或者是使用永久带来实现了方法区，但是本质上这两者是不能对等的

  java虚拟机规范对于方法区的限制也是很宽松的，如存储的不需要物理上连续的，只要逻辑上连续即可，与java堆一样。

  这个区域的GC是很少的，因为要回收的内容要求太苛刻，造成回收的效率不高，但是并不意味着就不进行GC，只是回收的成绩不够满意

   此区域也会出现OutOfMemoryError异常

**6.运行时常量池**

  运行时常量池是方法区的一部分，保存着编译期生成的各种字面量和符号引用

  运行时常量池相对于class文件常量池，多了动态性。java并不要求常量只有在编译器产生，运行期间也有可能将常量写入运行时常量池中，如String的intern方法

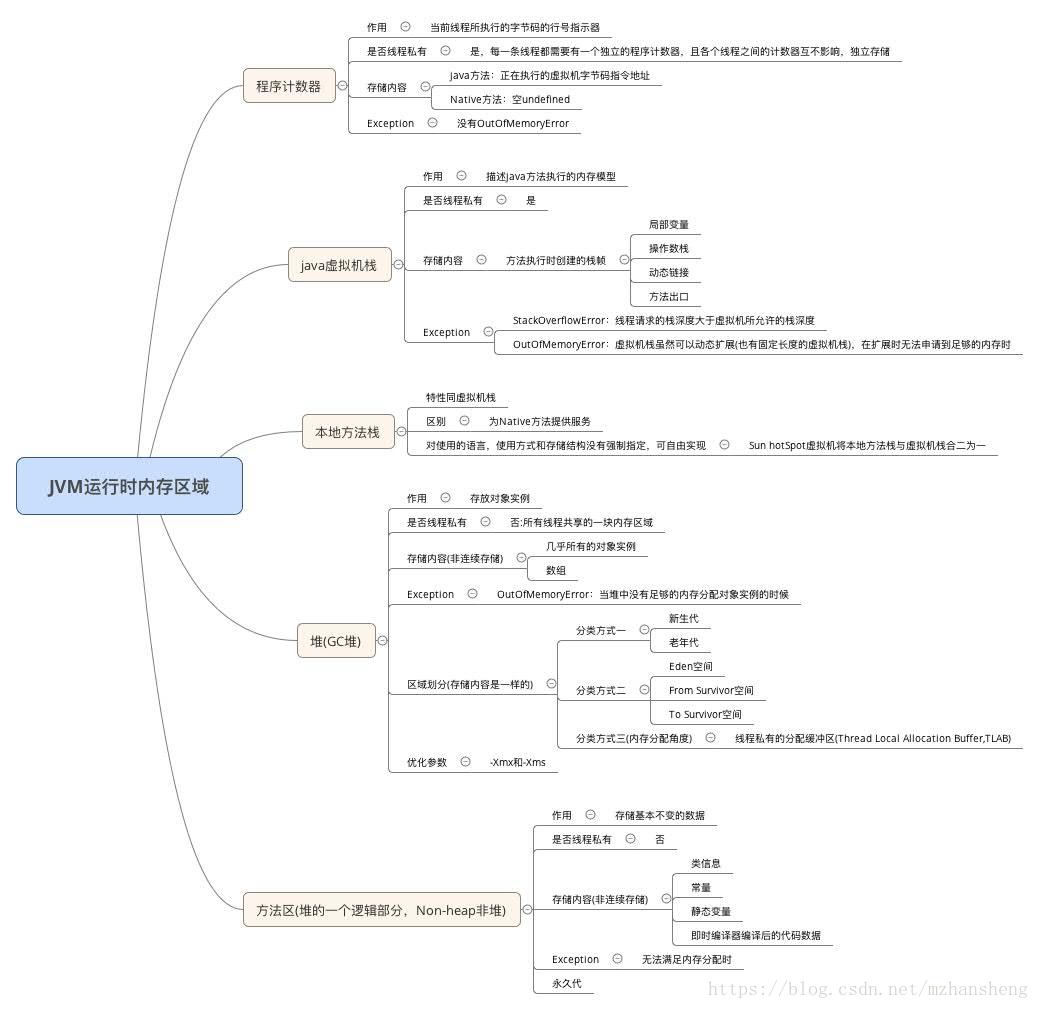
  此区域与方法区一样，会跑村OutOfMemoryError

**7.直接内存**

  直接内存并不是虚拟机运行时数据区域的一部分，也不是java虚拟机规范中的内存区域  自jdk1.4之后，加入了NIO，引入

了一种基于通道与缓存区的I/O方式，可以使用直接内存，提高了效率，因为使用直接内存便不用将对象从直接内存拷贝到堆中一样，减少了交互，提升了体验

  此区域也会抛出OutOfMemoryError异常



### 1.5.2内存溢出OOM和堆栈溢出SOE的示例及原因、如何排查与解决

**1、 内存溢出的原因是什么？**

内存溢出是由于没被引用的对象（垃圾）过多造成JVM没有及时回收，造成的内存溢出。如果出现这种现象可行代码排查：

一）是否App中的类中和引用变量过多使用了Static修饰 如public staitc Student s；在类中的属性中使用 static修饰的最好只用基本类型或字符串。如public static int i = 0; //public static String str;

二）是否App中使用了大量的递归或无限递归（递归中用到了大量的建新的对象）

三）是否App中使用了大量循环或死循环（循环中用到了大量的新建的对象）

四）检查App中是否使用了向数据库查询所有记录的方法。即一次性全部查询的方法，如果数据量超过10万多条了，就可能会造成内存溢出。所以在查询时应采用“分页查询”。

五）检查是否有数组，List，Map中存放的是对象的引用而不是对象，因为这些引用会让对应的对象不能被释放。会大量存储在内存中。

六）检查是否使用了“非字面量字符串进行+”的操作。因为String类的内容是不可变的，每次运行"+"就会产生新的对象，如果过多会造成新String对象过多，从而导致JVM没有及时回收而出现内存溢出。

七）使用 DDMS工具进行查找内存溢出的大概位置

**2、栈溢出的原因**

一）、是否有递归调用

二）、是否有大量循环或死循环

三）、全局变量是否过多

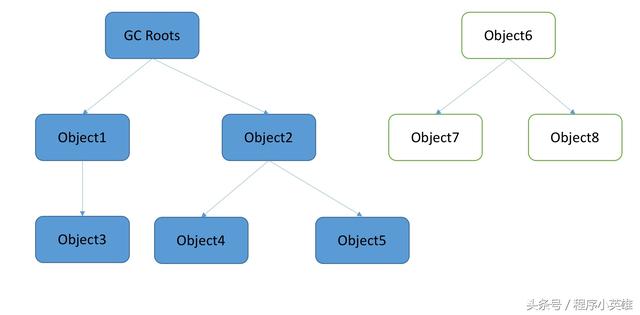
四）、 数组、List、map数据是否过大

五）使用DDMS工具进行查找大概出现栈溢出的位置

### 1.5.3如何判断对象是否可以回收或存活

它把内存中的每一个对象都看作一个节点，并且定义了一些对象作为根节点“GC Roots”。

JVM会起一个线程从所有的GC Roots开始往下遍历，当遍历完之后如果发现有一些对象不可到达，那么就认为这些对象已经没有用了，需要被回收。



**GC Roots的定义：**

第一种是**虚拟机栈中的引用的对象；**

第二种是我们在类中定义了**全局的静态的对象（**这种对象的引用会保存在共有的****方法区中**）**；

第三种便是**常量引用;**

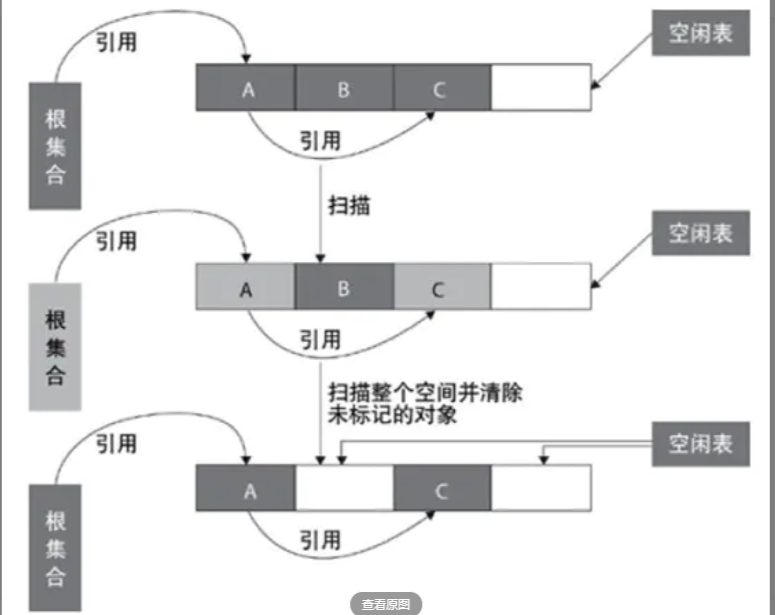
第四种便是**本地方法栈中引用的对象。**

### 1.5.4常见的GC回收算法及其含义

参考资料：<https://www.jianshu.com/p/3fc4450e1bbd>

1. 标记清除算法

标记-清除算法分为标记和清除两个阶段。该算法首先从根集合进行扫描，对存活的对象对象标记，标记完毕后，再扫描整个空间中未被标记的对象并进行回收



标记-清除算法的主要不足有两个：

**效率问题：**标记和清除两个过程的效率都不高;

**空间问题：**标记-清除算法不需要进行对象的移动，并且仅对不存活的对象进行处理，因此标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

1. 复制算法（新生代）

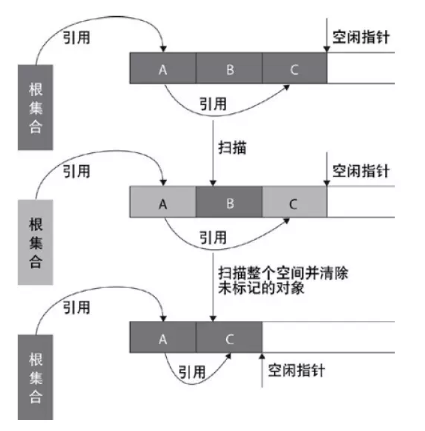
　复制算法将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这种算法适用于对象存活率低的场景，比如新生代。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。



1. 标记整理算法（老年代）

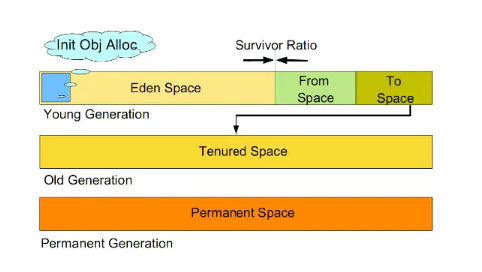
　复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。

记整理算法的标记过程类似标记清除算法，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。



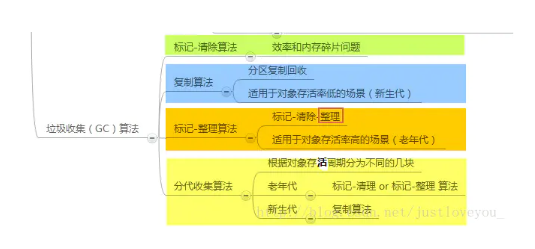
1. 分代收集算法

分代收集算法是基于这样一个事实：不同的对象的生命周期(存活情况)是不一样的，而不同生命周期的对象位于堆中不同的区域，因此对堆内存不同区域采用不同的策略进行回收可以提高 JVM 的执行效率。



1. 新生代：新生代的目标就是尽可能快速的收集掉那些生命周期短的对象。
2. 老年代：老年代存放的都是一些生命周期较长的对象，
3. 永久代：永久代主要用于存放静态文件，如Java类、方法等。

总结：



### 1.5.5常见的JVM性能监控和故障处理工具类：

jps、jstat、jmap、jinfo、jconsole等

### 1.5.6 JVM如何设置参数

参考资料：<https://blog.csdn.net/hua00shao/article/details/78258169>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **参数名称** | **含义** | **默认值** | **示例** | **说明** |
| -Xms | 初始堆大小 | 物理内存的1/64(1GB)，Server端JVM最好将-Xms和-Xmx设为相同值，开发测试机JVM可以保留默认值 | -Xms1000M | 默认(MinHeapFreeRatio参数可以调整)空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制. |
| -Xmx | 最大堆大小 | 物理内存的1/4(1GB)，最佳设值应该视物理内存大小及计算机内其他内存开销而定 | -Xms1000M | 默认(MaxHeapFreeRatio参数可以调整)空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制 |
| -Xmn | 年轻代大小(1.4or lator) | 不熟悉最好保留默认值 | 默认值 | 注意：此处的大小是（eden+ 2 survivor space)。与jmap -heap中显示的New gen是不同的。整个堆大小=年轻代大小 + 年老代大小 + 持久代大小。增大年轻代后,将会减小年老代大小.此值对系统性能影响较大,Sun官方推荐配置为整个堆的3/8 |
| -XX:NewSize | 设置年轻代大小(for 1.3/1.4) | 无 | 不需要设置 | 无 |
| -XX:MaxNewSize | 年轻代最大值(for 1.3/1.4) | 无 | 不需要设置 | 无 |
| -XX:PermSize | 设置持久代(perm gen)初始值 | 物理内存的1/64，不熟悉保留默认值 | 默认值 | 无 |
| -XX:MaxPermSize | 设置持久代最大值 | 物理内存的1/4，不熟悉保留默认值 | 默认值 | 无 |
| -Xss | 每个线程的堆栈大小 | JDK5.0 以后每个线程堆栈大小为1M,以前每个线程堆栈大小为256K. | 默认值 | 根据应用的线程所需内存大小进行调整。在相同物理内存下,减小这个值能生成更多的线程.但是操作系统对一个进程内的线程数还是有限制的,不能无限生成,经验值在3000~5000左右，一般小的应用， 如果栈不是很深， 应该是128k够用的 大的应用建议使用256k。这个选项对性能影响比较大，需要严格的测试。（校长）和threadstacksize选项解释很类似,官方文档似乎没有解释,在论坛中有这样一句话:"”-Xss is translated in a VM flag named ThreadStackSize”一般设置这个值就可以了。 |
| -XX:ThreadStackSize | Thread Stack Size | 上面的-Xss不需要设置，如果要设置直接设置这个参数就可以 | 默认值 | (0 means use default stack size) [Sparc: 512; Solaris x86: 320 (was 256 prior in 5.0 and earlier); Sparc 64 bit: 1024; Linux amd64: 1024 (was 0 in 5.0 and earlier); all others 0.] |
| -XX:NewRatio | 年轻代(包括Eden和两个Survivor区)与年老代的比值(除去持久代) | Xms=Xmx并且设置了Xmn的情况下，该参数不需要进行设置。 | 默认值 | -XX:NewRatio=4表示年轻代与年老代所占比值为1:4,年轻代占整个堆栈的1/5。 |
| -XX:SurvivorRatio | Eden区与Survivor区的大小比值 |  | 默认值 | 设置为8,则两个Survivor区与一个Eden区的比值为2:8,一个Survivor区占整个年轻代的1/10 |
| -XX:LargePageSizeInBytes | 内存页的大小 | =128m，不可设置过大， 会影响Perm的大小 | 默认值 | 128m |
| -XX:+UseFastAccessorMethods | 原始类型的快速优化(jdk 1.6 or later) | - | 默认值 | 无 |
| -XX:+DisableExplicitGC | 关闭System.gc() | - | 默认值 | 这个参数需要严格的测试 |
| -XX:MaxTenuringThreshold | 垃圾最大年龄，表示对象被移到老年代的年龄阈值的最大值 | 15 | 默认值 | 控制对象能经过几次GC才被转移到老年代。回收如果设置为0的话,则年轻代对象不经过Survivor区,直接进入年老代. 对于年老代比较多的应用,可以提高效率.如果将此值设置为一个较大值,则年轻代对象会在Survivor区进行多次复制,这样可以增加对象再年轻代的存活 时间,增加在年轻代即被回收的概率。该参数只有在串行GC时才有效。 |
| -XX:+AggressiveOpts | 加快编译 | - | 默认值 | 启用该选项之后，需要考虑到性能的提升，同样也需要考虑到性能提升所带来的不稳定风险。 |
| -XX:+UseBiasedLocking | 锁机制的性能改善 (Java 5 update 6 or later) | + | 默认值 | Java 5 HotSpot JDK需要明确的命令来启用这个特性，在使用-XX:+AggressiveOpts选项，有偏见的锁会Java 5中会被自动启用。在Java 6中是默认启用的。 |
| -XX:TLABWasteTargetPercent | TLAB占eden区的百分比 | 1% | 默认值 | 无 |
| -XX:+CollectGen0First | FullGC时是否先YGC | false | 默认值 | 无 |

收集器相关参数

| **参数名称** | **含义** | **默认值** | **示例** | **说明** |
| --- | --- | --- | --- | --- |
| -XX:+UseParNewGC | 设置年轻代为并行收集 | - | + | 可与CMS收集同时使用，JDK5.0以上,JVM会根据系统配置自行设置,所以无需再设置此值 |
| -XX:ParallelGCThreads | 并行收集器的线程数 | 默认为CPU核心数 | 默认值 | 此值最好配置与处理器数目相等 同样适用于CMS |
| -XX:+UseConcMarkSweepGC | 使用CMS内存收集 | - | + | 注意最新的JVM版本，当开启此选项时，-XX：UseParNewGC会自动开启。因此，如果年轻代的并行GC不想开启，可以通过设置-XX：-UseParNewGC来关掉。 |
| -XX:ParallelCMSThreads | CMS并发收集线程数 | 默认为CPU核心数 | 默认值 | 如果还标志未设置，JVM会根据并行收集器中的-XX：ParallelGCThreads参数的值来计算出默认的并行CMS线程数。该公式是ConcGCThreads = (ParallelGCThreads + 3)/4。因此，对于CMS收集器， -XX:ParallelGCThreads标志不仅影响“stop-the-world”垃圾收集阶段，还影响并发阶段。总之，有不少方法可以配置CMS收集器的多线程执行。正是由于这个原因,建议第一次运行CMS收集器时使用其默认设置, 然后如果需要调优再进行测试 |
| -XX:+UseCMSCompactAtFullCollection | 在FULL GC的时候， 对年老代的压缩 | - | + | CMS是不会移动内存的， 因此， 这个非常容易产生碎片， 导致内存不够用， 因此， 内存的压缩这个时候就会被启用。 增加这个参数是个好习惯。可能会影响性能,但是可以消除碎片 |
| -XX:CMSFullGCsBeforeCompaction | full gc多少次后进行内存压缩 | 默认为0 | 默认值 | 由于并发收集器不对内存空间进行压缩,整理,所以运行一段时间以后会产生"碎片",使得运行效率降低.此值设置运行多少次GC以后对内存空间进行压缩,整理. |
| -XX:CMSInitiatingOccupancyFraction | 老年代使用70％后开始CMS收集 | =92 | =75 | 为了保证不出现promotion failed(见下面介绍)错误,该值的设置需要满足以下公式 |
| -XX:+CMSClassUnloadingEnabled | 持久代使用CMS并发收集 | jdk1.7默认关闭,1.8默认打开 | - | 它会增加CMS remark的暂停时间，如果没有程序产生大量的临时类，新类加载并不频繁，这个参数还是不开的好 |
| -XX:CMSInitiatingPermOccupancyFraction | 设置Perm Gen使用到达多少比率时触发 | =92 | 默认值 | 无 |

日志：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 参数名称 | 含义 | 默认值 | 示例 | 说明 |
| -Xloggc | 记录日志文件位置 | 无 | /data/log/jetty/gc.log | 无 |
| -XX:+PrintGCDateStamps | 打印可读的日期而不是时间戳 | - | + | 无 |
| -XX:+PrintGCDetails | 打印日志详情 | - | + | 输出形式:[GC [DefNew: 8614K->781K(9088K), 0.0123035 secs] 118250K->113543K(130112K), 0.0124633 secs][GC [DefNew: 8614K->8614K(9088K), 0.0000665 secs][Tenured: 112761K->10414K(121024K), 0.0433488 secs] 121376K->10414K(130112K), 0.0436268 secs] |
| -XX:+PrintGCApplicationStoppedTime | 打印GC停顿时间 | - | + | 它除了打印清晰的GC停顿时间外，还可以打印其他的停顿时间，比如取消偏向锁，class 被agent redefine，code deoptimization等等，有助于发现一些原来没想到的问题，建议也加上。输出形式:Total time for which application threads were stopped: 0.0468229 seconds |
| -XX:+PrintCommandLineFlags | 打印已配置的XX类参数 | - | + | 打印出命令行里设置了的参数以及因为这些参数隐式影响的参数，比如开了CMS后，-XX:+UseParNewGC也被自动打开 |
| -XX:+HeapDumpOnOutOfMemoryError | 输出Heap Dump到指定文件 | - | + | 在Out Of Memory，JVM快死快死掉的时候，输出Heap Dump到指定文件。不然开发很多时候还真不知道怎么重现错误。路径只指向目录，JVM会保持文件名的唯一性，叫java\_pid${pid}.hprof。如果指向文件，而文件已存在，反而不能写入。 |
| -XX:HeapDumpPath | 设置Heap Dump输出路径 | 无 | =${LOGDIR}/ | 无 |

小结

性能相关

-XX:-UseBiasedLocking -XX:-UseCounterDecay -XX:AutoBoxCacheMax=20000 -XX:+PerfDisableSharedMem -XX:+AlwaysPreTouch -Djava.security.egd=file:/dev/./urandom

内存大小相关(JDK7)

-Xms4096m -Xmx4096m -Xmn2048m -XX:MaxDirectMemorySize=4096m-XX: PermSize=256m -XX:MaxPermSize=512m -XX:ReservedCodeCacheSize=240M

CMS GC 相关

-XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly -XX:MaxTenuringThreshold=6 -XX:+ExplicitGCInvokesConcurrent -XX:+ParallelRefProcEnabled

GC 日志 相关

-Xloggc:/dev/shm/app-gc.log -XX:+PrintGCApplicationStoppedTime -XX:+PrintGCDateStamps -XX:+PrintGCDetails

异常 日志 相关

-XX:-OmitStackTraceInFastThrow -XX:ErrorFile=${LOGDIR}/hs\_err\_%p.log -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=${LOGDIR}/

### 1.5.7 JVM性能调优

**参考资料**：https://blog.csdn.net/weixin\_42447959/article/details/81637909

1、JVM调优目标：使用较小的内存占用来获得较高的吞吐量或者较低的延迟。

重要指标：

内存占用：程序正常运行需要的内存大小。

延迟：由于垃圾收集而引起的程序停顿时间。

吞吐量：用户程序运行时间占用户程序和垃圾收集占用总时间的比值。

2、JVM调优工具

（1）调优可以依赖、参考的数据有系统运行日志、堆栈错误信息、gc日志、线程快照、堆转储快照等。

（2）JVM调优工具

①用 jps（JVM process Status）可以查看虚拟机启动的所有进程、执行主类的全名、JVM启动参数，

②用jstat（JVM Statistics Monitoring Tool）监视虚拟机信息

jstat -gc pid 500 10 ：每500毫秒打印一次Java堆状况（各个区的容量、使用容量、gc时间等信息），打印10次

③用jmap（Memory Map for Java）查看堆内存信息

执行jmap -histo pid可以打印出当前堆中所有每个类的实例数量和内存占用，

④利用jconsole、jvisualvm分析内存信息(各个区如Eden、Survivor、Old等内存变化情况)，如果查看的是远程服务器的JVM

1. JVM调优经验

-Xms和-Xmx的值设置成相等，堆大小默认为-Xms指定的大小，默认空闲堆内存小于40%时，JVM会扩大堆到-Xmx指定的大小；空闲堆内存大于70%时，JVM会减小堆到-Xms指定的大小。如果在Full GC后满足不了内存需求会动态调整，这个阶段比较耗费资源。

新生代尽量设置大一些，让对象在新生代多存活一段时间，每次Minor GC 都要尽可能多的收集垃圾对象，防止或延迟对象进入老年代的机会，以减少应用程序发生Full GC的频率。

老年代如果使用CMS收集器，新生代可以不用太大，因为CMS的并行收集速度也很快，收集过程比较耗时的并发标记和并发清除阶段都可以与用户线程并发执行。

方法区大小的设置，1.6之前的需要考虑系统运行时动态增加的常量、静态变量等，1.7只要差不多能装下启动时和后期动态加载的类信息就行。

**代码实现方面**，性能出现问题比如程序等待、内存泄漏除了JVM配置可能存在问题

避免创建过大的对象及数组：过大的对象或数组在新生代没有足够空间容纳时会直接进入老年代，如果是短命的大对象，会提前出发Full GC。

避免同时加载大量数据，如一次从数据库中取出大量数据，或者一次从Excel中读取大量记录，可以分批读取，用完尽快清空引用。

当集合中有对象的引用，这些对象使用完之后要尽快把集合中的引用清空，这些无用对象尽快回收避免进入老年代。

可以在合适的场景（如实现缓存）采用软引用、弱引用，比如用软引用来为ObjectA分配实例：SoftReference objectA=new SoftReference(); 在发生内存溢出前，会将objectA列入回收范围进行二次回收，如果这次回收还没有足够内存，才会抛出内存溢出的异常。

避免产生死循环，产生死循环后，循环体内可能重复产生大量实例，导致内存空间被迅速占满。

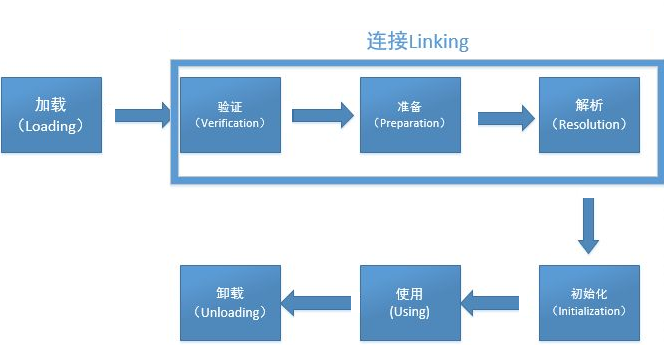
尽量避免长时间等待外部资源（数据库、网络、设备资源等）的情况，缩小对象的生命周期，避免进入老年代，如果不能及时返回结果可以适当采用异步处理的方式等。

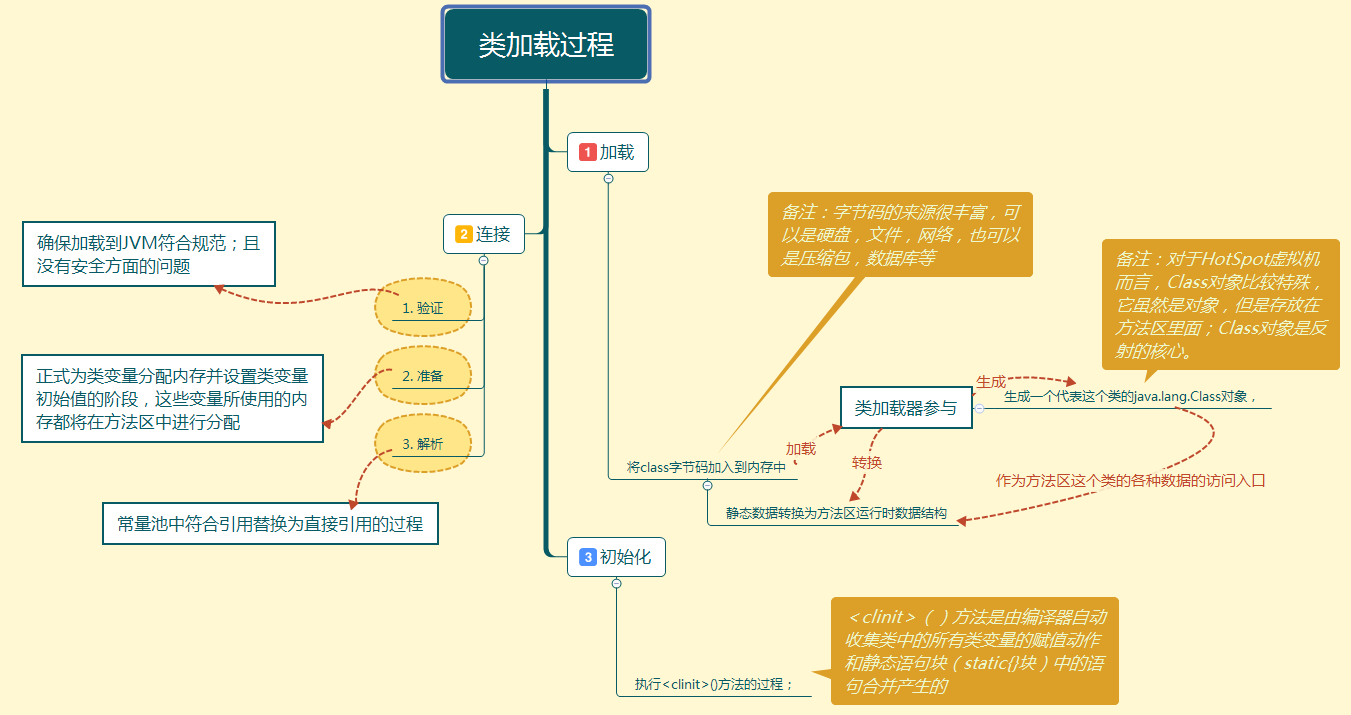
### 1.5.8类加载器、双亲委派模型、一个类的生命周期、类是如何加载到JVM中的

双亲委派模型：①向上委托给父类加载，父类加载不了，再自己加载。②好处是避免重复加载和篡改Java核心API。

### 1.5.9 类加载的过程

Java虚拟机中类加载的全过程，即 **加载**、**验证**、**准备**、**解析**和**初始化；**





1. **加载**
   1. **加载阶段需要完成以下三件事情**

| **三个阶段** |
| --- |
| 通过一个类的全限定名来获取定义此类的二进制字节流 |
| 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构 |
| 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口 |

**1.2 数组类与非数组类**

**(一)非数组类：**

一个非数组类的加载阶段（准确地说，是加载阶段中获取类的二进制字节流的动作）是开发人员可控性最强的，因为加载阶段既可以使用系统提供的引导类加载器来完成，也可以由用户自定义的类加载器去完成，开发人员可以通过定义自己的类加载器去控制字节流的获取方式（即重写一个类加载器的loadClass()方法）

**(二)数组类：**

对于数组类而言，情况就有所不同，数组类本身不通过类加载器创建，它是由Java虚拟机直接创建的。但数组类与类加载器仍然有很密切的关系，因为数组类的元素类型（ElementType，指的是数组去掉所有维度的类型）最终是要靠类加载器去创建

一个数组类（下面简称为C）创建过程就遵循以下规则：

规则描述

如果数组的组件类型（Component Type，指的是数组去掉一个维度的类型）是引用类型，那就递归采用本节中定义的加载过程去加载这个组件类型，数组C将在加载该组件类型的类加载器的类名称空间上被标识

如果数组的组件类型不是引用类型（例如int[]数组），Java虚拟机将会把数组C标记为与引导类加载器关联

数组类的可见性与它的组件类型的可见性一致，如果组件类型不是引用类型，那数组类的可见性将默认为public.

加载阶段与连接阶段的部分内容（如一部分字节码文件格式验证动作）是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始，但这些夹在加载阶段之中进行的动作，仍然属于连接阶段的内容，这两个阶段的开始时间仍然保持着固定的先后顺序

**二、验证**

验证是连接阶段的第一步，这一阶段的目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全

验证阶段大致上会完成下面4个阶段的检验动作：文件格式验证、元数据验证、字节码验证、符号引用验证

**2.1 文件格式验证**

第一阶段要验证字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理；主要目的是保证输入的字节流能正确地解析并存储于方法区之内，格式上符合描述一个Java类型信息的要求。这阶段的验证是基于二进制字节流进行的，只有通过了这个阶段的验证后，字节流才会进入内存的方法区中进行存储，所以后面的3个验证阶段全部是基于方法区的存储结构进行的，不会再直接操作字节流

**2.2 元数据验证**

第二阶段是对字节码描述的信息进行语义分析，以保证其描述的信息符合Java语言规范的要求

**2.3 字节码验证**

第三阶段是整个验证过程中最复杂的一个阶段，主要目的是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。在第二阶段对元数据信息中的数据类型做完校验后，这个阶段将对类的方法体进行校验分析，保证被校验类的方法在运行时不会做出危害虚拟机安全的事件

**2.4 符号引用验证**

最后一个阶段的校验发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在连接的第三阶段——解析阶段中发生。符号引用验证可以看做是对类自身以外（常量池中的各种符号引用）的信息进行匹配性校验

**三、准备**

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法区中进行分配

这时候进行内存分配的仅包括类变量（被static修饰的变量），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在Java堆中

其次，这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量的定义为：

public static int value=123；

那变量value在准备阶段过后的初始值为0而不是123，因为这时候尚未开始执行任何Java方法，而把value赋值为123的putstatic指令是程序被编译后，存放于类构造器＜clinit＞（）方法之中，所以把value赋值为123的动作将在初始化阶段才会执行

在“通常情况”下初始值是零值，那相对的会有一些“特殊情况”：如果类字段的字段属性表中存在ConstantValue属性，那在准备阶段变量value就会被初始化为ConstantValue属性所指定的值，假设上面类变量value的定义变为：

public static final int value=123；

编译时Javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将value赋值为123。

**四、解析**

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程

对同一个符号引用进行多次解析请求是很常见的事情，除invokedynamic指令以外，虚拟机实现可以对第一次解析的结果进行缓存（在运行时常量池中记录直接引用，并把常量标识为已解析状态）从而避免解析动作重复进行。

虚拟机规范之中并未规定解析阶段发生的具体时间;解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行

**注：解析的过程比较复杂，具体可以参考《深入理解java虚拟机》**

**五、初始化**

类初始化阶段是类加载过程的最后一步，前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的Java程序代码

原文链接：<https://blog.csdn.net/qq_31156277/article/details/80188110>

### 1.5.10强引用、软引用、弱引用、虚引用

Java中提供这四种引用类型主要有两个目的：第一是可以让程序员通过代码的方式决定某些对象的生命周期；第二是有利于JVM进行垃圾回收。

**强引用：**

如果一个对象具有强引用，那就类似于必不可少的物品，不会被垃圾回收器回收。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不回收这种对象。

**软引用：**

软引用是用来描述一些**有用但并不是必需的对象**，在Java中用java.lang.ref.SoftReference类来表示。对于软引用关联着的对象，只有在内存不足的时候JVM才会回收该对象。

**弱引用：**

弱引用也是用来描述**非必需对象**的，当JVM进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。在java中，用java.lang.ref.WeakReference类来表示。

弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。所以被软引用关联的对象只有在内存不足时才会被回收，而被弱引用关联的对象在JVM进行垃圾回收时总会被回收。

**虚引用：**

虚引用和前面的软引用、弱引用不同，它并不影响对象的生命周期。在java中用java.lang.ref.PhantomReference类表示。如果一个对象与虚引用关联，则跟没有引用与之关联一样，在任何时候都可能被垃圾回收器回收。虚引用主要用来跟踪对象被垃圾回收的活动。

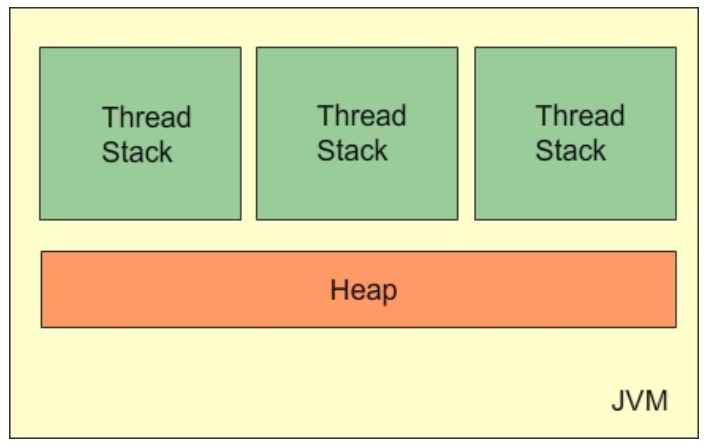
虚引用必须和引用队列关联使用，当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会把这个虚引用加入到与之 关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

**总结：**

|  |  |  |  |
| --- | --- | --- | --- |
| 引用类型 | 被回收时间 | 用途 | 生存时间 |
| 强引用 | 从来不会 | 对象的一般状态 | JVM停止运行时 |
| 软引用 | 内存不足时 | 对象缓存 | 内存不足时 |
| 弱引用 | jvm垃圾回收时 | 对象缓存 | gc运行后 |
| 虚引用 | 未知 | 未知 | 未知 |

**原文链接：**<https://blog.csdn.net/qq_39192827/article/details/85611873>

### 1.5.11 Java内存模型JMM



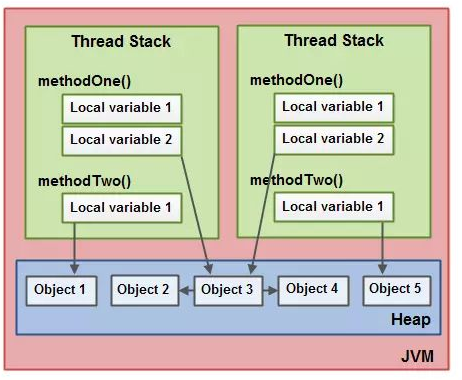
**Java内存模型（JMM）**

我们常说的JVM内存模式指的是JVM的内存分区；而Java内存模式是一种虚拟机规范。

Java虚拟机规范中定义了Java内存模型（Java Memory Model，JMM），用于屏蔽掉各种硬件和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的并发效果，JMM规范了Java虚拟机与计算机内存是如何协同工作的：规定了一个线程如何和何时可以看到由其他线程修改过后的共享变量的值，以及在必须时如何同步的访问共享变量。

原始的Java内存模型存在一些不足，因此Java内存模型在Java1.5时被重新修订。这个版本的Java内存模型在Java8中仍然在使用。

Java内存模型（不仅仅是JVM内存分区）：调用栈和本地变量存放在线程栈上，对象存放在堆上。



原文链接：<https://zhuanlan.zhihu.com/p/29881777>

### 1.5.12 JVM怎样判断某个对象应被回收？

引用计数法与可达性分析法。

① 【引用计数法】系统会为对象添加一个计数器，当有新的引用时加1，引用失效时减1。但是此方法无法解决两个对象循环引用的问题。

② 【可达性分析法】通过对象的引用链来判断该对象是否需要被回收，通过一系列的GC Roots的对象作为起始点，从这些起节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的，就需要回收此对象。

### 1.5.13 发生OOM（outOfMemoryError）时，年轻代和老年代都正常，那有可能是什么原因导致的？

① 原生内存不足（操作系统不允许申请更大的内存）；

② 永久代或元空间不足；

③ JVM执行GC耗时太久

## 1.6、设计模式(未完成)

### 1.6.1常见的设计模式

### 1.6.2设计模式的的六大原则及其含义

### 1.6.3常见的单例模式以及各种实现方式的优缺点，哪一种最好，手写常见的单利模式

### 1.6.4设计模式在实际场景中的应用

### 1.6.5 Spring中用到了哪些设计模式

### 1.6.6 MyBatis中用到了哪些设计模式

### 1.6.7你项目中有使用哪些设计模式

### 1.6.8说说常用开源框架中设计模式使用分析

动态代理很重要！！！

## 1.7、数据结构（未完成）

### 1.7.1树（二叉查找树、平衡二叉树、红黑树、B树、B+树）

### 1.7.2 深度有限算法、广度优先算法

### 1.7.3克鲁斯卡尔算法、普林母算法、迪克拉斯算法

### 1.7.4什么是一致性Hash及其原理、Hash环问题

### 1.7.5常见的排序算法和查找算法：快排、折半查找、堆排序等

## 1.8、网络/IO基础（未完成）

### 1.8.1 网络

#### 1.8.1.1什么是长连接和短连接

#### 1.8.1.2 Http1.0和2.0相比有什么区别，可参考《Http 2.0》

#### 1.8.1.3 Https的基本概念

#### 1.8.1.4 三次握手和四次挥手、为什么挥手需要四次

#### 1.8.1.5 从游览器中输入URL到页面加载的发生了什么？可参考《从输入URL到页面加载发生了什么》

### 1.8.2 IO/NIO

#### 1.8.2.1 I/O与NIO的区别

① IO是面向流的，NIO是面向缓冲区的

② IO是阻塞的，NIO是非阻塞的

③ IO无Selector，NIO需要Selector

#### 1.8.2.2 NIO和AIO的本质区别是什么

①NIO读写数据是由应用进程进行的；

②AIO读写数据是有操作系统完成的，当操作完成读写，主动通知应用进程

#### 1.8.2.3 BIO、NIO、AIO的概念

# 第二章 数据存储和消息队列

## 2.1、数据库

### 2.1.1 MySQL 索引使用的注意事项

**索引的类型**：

* UNIQUE(唯一索引)：不可以出现相同的值，可以有NULL值
* INDEX(普通索引)：允许出现相同的索引内容
* PROMARY KEY(主键索引)：不允许出现相同的值
* fulltext index(全文索引)：可以针对值中的某个单词，但效率确实不敢恭维
* 组合索引：实质上是将多个字段建到一个索引里，列值的组合必须唯一

**全文索引**：主要用来查找文本中的关键字，而不是直接与索引中的值相比较。fulltext索引跟其它索引大不相同，它更像是一个搜索引擎，而不是简单的where语句的参数匹配。fulltext索引配合match against操作使用，而不是一般的where语句加like。它可以在create table，alter table ，create index使用，不过目前只有char、varchar，text 列上可以创建全文索引。值得一提的是，在数据量较大时候，现将数据放入一个没有全局索引的表中，然后再用CREATE index创建fulltext索引，要比先为一张表建立fulltext然后再将数据写入的速度快很多。

组合索引需要遵循“**最左前缀**”，就是只从最左边的开始组合，并不是只要包含这三列的查询都会用到该组合索引。**如果没有左前索引，mysql不会执行索引查询。**

**不使用索引的情况：**

|  |
| --- |
| SELECT `sname` FROM `stu` WHERE `age`+**10=30;-- 不会使用索引,因为所有索引列参与了计算** |
| **SELECT `sname` FROM `stu` WHERE LEFT(`date`,4) <1990; -- 不会使用索引,因为使用了函数运算,原理与上面相同** |
| **SELECT \* FROM `houdunwang` WHERE `uname` LIKE'后盾%' -- 走索引**  **SELECT \* FROM `houdunwang` WHERE `uname` LIKE "%后盾%" -- 不走索引** |
| **-- 正则表达式不使用索引,这应该很好理解,所以为什么在SQL中很难看到regexp关键字的原因** |
| **-- 字符串与数字比较不使用索引;**  **CREATE TABLE `a` (`a` char(10));**  **EXPLAIN SELECT \* FROM `a` WHERE `a`="1" -- 走索引**  **EXPLAIN SELECT \* FROM `a` WHERE `a`=1 -- 不走索引** |
| **select \* from dept where dname='xxx' or loc='xx' or deptno=45 --如果条件中有or,即使其中有条件带索引也不会使用。换言之,就是要求使用的所有字段,都必须建立索引, 我们建议大家尽量避免使用or 关键字** |
| **-- 如果mysql估计使用全表扫描要比使用索引快,则不使用索引** |

**索引的缺点：**

* 虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT,UPDATE和DELETE。因为更新表时，mysql不仅要保存数据，还要保存一下索引文件
* 建立索引会占用磁盘空间的索引文件。一般情况这个问题不太严重，但如果你在要给大表上建了多种组合索引，索引文件会膨胀很宽

      索引只是提高效率的一个方式，如果mysql有大数据量的表，就要花时间研究建立最优的索引，或优化查询语句。

**索引的使用技巧：**

|  |
| --- |
| 1.索引不会包含有NULL的列         只要列中包含有NULL值，都将不会被包含在索引中，复合索引中只要有一列含有NULL值，那么这一列对于此符合索引就是无效的。 |
| 2.使用短索引         对串列进行索引，如果可以就应该指定一个前缀长度。例如，如果有一个char（255）的列，如果在前10个或20个字符内，多数值是唯一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作。 |
| 3.索引列排序         mysql查询只使用一个索引，因此如果where子句中已经使用了索引的话，那么order by中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作，尽量不要包含多个列的排序，如果需要最好给这些列建复合索引。 |
| 4.like语句操作        一般情况下不鼓励使用like操作，如果非使用不可，注意正确的使用方式。like ‘%aaa%’不会使用索引，而like ‘aaa%’可以使用索引。 |
| 5.不要在列上进行运算 |
| 6.不使用NOT IN 、<>、！=操作，但<,<=，=，>,>=,BETWEEN,IN是可以用到索引的 |
| 7.索引要建立在经常进行select操作的字段上。         这是因为，如果这些列很少用到，那么有无索引并不能明显改变查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。 |
| 8.索引要建立在值比较唯一的字段上。 |
| 9.对于那些定义为text、image和bit数据类型的列不应该增加索引。因为这些列的数据量要么相当大，要么取值很少。 |
| 10.在where和join中出现的列需要建立索引。 |
| 11.where的查询条件里有不等号(where column != …),mysql将无法使用索引。 |
| 12.如果where字句的查询条件里使用了函数(如：where DAY(column)=…),mysql将无法使用索引。 |
| 13.在join操作中(需要从多个数据表提取数据时)，mysql只有在**主键和外键的数据类型相同**时才能使用索引，否则及时建立了索引也不会使用。 |

### 2.1.2 DDL、DML、DCL分别指什么

**DML（data manipulation language）**： 它们是SELECT、UPDATE、INSERT、DELETE，就象它的名字一样，这4条命令是用来对数据库里的数据进行操作的语言

**DDL（data definition language）**：DDL比DML要多，主要的命令有CREATE、ALTER、DROP等，DDL主要是用在定义或改变表（TABLE）的结构，数据类型，表之间的链接和约束等初始化工作上，他们大多在建立表时使用

**DCL（Data Control Language）：**是数据库控制功能。是用来设置或更改数据库用户或角色权限的语句，包括（grant,deny,revoke等）语句。在默认状态下，只有sysadmin,dbcreator,db\_owner或db\_securityadmin等人员才有权力执行DCL

### 2.1.3 explain命令

• EXPLAIN不会告诉你关于触发器、存储过程的信息或用户自定义函数对查询的影响情况  
• EXPLAIN不考虑各种Cache  
• EXPLAIN不能显示MySQL在执行查询时所作的优化工作  
• 部分统计信息是估算的，并非精确值  
• EXPALIN只能解释SELECT操作，其他操作要重写为SELECT后查看执行计划。

### 2.1.4 left join，right join，inner join

left join(左联接) 返回包括左表中的所有记录和右表中联结字段相等的记录  
right join(右联接) 返回包括右表中的所有记录和左表中联结字段相等的记录  
inner join(等值连接) 只返回两个表中联结字段相等的行

### 2.1.5数据库事物ACID（原子性、一致性、隔离性、持久性）

**原子性（Atomicity）**:原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。

**一致性（Consistency）**:事务前后数据的完整性必须保持一致。

**隔离性（Isolation）**:事务的隔离性是多个用户并发访问数据库时，数据库为每一个用户开启的事务，不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离。

**持久性（Durability）**:持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响

### 2.1.6 事物的隔离级别（读未提交、读以提交、可重复读、可序列化读）

|  |  |
| --- | --- |
| 设置 | 描述 |
| Serializable | 可避免脏读、不可重复读、虚读情况的发生。（串行化） |
| Repeatable read | 可避免脏读、不可重复读情况的发生。（可重复读） |
| Read committed | 可避免脏读情况发生（读已提交）。 |
| Read uncommitted | 最低级别，以上情况均无法保证。(读未提交) |

### 2.1.7 脏读、幻读、不可重复读

**脏读**：指一个事务读取了另外一个事务未提交的数据。

**不可重复读**：在一个事务内读取表中的某一行数据，多次读取结果不同。（这个不一定是错误，只是某些场合不对）

**虚读(幻读)**:是指在一个事务内读取到了别的事务插入的数据，导致前后读取不一致。

### 2.1.8数据库的几大范式

1. **第一范式(确保每列保持原子性)**（1NF）**:**数据库表中的所有字段值都是不可分解的原子值;
2. **第二范式**（2NF）**:**要求数据库表中的每个实例或记录必须可以被唯一地区分(第二范式就是在第一范式的基础上属性完全依赖于主键);
3. **第三范式(确保每列都和主键列直接相关,而不是间接相关)**（3NF）:第三范式（3NF）是第二范式（2NF）的一个子集，即满足第三范式（3NF）必须满足第二范式（2NF）;
4. **巴斯-科德范式**（BCNF）:在3NF基础上，任何非主属性不能对主键子集依赖（在3NF基础上消除对主码子集的依赖）; 巴斯-科德范式（BCNF）是第三范式（3NF）的一个子集，即满足巴斯-科德范式（BCNF）必须满足第三范式（3NF）。
5. **[第四范式](https://baike.baidu.com/item/%E7%AC%AC%E5%9B%9B%E8%8C%83%E5%BC%8F/3193985" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E8%8C%83%E5%BC%8F/_blank)**(4NF）:设关系R（X，Y，Z），其中X，Y，Z是成对的、不相交属性的集合。若存在非平凡多值依赖，则意味着对R中的每个属性存在有函数依赖（X必包含键）。
6. **[第五范式](https://baike.baidu.com/item/%E7%AC%AC%E4%BA%94%E8%8C%83%E5%BC%8F/5025271" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E8%8C%83%E5%BC%8F/_blank)**（5NF，又称完美范式）:如果[关系模式](https://baike.baidu.com/item/%E5%85%B3%E7%B3%BB%E6%A8%A1%E5%BC%8F/1237324" \t "https://baike.baidu.com/item/%E7%AC%AC%E4%BA%94%E8%8C%83%E5%BC%8F/_blank)R中的每一个连接依赖均由R的[候选码](https://baike.baidu.com/item/%E5%80%99%E9%80%89%E7%A0%81/761615" \t "https://baike.baidu.com/item/%E7%AC%AC%E4%BA%94%E8%8C%83%E5%BC%8F/_blank)所隐含，则称此关系模式符合第五范式。

### 2.1.9说说分库与分表设计

**垂直分表:**拆分是基于关系型数据库中的“列”（字段）进行的。

**垂直分库:**基本的思路就是按照业务模块来划分出不同的数据库，而不是像早期一样将所有的数据表都放到同一个数据库中。

**水平分表:**横向分表，就是将表中不同的数据行按照一定规律分布到不同的数据库表中（这些表保存在同一个数据库中），这样来降低单表数据量，优化查询性能。

**水平分库:**唯一不同的就是将这些拆分出来的表保存在不同的数据中。

### 2.1.10分库与分表带来的分布式困境与应对之策（如何解决分布式下的分库分表，全局表？）

困境：数据迁移与扩容问题；表关联问题；分页与排序问题；分布式事务问题；分布式全局唯一ID；

### 2.1.11说说 SQL 优化之道

1. 负向条件查询不能使用索引（not in/not exists都不是好习惯--不走索引）
2. 前导模糊查询不能使用索引
3. 数据区分度不大的字段不宜使用索引
4. 在属性上进行计算不能命中索引
5. 如果业务大部分是单条查询，使用Hash索引性能更好（B-Tree索引的时间复杂度是O(log(n))；Hash索引的时间复杂度是O(1)）
6. 允许为null的列，查询有潜在大坑
7. 复合索引最左前缀，并不是值SQL语句的where顺序要和复合索引一致
8. 使用ENUM而不是字符串
9. 如果明确知道只有一条结果返回，limit 1能够提高效率
10. 把计算放到业务层而不是数据库层，除了节省数据的CPU，还有意想不到的查询缓存优化效果
11. 强制类型转换会全表扫描

### 2.1.12 MySQL遇到的死锁问题、如何排查与解决

**1、锁类型介绍：**

MySQL有三种锁的级别：页级、表级、行级。

**表级锁**：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高,并发度最低。（**每次操作锁住整张表。开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低；** ）

**行级锁**：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。（ **每次操作锁住一行数据。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；**）

**页面锁**：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

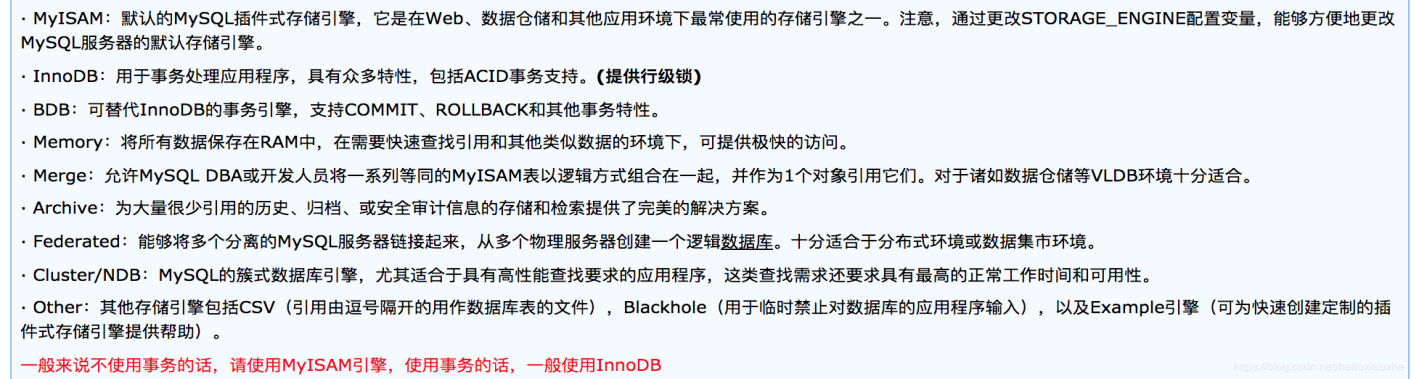
**2、产生原因：**

        所谓死锁<DeadLock>：是指两个或两个以上的进程在执行过程中,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.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称为死锁进程。表级锁不会产生死锁.所以解决死锁主要还是针对于最常用的InnoDB。

死锁的关键在于：两个(或以上)的Session加锁的顺序不一致。

那么对应的解决死锁问题的关键就是：**让不同的session加锁有次序**

### 2.1.13存储引擎的 InnoDB与MyISAM区别，优缺点，使用场景



**MyISAM：**由早期的 ISAM （Indexed Sequential Access Method：有索引的顺序访问方法）所改良。虽然性能极佳，而且提供了大量的特性，包括全文索引、压缩、空间函数等，但MyISAM不支持事务和行级锁，而且最大的缺陷就是崩溃后无法安全恢复。

**MyISAM特点：**

不支持行锁(MyISAM只有表锁)，读取时对需要读到的所有表加锁，写入时则对表加排他锁；

不支持事务

不支持外键

不支持崩溃后的安全恢复

在表有读取查询的同时，支持往表中插入新纪录

支持BLOB和TEXT的前500个字符索引，支持全文索引

支持延迟更新索引，极大地提升了写入性能

对于不会进行修改的表，支持 压缩表 ，极大地减少了磁盘空间的占用

**InnoDB特点：**

支持行锁，采用MVCC来支持高并发，有可能死锁

支持事务

支持外键

支持崩溃后的安全恢复

不支持全文索引

**MyISAM和InnoDB两者的**具体区别：****

|  |  |  |
| --- | --- | --- |
| **内容** | **MyISAM** | **InnoDB** |
| **事务支持** | **不支持** | **支持** |
| **存储结构** | **每个MyISAM在磁盘上存储成三个文件。第一个文件的名字以表的名字开始，扩展名指出文件类型。.frm文件存储表定义。数据文件的扩展名为.MYD (MYData)。索引文件的扩展名是.MYI (MYIndex)。** | **所有的表都保存在同一个数据文件中（也可能是多个文件，或者是独立的表空间文件），InnoDB表的大小只受限于操作系统文件的大小，一般为2GB。** |
| **存储空间** | **可被压缩，存储空间较小。支持三种不同的存储格式：静态表(默认，但是注意数据末尾不能有空格，会被去掉)、动态表、压缩表。** | **需要更多的内存和存储，它会在主内存中建立其专用的缓冲池用于高速缓冲数据和索引。** |
| **可移植性、备份及恢复** | **数据是以文件的形式存储，所以在跨平台的数据转移中会很方便。在备份和恢复时可单独针对某个表进行操作。** | **免费的方案可以是拷贝数据文件、备份 binlog，或者用 mysqldump，在数据量达到几十G的时候就相对痛苦了。** |
| **事务支持** | **不支持** | **支持** |
| **AUTO\_INCREMENT** | 可以和其他字段一起建立联合索引。 | InnoDB中必须包含只有该字段的索引。 |
| **表锁差异** | **只支持表级锁** | **支持事务和行级锁** |
| **全文索引** | **支持(FULLTEXT类型的)全文索引** | **不支持(FULLTEXT类型的)全文索引，但是innodb可以使用sphinx插件支持全文索引，并且效果更好** |
| **表主键** | 允许没有任何索引和主键的表存在，索引都是保存行的地址。 | 如果没有设定主键或者非空唯一索引，就会自动生成一个6字节的主键(用户不可见)，数据是主索引的一部分，附加索引保存的是主索引的值。InnoDB的主键范围更大，最大是MyISAM的2倍。 |
| **表的具体行数** | 保存有表的总行数，如果select count(\*) from table;会直接取出出该值。 | 没有保存表的总行数(只能遍历)，如果使用select count(\*) from table；就会遍历整个表，消耗相当大，但是在加了wehre条件后，myisam和innodb处理的方式都一样。 |
| **CURD操作** | 如果执行大量的SELECT，MyISAM是更好的选择。 | 如果你的数据执行大量的INSERT或UPDATE，出于性能方面的考虑，应该使用InnoDB表。 |
| **外键** | 不支持 | 支持 |
| **查询效率** | **没有where的count(\*)使用MyISAM要比InnoDB快得多。因为MyISAM内置了一个计数器，count(\*)时它直接从计数器中读，而InnoDB必须扫描全表。** |  |

**MyISAM和InnoDB两者的应用场景：**

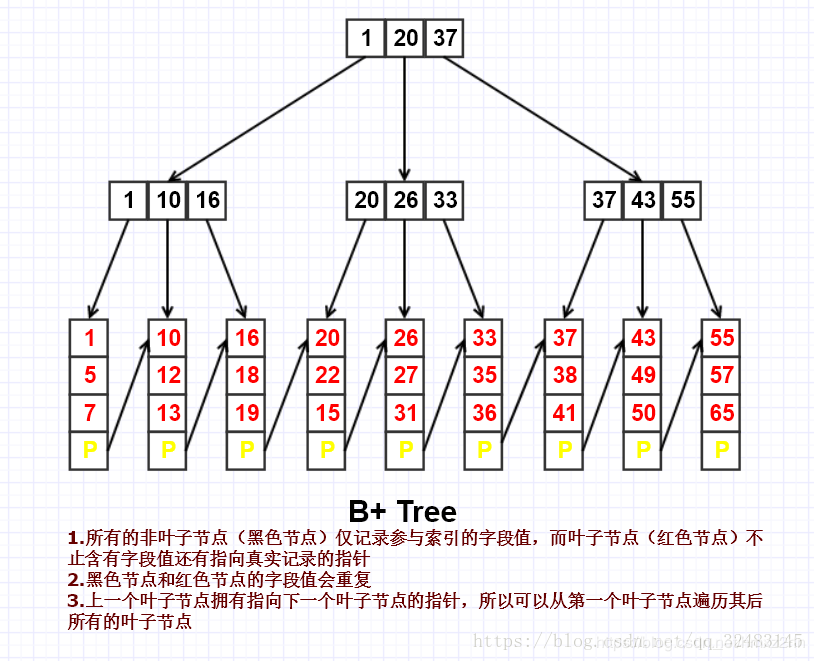
1) MyISAM管理非事务表。它提供高速存储和检索，以及全文搜索能力。如果应用中需要执行大量的SELECT查询，那么MyISAM是更好的选择。

2) InnoDB用于事务处理应用程序，具有众多特性，包括ACID事务支持。如果应用中需要执行大量的INSERT或UPDATE操作，则应该使用InnoDB，这样可以提高多用户并发操作的性能。

### 2.1.14 索引类别（B+树索引、全文索引、哈希索引）、索引的原理

索引类别：

1. B+树索引：使用B-tree数据结构来存储数据（实际上一般使用的是B+tree，即每一个叶子节点都包含指向下一个叶子节点的指针，为了方便叶子节点的范围遍历）



**说明：**1.所有的非叶子节点（黑色节点）仅记录参与索引的字段值，而叶子节点（红色节点）不止含有字段值还有指向真实记录的指针；

2.黑色节点和红色节点的字段值会重复；

3.上一个叶子节点群有指向下一个叶子节点的指针，所以可以从第一个叶子节点遍历其后所有的叶子节点。

**优点：**

全值匹配：指的是和索引中所有列进行匹配。

匹配最左前缀：假设以(姓，名，出生日期)三个数据项建立复合索引，可以查找所有姓张的人

匹配列前缀：假设有姓为司徒，司马的人，我们也可以查找第一列的前缀部分，如查找所有以司开头的姓的人

匹配范围值：可以查找所有在李和张之间的姓的人，注意范围查询只在复合索引的优先排序的第一列。（假设姓名按照拼音排序）

精确匹配前面列并范围匹配后一列：可以查找姓名为张三并出生日期在2000-12-12之后的人，注意第一个范围查询后面的列无法再使用索引查询

只访问索引的查询：即查询只需访问索引，而无需访问数据行。（此时应想到索引中的覆盖索引）

**缺点：**

如果不是按照索引的最左列开始查找，则无法使用索引。如无法查找名为龙的人，也无法查找在2000-12-12之后出生的人，当然也无法查找姓中以龙结尾的人（注意为和含有的区别）

不能跳过索引中的列：无法查找姓李并在2000-12-12之后出生的人

如果查询中包括某个列的范围查询，则其右边所有列都无法使用索引优化查询

1. 全文索引:通过查找文本中的关键词，类似于搜索引擎，而不是简单的where条件匹配。

注意：MySQL 5.6 以前的版本，只有 MyISAM 存储引擎支持全文索引；

MySQL 5.6 及以后的版本，MyISAM 和 InnoDB 存储引擎均支持全文索引;

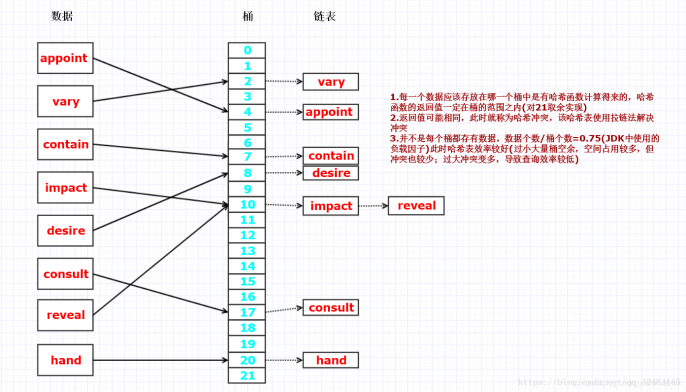
**只有字段的数据类型为 char、varchar、text 及其系列才可以建全文索引。**

1. 哈希索引:哈希索引是基于哈希表实现的。只有精确匹配索引所有列的的查询才有效。他的实现是存储殷勤会对每一行数据的索引列计算哈希码，并将哈希码和指向该记录的指针维护起来，对于hash相同的，采用链表的方式解决冲突。

说明：1.每一个数据都应该存放在哪一个桶中是有哈希函数计算得来的，哈希函数的返回值一定在捅得范围之内（对21取余实现）

2.返回值可能相同，此时称为哈希冲突，该哈希表采用拉链法解决冲突；

3.并不是每个捅都有数据，数据个数/捅个数=0.75（JDK中使用的负载因子）此时哈希表效率较好（过小大量捅空余，空间占用较多，但冲突较少，过大冲突较多，导致查询效率较低）



**哈希索引优点：**

快速查询：参与索引的字段只要进行Hash运算之后就可以快速定位到该记录，时间复杂度约为1

**哈希索引缺点：**

哈希索引只包含哈希值和行指针，所以不能用索引中的值来避免读取行

哈希索引数据并不是按照索引值顺序存储的，所以也就无法用于排序和范围查询

哈希索引也不支持部分索引列查询，因为哈希索引始终是使用索引列的全部数据进行哈希计算的。

哈希索引只支持等值比较查询，如=，IN()，<=>操作

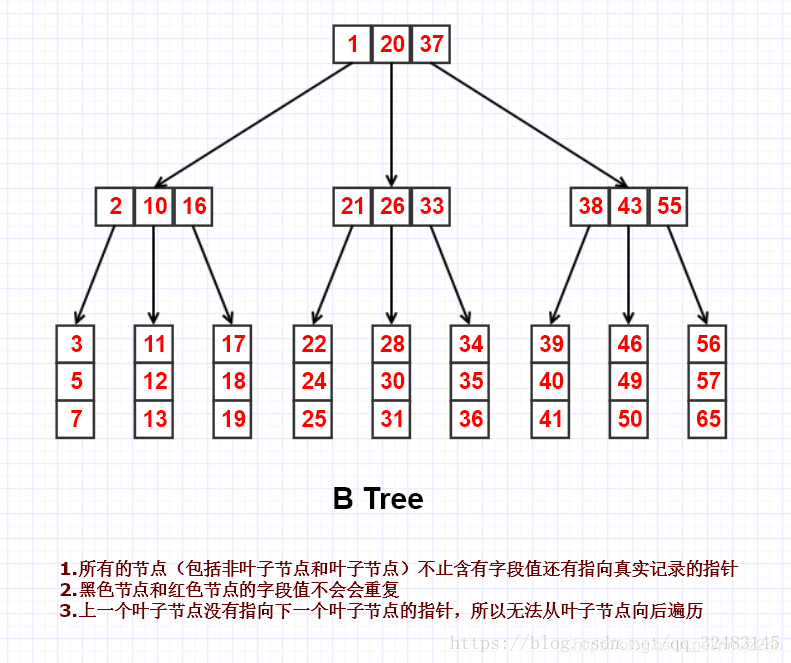
如果哈希冲突较多，一些索引的维护操作的代价也会更高

1. B Tree索引：

说明：1.所有的节点（包含非叶子节点和叶子节点）不止含有字段值还有指向真实记录的指针；

2.黑色节点和红色节点的字段值不会重复；

3.上一个叶子节点没有指向下一个叶子节点的指针，所以无法从叶子节点向后遍历



B+树相对于B树有一些自己的优势，可以归结为下面几点。

单一节点存储的元素更多，使得查询的IO次数更少，所以也就使得它更适合做为数据库MySQL的底层数据结构了。

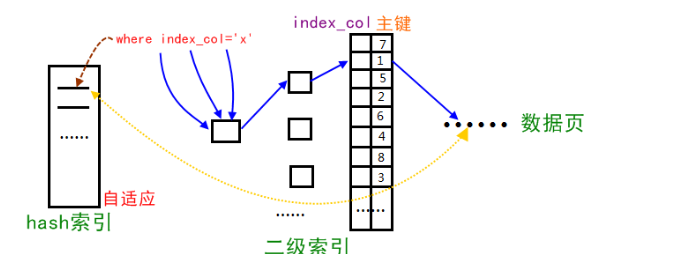
所有的查询都要查找到叶子节点，查询性能是稳定的，而B树，每个节点都可以查找到数据，所以不稳定。

所有的叶子节点形成了一个有序链表，更加便于查找。

**只有memory引擎显示支持哈希索引(注：为非唯一哈希索引)，NDB支持唯一哈希索引，innodb支持自适应哈希索引**

### 2.1.15什么是自适应哈希索引（AHI）

**1.自适应hash索引原理：**



1. Innodb存储引擎会监控对表上二级索引的查找，如果发现某二级索引被频繁访问，二级索引成为热数据，建立哈希索引可以带来速度的提升

|  |
| --- |
| mysql> show variables like '%ap%hash\_index';+----------------------------+-------+| Variable\_name | Value |+----------------------------+-------+| innodb\_adaptive\_hash\_index | ON |+----------------------------+-------+**1** row in set (**0.01** sec) |

1. 经常访问的二级索引数据会自动被生成到hash索引里面去(最近连续被访问三次的数据)，自适应哈希索引通过缓冲池的B+树构造而来，因此建立的速度很快。

**2.特点**

　　1、无序，没有树高

　　2、降低对二级索引树的频繁访问资源

　　　　索引树高<=4，访问索引：访问树、根节点、叶子节点

　　3、自适应

**3.缺陷**

　　1、hash自适应索引会占用innodb buffer pool；

　　2、自适应hash索引只适合搜索等值的查询，如select \* from table where index\_col='xxx'，而对于其他查找类型，如范围查找，是不能使用的；

3、极端情况下，自适应hash索引才有比较大的意义，可以降低逻辑读。

### 2.1.16为什么要用 B+tree作为MySQL索引的数据结构（未完成）

1.文件很大，不可能全部存储在内存中，故要存储到磁盘上

2.索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数（为什么使用B-/+Tree，还跟磁盘存取原理有关。）

3.局部性原理与磁盘预读，预读的长度一般为页（page）的整倍数，（在许多操作系统中，页得大小通常为4k）

4.数据库系统巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入，(由于节点中有两个数组，所以地址连续)。而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性

二叉查找树进化品种的红黑树等数据结构也可以用来实现索引，但是文件系统及数据库系统普遍采用B-/+Tree作为索引结构。

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，相对于内存存取，I/O存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘I/O操作次数的渐进复杂度。换句话说，索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数。为什么使用B-/+Tree，还跟磁盘存取原理有关。

### 2.1.17聚集索引与非聚集索引的区别

**聚集索引**表示表中存储的数据按照索引的顺序存储，检索效率比非聚集索引高，但对数据更新影响较大。

**非聚集索引**表示数据存储在一个地方，索引存储在另一个地方，索引带有指针指向数据的存储位置，非聚集索引检索效率比聚集索引低，但对数据更新影响较小。

**聚集索引：**该索引中键值的逻辑顺序决定了表中相应行的物理顺序。

聚集索引确定表中数据的物理顺序。聚集索引类似于电话簿，后者按姓氏排列数据。由于聚集索引规定数据在表中的物理存储顺序，因此一个表只能包含一个聚集索引。但该索引可以包含多个列（组合索引），就像电话簿按姓氏和名字进行组织一样。

**聚集索引使用注意事项**

定义聚集索引键时使用的列越少越好。

• 包含大量非重复值的列。

.• 使用下列运算符返回一个范围值的查询：BETWEEN、>、>=、< 和 <=。

• 被连续访问的列。

• 回大型结果集的查询。

• 经常被使用联接或 GROUP BY 子句的查询访问的列；一般来说，这些是外键列。对 ORDER BY 或 GROUP BY 子句中指定的列进行索引，可以使 SQL Server 不必对数据进行排序，因为这些行已经排序。这样可以提高查询性能。

• OLTP 类型的应用程序，这些程序要求进行非常快速的单行查找（一般通过主键）。应在主键上创建聚集索引。

**聚集索引不适用于：**

• 频繁更改的列 。这将导致整行移动（因为 SQL Server 必须按物理顺序保留行中的数据值）。这一点要特别注意，因为在大数据量事务处理系统中数据是易失的。

• 宽键 。来自聚集索引的键值由所有非聚集索引作为查找键使用，因此存储在每个非聚集索引的叶条目内。

**非聚集索引：**数据存储在一个地方，索引存储在另一个地方，索引带有指针指向数据的存储位置。

非聚集索引中的项目按索引键值的顺序存储，而表中的信息按另一种顺序存储（这可以由聚集索引规定）。对于非聚集索引，可以为在表非聚集索引中查找数据时常用的每个列创建一个非聚集索引。有些书籍包含多个索引。例如，一本介绍园艺的书可能会包含一个植物通俗名称索引，和一个植物学名索引，因为这是读者查找信息的两种最常用的方法。

**何时使用：**



### 2.1.18遇到过索引失效的情况没，什么时候可能会出现，如何解决

失效情况汇总：

列与列对比；

存在NULL值条件；

NOT条件；类似有：<>、NOT、in、not exists

LIKE通配符；(使用后置通配符.避免模糊查询失效)

条件上包括函数；类似有：to\_char、to\_date、to\_number、trunc等

复合索引前导列区分大；

数据类型的转换；

Connect By Level；

谓词运算；

Vistual Index；

Invisible Index；

### 2.1.19 limit 20000 加载很慢怎么解决

前端加缓存，或者其他方式，减少落到库的查询操作，例如某些系统中数据在搜索引擎中有备份的，可以用 es 等进行搜索

使用延迟关联，即先通用 limit 得到需要数据的索引字段，然后再通过原表和索引字段关联获得需要数据

从业务上实现，不分页如此多，例如只能分页前 100 页，后面的不允许再查了

不使用 limit N,M, 而是使用 limit N，即将 offset 转化为 where 条件。

### 2.1.20如何选择合适的分布式主键方案

具体内容后续可见：<https://blog.csdn.net/chenshun123/article/details/79588208>

**在集群环境下除自增ID 外的其它创建主键的方案**

1> 通过应用程序生成一个 GUID，然后和数据一起插入切分后的集群

优点 : 维护简单，实现也容易

缺点 : 应用的计算成本较大，且 GUID 的长度比较长，占用数据库存储空间较大，涉及到应用的开发

2> 通过独立的应用程序事先在数据库中生成一系列唯一的 ID，各应用程序通过接口或者自己去读取再和数据一起插入到切分后的集群中

优点 : 全局唯一主键简单，维护相对容易

缺点 : 实现复杂，需要应用开发，且 ID表 要频繁查和频繁更新，插入数据时，影响性能

3> 通过 中心数据库服务器 利用数据库自身的自增类型 (如 MySQL 的 auto\_increment 字段)，或者自增对象 (如 Oracle 的 Sequence) 等先生成一个唯一ID 再和数据一起插入切分后的集群 (不推荐)

优点 : 没有特别明显的优点

缺点 : 实现较为复杂，且整体可用性维系在这个中心数据库服务器上，一旦这里 crash，所有的集群都无法进行插入操作

4> 通过集群编号加集群内的自增 (auto\_increment类型) 两个字段 共同组成唯一主键 (虽然是两个字段，但是这方式存储空间最小，仅仅多了一个 smallint 两个字节)

优点 : 实现简单，维护也比较简单，对应用透明

缺点 : 引用关联操作相对比较复杂，需要两个字段，主键占用空间较大，在使用 InnoDB 的时候这一点的副作用很明显

5> 通过设置每个集群中自增 ID 起始点 (auto\_increment\_offset)，将各个集群的 ID 进行绝对的分段来实现全局唯一。当遇到某个集群数据增长过快后，通过命令调整下一个 ID 起始位置跳过可能存在的冲突

优点 : 实现简单，且比较容易根据 ID 大小直接判断出数据处在哪个集群，对应用透明

缺点 : 维护相对较复杂，需要高度关注各个集群 ID 增长状况

6> 通过设置每个集群中自增 ID 起始点 (auto\_increment\_offset) 以及 ID 自增步长 (auto\_increment\_increment)，让目前每个集群的起始点错开 1，步长选择大于将来基本不可能达到的切分集群数，达到将 ID 相对分段的效果来满足全局唯一的效果 (避免重合需要多种方案结合)

实现 : N台数据库，第一台 mysql 主键从1 开始每次加 N，第二台从2开始每次加 N，以此类推，在获取数据时如果第一台 Server 获取失败，则从第二台 Server 上获取

优点 : 实现简单，后期维护简单，对应用透明

缺点 : 第一次设置相对较为复杂

**Redis 生成 ID**

当使用数据库来生成 ID 性能不够要求的时候，可以尝试使用 Redis 来生成ID。这主要依赖于 Redis 是单线程的，所以也可以用生成全局唯一的 ID。可以用 Redis 原子操作 INCR 和 INCRBY 来实现。可以使用 Redis 集群来获取更高的吞吐量。

优点 : 1> 不依赖于数据库，灵活方便，且性能优于数据库

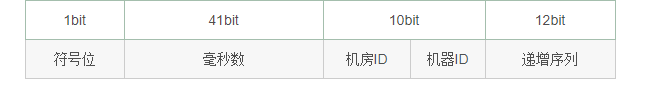
2> 数字ID天然排序，对分页或者需要排序的结果很有帮助

缺点 : 1> 如果系统中没有 Redis，还需要引入新的组件，增加系统复杂度

1. 需要编码和配置的工作量比较大

**Twitter 的 snowflake算法**

snowflake算法 的原理就是用 64位整数来表示主键，其结构如下图 :



1 bit符号位 : 表示正负，方便使用负数标识不正确的ID

41 bit毫秒时间 : 2^41 / (365 \* 24 \* 3600 \* 1000) ≈ 69年

10 bit机房ID + 机器ID : 最大值为1023

12 bit递增序列 : 最大值为4095

snowflake算法可以根据自身项目的需要进行一定的修改，比如估算未来的数据中心个数，每个数据中心的机器数以及统一毫秒可以能的并发数来调整在算法中所需要的 bit数

优点 : 1> 不依赖于数据库，灵活方便，且性能优于数据库

         2> ID按照时间在单机上是递增的

缺点 : 1> 在单机上是递增的，但是由于涉及到分布式环境，每台机器上的时钟不可能完全同步，也许有时候也会出现不是全局递增的情况

**利用 zookeeper生成唯一ID**

zookeeper 主要通过其 znode数据版本来生成序列号，可以生成32位和64位的数据版本号，客户端可以使用这个版本号来作为唯一的序列号。

很少会使用 zookeeper来生成唯一ID。主要是由于需要依赖 zookeeper，并且是多步调用API，如果在竞争较大的情况下，需要考虑使用分布式锁。因此，性能在高并发的分布式环境下，也不甚理想

**MongoDB 的 ObjectId**

MongoDB 的 ObjectId 和 snowflake 算法类似，它设计成轻量型的，不同的机器都能用全局唯一的同种方法方便地生成它。MongoDB 从一开始就设计用来作为分布式数据库，处理多个节点是一个核心要求，使其在分片环境中要容易生成得多。

**原文链接**：https://blog.csdn.net/chenshun123/article/details/79588208

### 2.1.21选择合适的数据存储方案

MySQL： 传统关系型数据库，事务性，复杂的SQL问题。

Redis：缓存、消息队列。

MongoDB：高伸缩性的场景，预期可扩展性的表结构。

Elasticsearch：全文搜索，提高模糊查询速度。

### 2.1.22常见的几种分布式ID的设计方案

**分布式ID的特性**

**唯一性**：确保生成的ID是全网唯一的。

**有序递增性**：确保生成的ID是对于某个用户或者业务是按一定的数字有序递增的。

**高可用性**：确保任何时候都能正确的生成ID。

**带时间**：ID里面包含时间，一眼扫过去就知道哪天的交易。

**分布式ID的生成方案**

**1. UUID**

算法的核心思想是结合机器的网卡、当地时间、一个随记数来生成UUID。

优点：本地生成，生成简单，性能好，没有高可用风险

缺点：长度过长，存储冗余，且无序不可读，查询效率低

**2. 数据库自增ID**

使用数据库的id自增策略，如 MySQL 的 auto\_increment。并且可以使用两台数据库分别设置不同步长，生成不重复ID的策略来实现高可用。

优点：数据库生成的ID绝对有序，高可用实现方式简单

缺点：需要独立部署数据库实例，成本高，有性能瓶颈

**3. 批量生成ID**

一次按需批量生成多个ID，每次生成都需要访问数据库，将数据库修改为最大的ID值，并在内存中记录当前值及最大值。

优点：避免了每次生成ID都要访问数据库并带来压力，提高性能

缺点：属于本地生成策略，存在单点故障，服务重启造成ID不连续

**4. Redis生成ID**

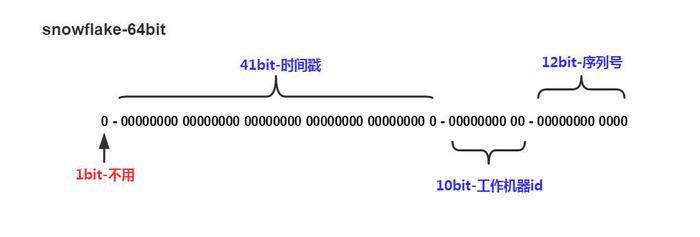
Redis的所有命令操作都是单线程的，本身提供像 incr 和 increby 这样的自增原子命令，所以能保证生成的 ID 肯定是唯一有序的。

优点：不依赖于数据库，灵活方便，且性能优于数据库；数字ID天然排序，对分页或者需要排序的结果很有帮助。

缺点：如果系统中没有Redis，还需要引入新的组件，增加系统复杂度；需要编码和配置的工作量比较大。

**5. Twitter的snowflake算法**

Twitter 利用 zookeeper 实现了一个全局ID生成的服务 Snowflake



**6. 百度UidGenerator**

UidGenerator是百度开源的分布式ID生成器，基于于snowflake算法的实现，看起来感觉还行。不过，国内开源的项目维护性真是担忧。

**7. 美团Leaf**

Leaf 是美团开源的分布式ID生成器，能保证全局唯一性、趋势递增、单调递增、信息安全，里面也提到了几种分布式方案的对比，但也需要依赖关系数据库、Zookeeper等中间件。

### 2.1.23常见的数据库优化方案，在你的项目中数据库如何进行优化的

### 2.1.24 Mysql 如何实现乐观锁 （\*\*\*）

1.使用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。

一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。

当提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，就进行更新操作，否则认为是过期数据。

2.第二种实现方式和第一种差不多，同样是在需要乐观锁控制的table中增加一个字段，名称无所谓，字段类型使用时间戳（timestamp）, 和上面的version类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致就更新，否则就是版本冲突。

### 2.1.25 mysql查询时on和where的区别（\*\*\*）

**on和where的选择条件的区别：**  
on后面的是连接条件，代表两个表建立关系所遵循的规则

where后面的可以看作是筛选条件，是对最终结果集进行过滤所遵循的规则

### 2.1.26 mysql 的编写顺序和执行顺序（\*\*\*）

一、sql执行顺序

(1)from

(3) join

(2) on

(4) where

(5)group by(开始使用select中的别名，后面的语句中都可以使用)

(6) avg,sum....

(7)having

(8) select

(9) distinct

(10) order by

二、sql 编写顺序：

<SELECT clause> [<**FROM** clause>] [<**WHERE** clause>] [<**GROUP BY** clause>] [<**HAVING** clause>] [<**ORDER BY** clause>] [<**LIMIT** clause>]

## 2.2、Redis

Redis 是一个开源（ BSD 许可）的，内存中的数据结构存储系统，它可以用作数据库、缓存和消息中间件。它支持的数据类型很丰富，如字符串、链表、集 合、以及散列等，并且还支持多种排序功能。

**相关命令**：

|  |
| --- |
| SET key value 设置key=value  GET key 或者键key对应的值  GETRANGE key start end 得到字符串的子字符串存放在一个键  **GETSET** key value 设置键的字符串值，并返回旧值  GETBIT key offset 返回存储在键位值的字符串值的偏移  MGET key1 [key2..] 得到所有的给定键的值  SETBIT key offset value 设置或清除该位在存储在键的字符串值偏移  SETEX key seconds value 键到期时设置值  SETNX key value 设置键的值，只有当该键不存在  SETRANGE key offset value 覆盖字符串的一部分从指定键的偏移  STRLEN key 得到存储在键的值的长度  MSET key value [key value...] 设置多个键和多个值  MSETNX key value [key value...] 设置多个键多个值，只有在当没有按键的存在时  PSETEX key milliseconds value 设置键的毫秒值和到期时间  INCR key 增加键的整数值一次  INCRBY key increment 由给定的数量递增键的整数值  INCRBYFLOAT key increment 由给定的数量递增键的浮点值  DECR key 递减键一次的整数值  DECRBY key decrement 由给定数目递减键的整数值  APPEND key value 追加值到一个键 |

**数据管理**操作：

|  |
| --- |
| DEL key 如果存在删除键  DUMP key 返回存储在指定键的值的序列化版本  EXISTS key 此命令检查该键是否存在  EXPIRE key seconds 指定键的过期时间  EXPIREAT key timestamp 指定的键过期时间。在这里，时间是在Unix时间戳格式  PEXPIRE key milliseconds 设置键以毫秒为单位到期  PEXPIREAT key milliseconds-timestamp 设置键在Unix时间戳指定为毫秒到期  KEYS pattern 查找与指定模式匹配的所有键  MOVE key db 移动键到另一个数据库  PERSIST key 移除过期的键  PTTL key 以毫秒为单位获取剩余时间的到期键。  TTL key 获取键到期的剩余时间。  RANDOMKEY 从Redis返回随机键  RENAME key newkey 更改键的名称  RENAMENX key newkey 重命名键，如果新的键不存在  TYPE key 返回存储在键的数据类型的值。 |

### 2.2.1 Redis 有哪些数据类型（\*\*\*）

string（字符串）

hash（哈希）

list（列表）

set（集合）

zset(sorted set：有序集合)

### 2.2.2 Redis 内部结构

Redis是一种key/value型数据库，其中，每个key和value都是使用对象表示的。每个Key都是一个字符串对象, 每个value是一个redisObject对象.

1. 简单动态字符串

底层数据结构为SDS(simple dynamic string)，即简单动态字符串.

2. 链表

链表是一种比较常见的数据结构，易于插入和删除，但随机访问困难。

列表list的底层实现就是链表(quicklist=链表+ziplist)，此外，Redis的发布和订阅，慢查询，监视器等也都用到链表

3. 字典

底层是key-value数据库.

4. 跳跃表skiplist

时间复杂度：平均O(logN),最坏O(N)

用途： sortedset

顺序性操作批量处理

数据结构：有序集合的底层实现之一（元素比较多，元素均为长的字符串），集群节点内部

5.整数集合intset

用途：set

类型：int16\_t,int32\_t,int64\_t

属性：encoding，length，int8\_t []

升级：新增整数大小超过之前的范围

6.压缩列表ziplist

用途：列表键list，哈希键hashtable

使用场景：列表键包含少量列表项，每个列表项均不是长字符串（小整数或短字符串）

### 2.2.3 Redis 使用场景

一：缓存——热数据

热点数据（经常会被查询，但是不经常被修改或者删除的数据），首选是使用redis缓存，毕竟强大到冒泡的QPS和极强的稳定性不是所有类似工具都有的，而且相比于memcached还提供了丰富的数据类型可以使用，另外，内存中的数据也提供了AOF和RDB等持久化机制可以选择，要冷、热的还是忽冷忽热的都可选。

二：计数器

诸如统计点击数等应用。由于单线程，可以避免并发问题，保证不会出错，而且100%毫秒级性能！

三：队列

相当于消息系统，ActiveMQ，RocketMQ等工具类似，但是个人觉得简单用一下还行，如果对于数据一致性要求高的话还是用RocketMQ等专业系统。

由于redis把数据添加到队列是返回添加元素在队列的第几位，所以可以做判断用户是第几个访问这种业务

队列不仅可以把并发请求变成串行，并且还可以做队列或者栈使用

四：位操作（大数据处理）

五：分布式锁与单线程机制

验证前端的重复请求（可以自由扩展类似情况），可以通过redis进行过滤：每次请求将request Ip、参数、接口等hash作为key存储redis（幂等性请求），设置多长时间有效期，然后下次请求过来的时候先在redis中检索有没有这个key，进而验证是不是一定时间内过来的重复提交

秒杀系统，基于redis是单线程特征，防止出现数据库“爆破”

全局增量ID生成，类似“秒杀”

六：最新列表

七：排行榜

### 2.2.4 Redis 持久化机制

**什么叫持久化？**

用一句话可以将持久化概括为：将数据（如内存中的对象）保存到可永久保存的存储设备中。持久化的主要应用是将内存中的对象存储在数据库中，或者存储在磁盘文件中、 XML 数据文件中等等。

**Redis 怎么实现持久化？**

在设计之初，Redis 就已经考虑到了这个问题。官方提供了多种不同级别的数据持久化的方式：

1、RDB持久化方式能够在指定的时间间隔能对你的数据进行快照存储。

2、AOF持久化方式记录每次对服务器写的操作,当服务器重启的时候会重新执行这些命令来恢复原始的数据,AOF命令以redis协议追加保存每次写的操作到文件末尾.Redis还能对AOF文件进行后台重写,使得AOF文件的体积不至于过大。

3、如果你只希望你的数据在服务器运行的时候存在,你也可以不使用任何持久化方式。

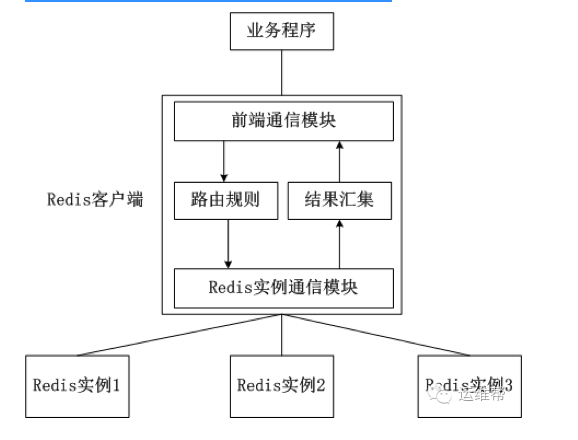
4、你也可以同时开启两种持久化方式, 在这种情况下, 当redis重启的时候会优先载入AOF文件来恢复原始的数据,因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整。

可参考《使用快照和AOF将Redis数据持久化到硬盘中》

### 2.2.5 Redis 集群方案与实现

1、客户端分片

客户端分片是把分片的逻辑放在Redis客户端实现，通过Redis客户端预先定义好的路由规则，把对Key的访问转发到不同的Redis实例中，最后把返回结果汇集。



客户端分片的好处是所有的逻辑都是可控的，不依赖于第三方分布式中间件。开发人员清楚怎么实现分片、路由的规则，不用担心踩坑。

客户端分片方案有下面这些缺点：

　　●这是一种静态的分片方案，需要增加或者减少Redis实例的数量，需要手工调整分片的程序。

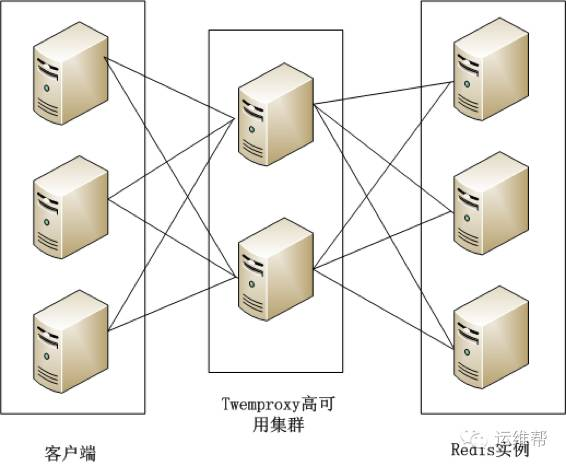
　　●可运维性差，集群的数据出了任何问题都需要运维人员和开发人员一起合作，减缓了解决问题的速度，增加了跨部门沟通的成本。

●在不同的客户端程序中，维护相同的分片逻辑成本巨大。例如，系统中有两套业务系统共用一套Redis集群，一套业务系统用Java实现，另一套业务系统用PHP实现。为了保证分片逻辑的一致性，在Java客户端中实现的分片逻辑也需要在PHP客户端实现一次。相同的逻辑在不同的系统中分别实现，这种设计本来就非常糟糕，而且需要耗费巨大的开发成本保证两套业务系统分片逻辑的一致性。

2、Twemproxy

Twemproxy是由Twitter开源的Redis代理，其基本原理是：Redis客户端把请求发送到Twemproxy，Twemproxy根据路由规则发送到正确的Redis实例，最后Twemproxy把结果汇集返回给客户端。

Twemproxy通过引入一个代理层，将多个Redis实例进行统一管理，使Redis客户端只需要在Twemproxy上进行操作，而不需要关心后面有多少个Redis实例，从而实现了Redis集群。



Twemproxy的优点如下：

　　●客户端像连接Redis实例一样连接Twemproxy，不需要改任何的代码逻辑。

　　●支持无效Redis实例的自动删除。

　　●Twemproxy与Redis实例保持连接，减少了客户端与Redis实例的连接数。

Twemproxy的缺点如下：

　　●由于Redis客户端的每个请求都经过Twemproxy代理才能到达Redis服务器，这个过程中会产生性能损失。

　　●没有友好的监控管理后台界面，不利于运维监控。

　　●最大的问题是Twemproxy无法平滑地增加Redis实例。对于运维人员来说，当因为业务需要增加Redis实例时工作量非常大。

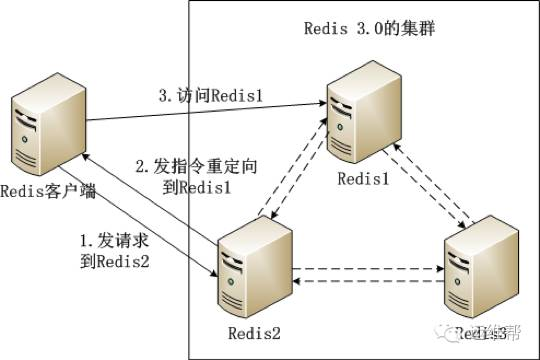
Twemproxy作为最被广泛使用、最久经考验、稳定性最高的Redis代理，在业界被广泛使用。

3、Redis 3.0集群

Redis 3.0集群采用了P2P的模式，完全去中心化。Redis把所有的Key分成了16384个slot，每个Redis实例负责其中一部分slot。集群中的所有信息（节点、端口、slot等），都通过节点之间定期的数据交换而更新。

Redis客户端在任意一个Redis实例发出请求，如果所需数据不在该实例中，通过重定向命令引导客户端访问所需的实例。

Redis 3.0集群的工作流程如下图所示：



### 2.2.6 Redis 为什么是单线程的？（\*\*\*）

1. **Redis服务端是个单线程的架构**，不同的Client虽然看似可以同时保持连接，但发出去的命令是序列化执行的，这在通常的数据库理论下是最高级别的隔离（serialize）  
2. 用MULTI/EXEC 来把多个命令组装成一次发送，达到原子性  
3. 用WATCH提供的乐观锁功能，在你EXEC的那一刻，如果被WATCH的键发生过改动，则MULTI到EXEC之间的指令全部不执行，不需要rollback  
4. 其他回答中提到的DISCARD指令只是用来撤销EXEC之前被暂存的指令，并不是回滚

### 2.2.7缓存雪崩、缓存穿透、缓存预热、缓存更新、缓存降级

**缓存雪崩**

缓存雪崩是指在我们设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到DB，DB瞬时压力过重雪崩。

**缓存服务降级**

服务降级的目的，是为了防止Redis服务故障，导致数据库跟着一起发生雪崩问题。因此，对于不重要的缓存数据，可以采取服务降级策略，例如一个比较常见的做法就是，Redis出现问题，不去数据库查询，而是直接返回默认值给用户。

**缓存预热**

在新启动的缓存系统中，如果没有任何数据，在重建缓存数据过程中，系统的性能和数据库复制都不太好，那么最好的缓存系统启动时就把热点数据加载好，例如对于缓存信息，在启动缓存加载数据库中全部数据进行预热。一般情况下，我们会开通一个同步数据的接口，进行缓存预热。

**缓存穿透**

如果因为不恰当的业务，或者恶意攻击持续地发请求某些不存在的数据，由于缓存没有保存该数据，所有的请求都会落到数据库上，会对数据库造成很大压力，甚至奔溃。一个简单的对策是将不存在的数据也缓存起来。

**缓存更新**

使用缓存过程中，我们经常会遇到缓存数据的不一致性和与脏读现象，我们有什么解决策略呢？

一般情况下，我们采取缓存双淘汰机制，在更新数据库的时候淘汰缓存。此外，设定超时时间，例如30分钟。极限场景下，即使有脏数据入cache，这个脏数据也最多存在三十分钟。

### 2.2.8使用缓存的合理性问题

**热点数据，缓存才有价值**

对于冷数据而言，大部分数据可能还没有再次访问到就已经被挤出内存，不仅占用内存，而且价值不大。

对于热点数据，比如我们的某IM产品，生日祝福模块，当天的寿星列表，缓存以后可能读取数十万次。再举个例子，某导航产品，我们将导航信息，缓存以后可能读取数百万次。

**频繁修改的数据，看情况考虑使用缓存**

数据更新前至少读取两次，缓存才有意义。这个是最基本的策略，如果缓存还没有起作用就失效了，那就没有太大价值了。

对于上面两个例子，寿星列表、导航信息都存在一个特点，就是信息修改频率不高，读取通常非常高的场景。

那存不存在，修改频率很高，但是又不得不考虑缓存的场景呢？有！比如，这个读取接口对数据库的压力很大，但是又是热点数据，这个时候就需要考虑通过缓存手段，减少数据库的压力，比如我们的某助手产品的，点赞数，收藏数，分享数等是非常典型的热点数据，但是又不断变化，此时就需要将数据同步保存到Redis缓存，减少数据库压力。

**数据不一致性**

一般会对缓存设置失效时间，一旦超过失效时间，就要从数据库重新加载，因此应用要容忍一定时间的数据不一致。还有一种是在数据更新时立即更新缓存，不过这也会更多系统开销和事务一致性问题。

### 2.2.9 Redis常见的回收策略

**volatile-lru**：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰

**volatile-ttl**：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰

**volatile-random**：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰

**allkeys-lru**：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

**allkeys-random**：从数据集（server.db[i].dict）中任意选择数据淘汰

**no-enviction**（驱逐）：禁止驱逐数据

### 2.2.10 Redis过期策略（\*\*\*）

Redis作为一个高性能的内存NoSQL数据库，其容量受到最大内存限制的限制。

运行时产生的额外内存，包括：

1.垃圾数据和过期Key所占空间

2.字典渐进式Rehash导致未及时删除的空间

3.Redis管理数据，包括底层数据结构开销，客户端信息，读写缓冲区等

4.主从复制，bgsave时的额外开销

5.其它

**Redis过期数据清理策略**

**过期数据清理时机**

为了防止一次性清理大量过期Key导致Redis服务受影响，Redis只在空闲时清理过期Key。

1. 访问Key时，会判断Key是否过期，逐出过期Key;
2. CPU空闲时在定期serverCron任务中，逐出部分过期Key;
3. 每次事件循环执行的时候，逐出部分过期Key;

**过期数据清理算法：**

1.Redis配置项hz定义了serverCron任务的执行周期，默认为10，即CPU空闲时每秒执行10次;

2.每次过期key清理的时间不超过CPU时间的25%，即若hz=1，则一次清理时间最大为250ms，若hz=10，则一次清理时间最大为25ms;

3.清理时依次遍历所有的db;

4.从db中随机取20个key，判断是否过期，若过期，则逐出;

5.若有5个以上key过期，则重复步骤4，否则遍历下一个db;

6.在清理过程中，若达到了25%CPU时间，退出清理过程;

## 2.3、消息队列（未完成）

### 2.3.1消息队列的使用场景

### 2.3.2消息的重发补偿解决思路

### 2.3.3 消息的幂等性解决思路

### 2.3.4消息的堆积解决思路

### 2.3.5自己如何实现消息队列

### 2.3.6如何保证消息的有序性

# 第三章 开源框架和容器

## 3.1、SSM/Servlet

### 3.1.0 MVC设计思想

M：modle 模型

V：view 视图

C：controller 控制器

MVC模式（Model–view–controller）是软件工程中的一种软件架构模式，把软件系统分为三个基本部分：模型（Model）、视图（View）和控制器（Controller）。一种软件设计典范，用一种业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑

控制器（Controller）- 负责转发请求，对请求进行处理。

视图（View） - 界面设计人员进行图形界面设计。

模型（Model） - 程序员编写程序应有的功能（实现算法等等）、数据库专家进行数据管理和数据库设计(可以实现具体的功能)

### 3.1.1 Servlet的生命周期（未完成）

### 3.1.2 转发与重定向的区别（未完成）

### 3.1.3 BeanFactory 和 ApplicationContext 有什么区别（未完成）

### 3.1.4 Spring

#### 3.1.4.1 Spring Bean 的生命周期

① 实例化Bean对象  
② 设置对象属性  
③ 检查是否实现Aware相关接口，并设置相关依赖  
④ 执行BeanPostProcessor前置处理  
⑤ 检查是否是InitializingBean以决定是否调用afterPropertiesSet方法  
⑥ 检查是否配置有自定义的init-method方法  
⑦ 执行BeanPostProcessor后置处理  
⑧ 注册必要的Destruction相关回调接口  
⑨ 使用中  
⑩ 是否实现DisposableBean接口，如果有，则执行相应的方法  
⑪ 是否配置有自定义的destroy方法，如果有则执行销毁

#### 3.1.4.2 Spring IOC 如何实现（\*\*\*）

**IOC: Inversion of Control ,即 "控制反转"** ， 不是什么技术，而是一种思想。原先需要自行实例化的对象， 交给IOC容器去实现。

**DI：Dependency Injection，即“依赖注入”**，动态的向某个对象提供它所需要的其他对象（反射原理--实例化一个类时，它通过反射调用类中set方法将事先保存在HashMap中的类属性注入到类中）

注入方式：

（1）接口注入

（2）setter

（3）构造器注入

**实现IOC的两种方式**

（1）IOC配置文件的方式

（2）IOC注解的方式

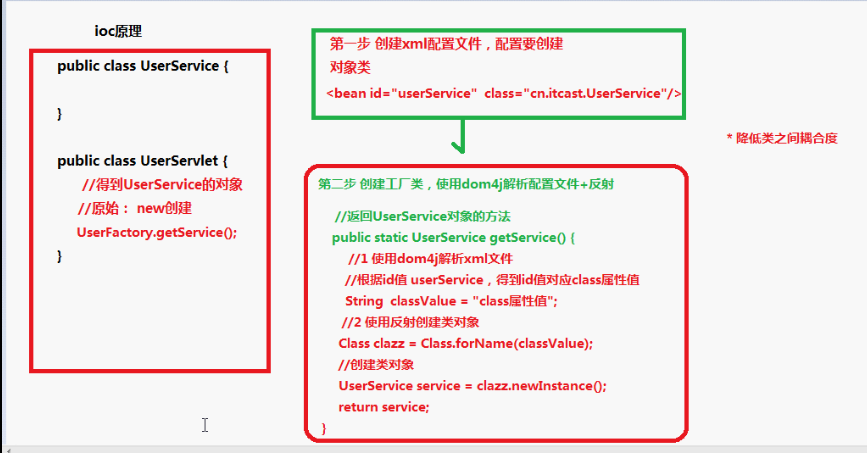
**IOC底层实现原理**

1.1 xml配置文件

1.2 dom4j解析xml

1.3 工厂模式

1.4 反射



Springboot 通过 @Bean 注解--实现

#### 3.1.4.5 Spring中Bean的作用域，默认的是哪一个

1）**singleton：**单例模式，使用 singleton定义的Bean在Spring容器中只有一个实例，这也是Bean默认的作用域。

2）**prototype**：原型模式，每次通过Spring容器获取prototype定义的Bean时，容器都将创建一个新的Bean实例。

3）**request**：在一次 HTTP 请求中，容器会返回该 Bean 的同一个实例。而对不同的 HTTP 请求，会返回不同的实例，该作用域仅在当前 HTTP Request 内有效。

4）**session**：在一次 HTTP Session 中，容器会返回该 Bean 的同一个实例。而对不同的 HTTP 请求，会返回不同的实例，该作用域仅在当前 HTTP Session 内有效。

5）**global Session**：在一个全局的 HTTP Session 中，容器会返回该 Bean 的同一个实例。该作用域仅在使用 portlet context 时有效。

#### 3.1.4.6 说说 Spring AOP、Spring AOP 实现原理（\*\*\*）

一、AOP是什么？

AOP的全称是Aspect Orient Programming，即面向切面编程。是对OOP（Object Orient Programming）的一种补充，战门用于处理一些具有横切性质的服务。常常用于日志输出、安全控制、事务管理、缓存等。

二、实现技术？

       AOP（这里的AOP指的是面向切面编程思想，而不是Spring AOP）主要的的实现技术主要有Spring AOP和AspectJ。

1、AspectJ的底层技术。

       AspectJ的底层技术是静态代理，即用一种AspectJ支持的特定语言编写切面，通过一个命令来编译，生成一个新的代理类，该代理类增强了业务类，这是在编译时增强，相对于下面说的运行时增强，编译时增强的性能更好。

2、Spring AOP

       Spring AOP采用的是动态代理，在运行期间对业务方法进行增强，所以不会生成新类，对于动态代理技术，Spring AOP提供了对JDK动态代理的支持以及CGLib的支持。

       JDK动态代理只能为接口创建动态代理实例，而不能对类创建动态代理。需要获得被目标类的接口信息（应用Java的反射技术），生成一个实现了代理接口的动态代理类（字节码），再通过反射机制获得动态代理类的构造函数，利用构造函数生成动态代理类的实例对象，在调用具体方法前调用invokeHandler方法来处理。

       CGLib动态代理需要依赖asm包，把被代理对象类的class文件加载进来，修改其字节码生成子类。

       但是Spring AOP基于注解配置的情况下，需要依赖于AspectJ包的标准注解，但是不需要额外的编译以及AspectJ的织入器，而基于XML配置不需要。

#### 3.1.4.7动态代理（CGLib 与 JDK）、优缺点、性能对比、如何选择

#### 3.1.4.8 Spring 事务实现方式、事务的传播机制、默认的事务类别（\*\*\*）

1、事务几种实现方式

（1）编程式事务管理对基于 POJO 的应用来说是唯一选择。我们需要在代码中调用beginTransaction()、commit()、rollback()等事务管理相关的方法，这就是编程式事务管理。

（2）基于 TransactionProxyFactoryBean的声明式事务管理

（3）基于 @Transactional 的声明式事务管理

（4）基于Aspectj AOP配置事务

2、事务的传播特性

事务传播行为就是多个事务方法调用时，如何定义方法间事务的传播。Spring定义了7中传播行为：

（1）propagation\_requierd：如果当前没有事务，就新建一个事务，如果已存在一个事务中，加入到这个事务中，这是Spring默认的选择。

（2）propagation\_supports：支持当前事务，如果没有当前事务，就以非事务方法执行。

（3）propagation\_mandatory：使用当前事务，如果没有当前事务，就抛出异常。

（4）propagation\_required\_new：新建事务，如果当前存在事务，把当前事务挂起。

（5）propagation\_not\_supported：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

（6）propagation\_never：以非事务方式执行操作，如果当前事务存在则抛出异常。

（7）propagation\_nested：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与propagation\_required类似的操作。

3、事务的隔离级别

（1）read uncommited：是最低的事务隔离级别，它允许另外一个事务可以看到这个事务未提交的数据。

（2）read commited：保证一个事物提交后才能被另外一个事务读取。另外一个事务不能读取该事物未提交的数据。

（3）repeatable read：这种事务隔离级别可以防止脏读，不可重复读。但是可能会出现幻象读。它除了保证一个事务不能被另外一个事务读取未提交的数据之外还避免了以下情况产生（不可重复读）。

（4）serializable：这是花费最高代价但最可靠的事务隔离级别。事务被处理为顺序执行。除了防止脏读，不可重复读之外，还避免了幻象读

（5）脏读、不可重复读、幻象读概念说明：

a.脏读：指当一个事务正字访问数据，并且对数据进行了修改，而这种数据还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。因为这个数据还没有提交那么另外一个事务读取到的这个数据我们称之为脏数据。依据脏数据所做的操作肯能是不正确的。

b.不可重复读：指在一个事务内，多次读同一数据。在这个事务还没有执行结束，另外一个事务也访问该同一数据，那么在第一个事务中的两次读取数据之间，由于第二个事务的修改第一个事务两次读到的数据可能是不一样的，这样就发生了在一个事物内两次连续读到的数据是不一样的，这种情况被称为是不可重复读。

c.幻象读：一个事务先后读取一个范围的记录，但两次读取的纪录数不同，我们称之为幻象读（两次执行同一条 select 语句会出现不同的结果，第二次读会增加一数据行，并没有说这两次执行是在同一个事务中）

#### 3.1.4.9 Spring 事务底层原理（未完成）

#### 3.1.4.10 Spring事务失效（事务嵌套）（未完成）

JDK动态代理给Spring事务埋下的坑，可参考《JDK动态代理给Spring事务埋下的坑！》

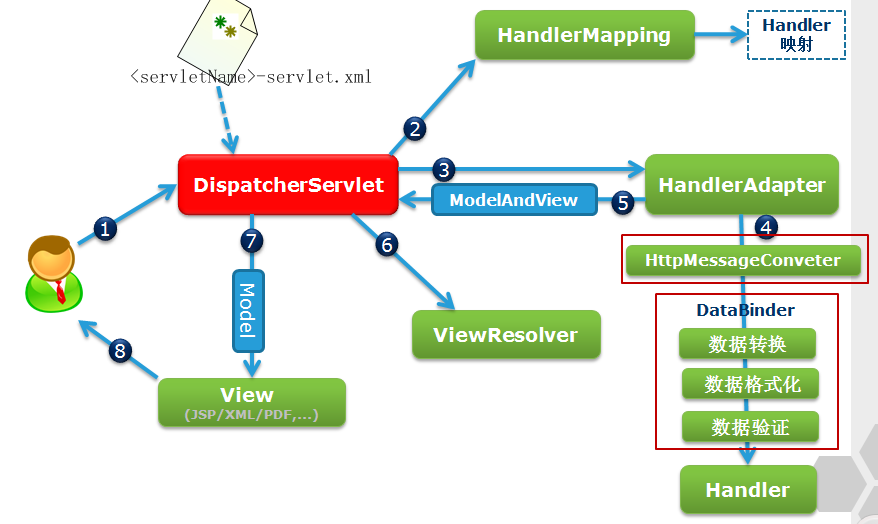
#### 3.1.4.11 如何自定义注解实现功能（未完成）

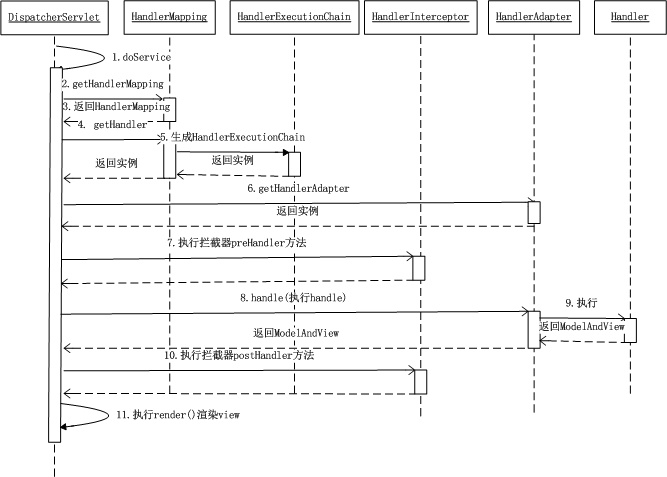
#### 3.1.4.12 Spring 的单例实现原理（未完成）

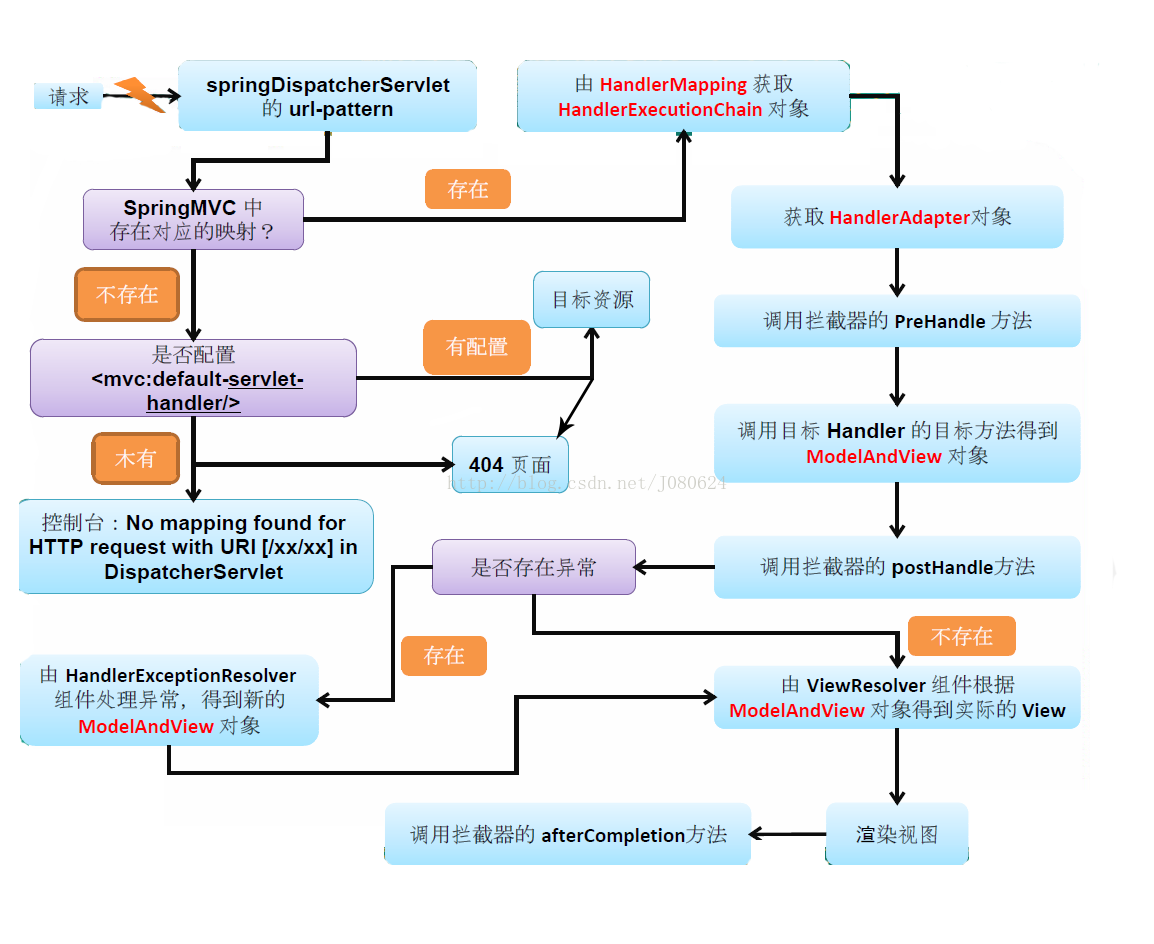
#### 3.1.4.13 Spring 框架中用到了哪些设计模式

### 3.1.5 Spring MVC 运行流程（\*\*\*）

**Spring MVC工作流程图**

****





****Spring工作流程描述****

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

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

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

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

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

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

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

      数据验证： 验证数据的有效性（长度、格式等），验证结果存储到BindingResult或Error中

      5.  Handler执行完成后，向DispatcherServlet 返回一个ModelAndView对象；

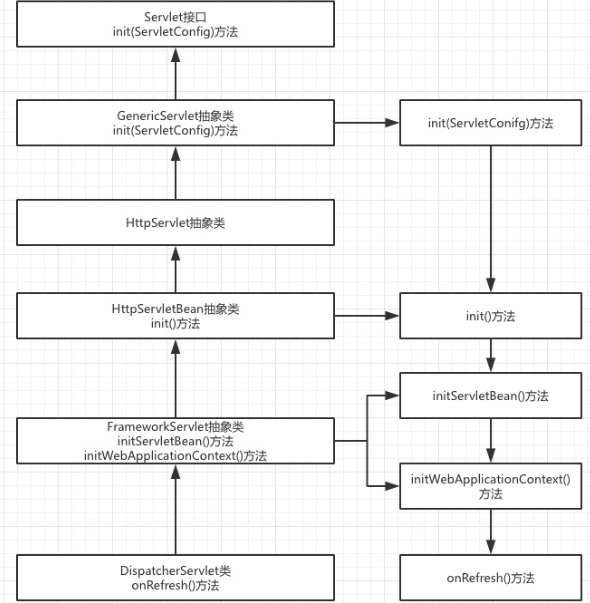
      6.  根据返回的ModelAndView，选择一个适合的ViewResolver（必须是已经注册到Spring容器中的ViewResolver)返回给DispatcherServlet ；

      7. ViewResolver 结合Model和View，来渲染视图

      8. 将渲染结果返回给客户端。

#### 3.1.5.1 Spring MVC 启动流程





### 3.1.6 Spring 其他产品

（Srping Boot、Spring Cloud、Spring Secuirity、Spring Data、Spring AMQP 等）

### 3.1.7 Spring Boot

#### 3.1.7.1有没有用到Spring Boot，Spring Boot的认识、原理

一、什么是SpringBoot?

SpringBoot是一个快速开发框架，快速的将一些常用的第三方依赖整合（原理：通过Maven子父工程的方式），简化XML配置，全部采用注解形式，内置Http服务器（Jetty和Tomcat），最终以java应用程序进行执行。

二、SpringBoot核心原理

1> 基于SpringMVC无配置文件（纯Java）完全注解化+内置tomcat-embed-core实现SpringBoot框架，Main函数启动。

1. SpringBoot核心快速整合第三方框架原理:Maven继承依赖关系。

可参考《为什么会有Spring》

可参考《为什么会有Spring AOP》

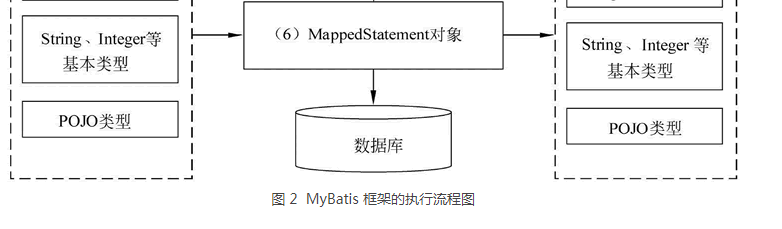
### 3.1.8 MyBatis的原理

MyBatis 的工作原理

1）读取 MyBatis 配置文件：mybatis-config.xml 为 MyBatis 的全局配置文件，配置了 MyBatis 的运行环境等信息，例如数据库连接信息。

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 的环境等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。

#### 3.1.8.1 #{}井花括号和${}刀乐花括号的区别是什么？

① #{}：是预编译处理。

② ${}：是字符串替换。

③ 使用#{}可以有效的防止SQL注入，提高系统的安全性

#### 3.1.8.2 Mybatis 动态SQL

mybatis中的动态SQL主要包含如下几种元素：if、choose、when、otherwise、trim、where、set以及foreach几种

#### 3.1.8.3 Mybatis 的一级缓存和二级缓存以及使用场景？

**一级缓存**

Mybatis的一级缓存是指Session缓存。一级缓存的作用域默认是一个SqlSession。Mybatis默认开启一级缓存。

**二级缓存**

Mybatis的二级缓存是指mapper映射文件。二级缓存的作用域是同一个namespace下的mapper映射文件内容，多个SqlSession共享。Mybatis需要手动设置启动二级缓存。

**使用情景**

如果不是在同一个事务中每个Mapper的每次查询操作都对应一个全新的SqlSession实例，这个时候就不会有一级缓存的命中，但是在同一个事务中时共用的是同一个SqlSession。  
如有需要可以启用二级缓存。

#### 3.1.8.4 Mybatis怎样生成Mapper接口的实现类执行sql

Mybatis运行时会使用JDK动态代理为Maper接口生成代理proxy对象，代理对象proxy会拦截接口方法，从而执行sql.

## 3.2、Netty

### 3.2.1为什么选择 Netty

### 3.2.2说说业务中，Netty 的使用场景

### 3.2.3原生的 NIO 在 JDK 1.7 版本存在 epoll bug

### 3.2.4什么是TCP 粘包/拆包

### 3.2.5 TCP粘包/拆包的解决办法

### 3.2.6 Netty 线程模型

### 3.2.7说说 Netty 的零拷贝

### 3.2.8 Netty 内部执行流程

### 3.2.9 Netty 重连实现

## 3.3、Tomcat

Tomcat的基础架构（Server、Service、Connector、Container）

Tomcat如何加载Servlet的

Pipeline-Valve机制

可参考：《四张图带你了解Tomcat系统架构！》

## 3.4、Linux

### 3.4.1 linux 常用命令

**1.查看目录结构命令：**

ls(list)功能：列出目录内容

**格式：ls[参数][文件或目录]**

-a或--all    所有文件和目录。注意隐藏文件、特殊目录、以“.”开头的和以“..”开头的

-l    使用详细格式列表

-t    用文件和目录的更改时间排序

-r   反向排序

--help   在线帮助

常用：

ls -l 列出当前目录下的文件信息（此命令很常用，简化的格式为 ll）

ls -al 列出当前目录下的文件信息（包括隐藏文件，特殊目录）

ll /home/ 显示根目录下home目录下的内容

**2.切换目录命令**

cd(change directory)功能：切换目录

**语法：cd[目录]**

常用：

cd   ~       当前用户主目录

cd   /        根目录

cd   -        上一次访问的目录

cd   ..       上一级目录

cd            当前用户主目录

**3.创建目录命令**

pwd:显示当前工作目录

mkdir: 创建目录

mkdir   a :当前目录下创建文件夹a

mkdir  /root/b  :在根目录下的root下创建目录b

mkdir  -pv  /root/c/e/d :在根目录下的root下创建目录结构c/e/d（一下创建多级目录）

-p：父目录不存在情况下先生成父目录

-v：显示命令执行过程中的详细信息

**4.文件浏览**

[注]：按tab键可以自动补齐文件名

**cat   文件名**：显示指定文件的所有内容（用于文件内容少的情况）

**more  文件名**：分页显示文件内容，还支持直接跳转行等功能（用于文件内容多的情况）

Enter ：向下n行，需要定义。默认为1行

空格键：向下滚动一屏  或 Ctrl+F

B    返回上一屏  或Ctrl+B

q   退出more

**less [参数]  文件名** ：分页显示文件内容，操作更详细

参数：-m 显示类似more命令的百分比        -N   显示每行的行号     两个参数可以结合用  -mN

|  |  |  |
| --- | --- | --- |
| 空格键：前进一页 或 page down | b：后退一页或 page up | d：前进半页 |
| y：后退一行或方向键向上 | /字符串：向下搜索 | ？字符串：向上搜索 |
| Enter：前进一行 或 方向键向下 | u：后退半页 | v：进入vim编辑 |

**tall命令**：用来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。

**tail[必要参数][选择参数][文件]**

 -n<行数> 显示行数

-f 循环读取

**5.文件操作**

**1.复制命令：cp**

cp(copy)功能：复制文件或目录

**语法：cp[参数]   [源文件或目录]    [目标文件或目录]**

-r或--recursive   递归处理，将指定目录下的文件与子目录一并处理

 -b  覆盖文件时，可以将源文件做一个备份

**复制目录**：(无论是单层目录还是多层目录都可以复制):

**2.移动命令：mv**

**mv（move）功能：移动或更名现有的文件或目录**

**语法：mv [源文件或目录][目标文件或目录**

-f  或 --force   若目标文件或目录与现有的文件目录重复，则直接覆盖现有的文件或目录

剪切文件：mv    ./1.txt    /root    将当前目录下的1.txt文件 剪切到root目录下

重命名文件：mv  ./1.txt   ./2.txt      将当前目录下的1.txt文件重命名为2.txt

**3.删除命令：rm**

**rm功能：删除文件或目录**

**语法：rm [-dfirv][--help][--version][文件或目录...]**

-f 或 --force   强制删除文件或目录

-r 或 -R 或 --recursive   递归处理，将指定目录下的所有文件及子目录一并处理

强制删除文件：rm   -f   ./1.txt

强制删除目录：rm   -rf   ./a

**4.查找命令：find**

**find功能：查找文件或目录**

**语法：find[目录...][参数]**

-name  指定字符串作为寻找文件或目录的范本样式

例如：find  /root/  -name  'test\*'   查找root目录下的文件开头是test的文件和文件夹，\*是通配符 。注意字符串要用单引号引起来。

**6.文档编辑**

**vi或vim命令**

基本操作

1.vim 文件名 ：进入一般模式（不能输入）

2.按下 i 从一般模式，进入到插入模式

3.按下esc从插入模式，退出到一般模式

4.在一般模式下，输入:wq,退出编辑。

**7.管道**

linux提供管道符号 “|”  作用是“命令1”的输出内容，将作为“命令2”的输入内容。一般与grep命令一起使用

**格式：命令1 | 命令2**

**1.grep命令**

功能：用于过滤/搜索特定字符。可使用正则表达式 能多种命令配合使用。

格式：**grep [option] pattern [file]**   gre 参数过滤条件文件   （-i 或--ignore-case 忽略字符大小写的差别）

例如：grep -i one /root/1.txt 显示root目录下的1.txt文件中含有one字符的一行数据（忽略大小写）

格式：其他命令 | grep  参数  过滤条件

例如：cat   /root/1.txt  |  grep  -i  one     在cat命令输出结果基础上，进行过滤

**8.系统命令**

**1.ps命令**

任务管理器：ps  -ef        查看当前所有进程（包括执行ps命令的那个时刻进程信息）

**格式：ps [参数]**

-e  此参数的效果和指定“A”参数相同，显示所有程序

-f  显示UID,PPIP,C与STIME栏位

例如：ps -ef  |  grep -i mysql     过滤出mysql这个进程

**2.kill命令**

kill功能：删除执行中的程序或工作

语法：**kill[参数][程序]**

-l<信息编号>       若不加<信息编号>选项，则-l参数会列出全部的信息名称。

**kill  -9  ：表示强制终止**

格式：kill    -9    pid

可先通过ps -ef 查找出所要删除的进程pid，再通过kill -9 pid 终止进程

**3.ifconfig命令**

功能：显示网络设备

格式：ping  主机名或ip地址    （按ctrl+c停止ping）

**9.备份压缩：tar命令**

**tar功能：文件备份压缩**

格式：tar      参数       压缩后的文件名      被压缩的文件名

         tar      参数       被解压的文件名      -C  指定目录            （如果省略 -C 指定目录，则解压到当前文件夹）

-c 建立一个压缩文件的参数指令（create）--压缩

-x 解开一个压缩文件的参数指令（extract）--解压

-z 是否需要用gzip压缩

-v 压缩的过程中显示文件（verbose）

-f 使用档名，在f之后要立即接档名（file）

常用解压参数组合：**zxvf**

常用压缩参数组合：**zcvf**

例如：

  tar  -zcvf  /root/1.tar   /root/a     将root目录下的a文件夹压缩成1.tar,放在root目录下。

  tar  -zxvf  ./1.tar        将当前目录下的1.tar文件解压缩

[注]：如果后缀名为.tar.gz的压缩包用-zxvf   如果后缀名为.tar的压缩包用-xvf

**10.关机  重启**

Linux centos 重启命令：reboot

Linux centos 关机命令：halt

**11.文件权限：chmod命令**

**chmod功能**：变更文件或目录的权限

语法：chmod [参数][<权限范围><符号><权限代号>]

-R  或 --recursive   递归处理，将指定目录下的所有文件及子目录一并处理

--权限范围的表示法如下：

u：User  即文件或目录的拥有者

g：Group 即文件或目录的所属群组

o：Other  除了文件或目录拥有者或所属群组之外，其他用户皆属于这个范围

a：All        即全部的用户，包含拥有者，所属群组及其他用户。

--符号

+ 添加权限

- 取消权限

-- 有关权限代号的部分

r：读取权限，数字代号为“4”

w：写入权限，数字代号为“2”

x：执行或切换权限，数字代号为“1”

- ：不具任何权限，数字代号为“0”

777 ：所有权限

[注]：3=1+2（wx）   5=4+1(rx)

mkdir xxx  创建xxx目录

例如：

chmod    u-rwx   xxx    取消xxx目录的用户“读写执行”权限

chmod    g-rwx   xxx    取消xxx目录的组“读写执行”权限

chmod    777      xxx    给xxx目录添加所有权限

原文链接：https://blog.csdn.net/weixin\_44706512/article/details/90604635

# 第四章 分布式

## 4.1、Nginx

### 4.1.1请解释什么是C10K问题或者知道什么是C10K问题吗？

### 4.1.2 Nginx简介，可参考《Nginx简介》

### 4.1.3 正向代理和反向代理.

### 4.1.4 Nginx几种常见的负载均衡策略

### 4.1.5 Nginx服务器上的Master和Worker进程分别是什么

### 4.1.6 使用“反向代理服务器”的优点是什么?

## 4.2、分布式其他

谈谈业务中使用分布式的场景

Session 分布式方案

Session 分布式处理

分布式锁的应用场景、分布式锁的产生原因、基本概念

分布是锁的常见解决方案

分布式事务的常见解决方案

集群与负载均衡的算法与实现

说说分库与分表设计，可参考《数据库分库分表策略的具体实现方案》

分库与分表带来的分布式困境与应对之策

## 4.3、Dubbo

什么是Dubbo，可参考《Dubbo入门》

什么是RPC、如何实现RPC、RPC 的实现原理，可参考《基于HTTP的RPC实现》

Dubbo中的SPI是什么概念

Dubbo的基本原理、执行流程

### Dubbo和springcloud 区别

|  |  |  |
| --- | --- | --- |
| 功能说明 | Dubbo | SpringCloud |
| 服务注册中心 | Zookeeper | Spring Cloud Netfix Eureka |
| 服务调用方式 | RPC | REST API |
| 服务监控 | Dubbo-monitor | Spring Boot Admin |
| 熔断器 | 不完善 | Spring Cloud Netflix Hystrix |
| 服务网关 | 无 | Spring Cloud Netflix Zuul |
| 分布式配置 | 无 | Spring Cloud Config |
| 服务跟踪 | 无 | Spring Cloud Sleuth |
| 数据流 | 无 | Spring Cloud Stream |
| 批量任务 | 无 | Spring Cloud Task |
| 信息总线 | 无 | Spring Cloud Bus |

**最大的区别：Dubbo**底层是使用**Netty这样的NIO框架**，是**基于TCP协议传输**的，配合以Hession序列化完成RPC通信。而**SpringCloud**是基**于Http协议+rest接口调用远程过程**的通信，相对来说，Http请求会有更大的报文，占的带宽也会更多。但是REST相比RPC更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下，显得更为合适，至于注重通信速度还是方便灵活性，具体情况具体考虑。

**背景区别：**Dubbo是来源于阿里团队，SpringCloud是来源于Spring团队，Spring广泛遍布全球各种企业开发中，可以确保SpringCloud的后续更新维护，Dubbo虽然来自国内顶尖的阿里团队,但是曾经被阿里弃用停更，但是后来阿里又低调重启维护。

**定位区别：**

Dubbo 是 SOA 时代的产物，它的关注点主要在于服务的调用，流量分发、流量监控和熔断。Spring Cloud 诞生于微服务架构时代，考虑的是微服务治理的方方面面，另外由于依托了 Spirng、Spirng Boot 的优势之上，两个框架在开始目标就不一致，Dubbo 定位服务治理、Spirng Cloud 是一个生态。因此可以大胆地判断，Dubbo 未来会在服务治理方面更为出色，而 Spring Cloud 在微服务治理上面无人能敌。

**模块区别：**

1. Dubbo主要分为服务注册中心，服务提供者，服务消费者，还有管控中心；
2. 相比起Dubbo简单的四个模块，SpringCloud则是一个完整的分布式一站式框架，他有着一样的服务注册中心，服务提供者，服务消费者，管控台，断路器，分布式配置服务，消息总线，以及服务追踪等；

**性能区别：**

Dubbo的每次测试除去网络波动之外，都表现非常稳定

Spring Cloud在第一次最慢，之后越来越快，连续测试4次以上单次测试性能超过dubbo

Spring Cloud-zuul在第一次最慢，之后也表现越来越快，连续4次以上测试 单次性能与dubbo相近，相差在0.02ms内

## 4.4、zookeeper

### 4.4.1 一致性协议:Zab协议

# 第五章 微服务

## 5.1、微服务

前后端分离是如何做的？

微服务哪些框架

Spring Could的常见组件有哪些？可参考《Spring Cloud概述》

领域驱动有了解吗？什么是领域驱动模型？充血模型、贫血模型

JWT有了解吗，什么是JWT，可参考《前后端分离利器之JWT》

你怎么理解 RESTful

说说如何设计一个良好的 API

如何理解 RESTful API 的幂等性

如何保证接口的幂等性

说说 CAP 定理、BASE 理论

怎么考虑数据一致性问题

说说最终一致性的实现方案

微服务的优缺点，可参考《微服务批判》

微服务与 SOA 的区别

如何拆分服务、水平分割、垂直分割

如何应对微服务的链式调用异常

如何快速追踪与定位问题

如何保证微服务的安全、认证

## 5.2、安全问题

如何防范常见的Web攻击、如何方式SQL注入

服务端通信安全攻防

HTTPS原理剖析、降级攻击、HTTP与HTTPS的对比

## 5.3、性能优化

性能指标有哪些

如何发现性能瓶颈

性能调优的常见手段

说说你在项目中如何进行性能调优

# 第六章 其他

## 6.1

## 6.1、设计能力

说说你在项目中使用过的UML图

你如何考虑组件化、服务化、系统拆分

秒杀场景如何设计

可参考：《秒杀系统的技术挑战、应对策略以及架构设计总结一二！》

## 6.2、业务工程

说说你的开发流程、如何进行自动化部署的

你和团队是如何沟通的

你如何进行代码评审

说说你对技术与业务的理解

说说你在项目中遇到感觉最难Bug，是如何解决的

介绍一下工作中的一个你认为最有价值的项目，以及在这个过程中的角色、解决的问题、你觉得你们项目还有哪些不足的地方

## 6.3、软实力

说说你的优缺点、亮点

说说你最近在看什么书、什么博客、在研究什么新技术、再看那些开源项目的源代码

说说你觉得最有意义的技术书籍

工作之余做什么事情、平时是如何学习的，怎样提升自己的能力

说说个人发展方向方面的思考

说说你认为的服务端开发工程师应该具备哪些能力

说说你认为的架构师是什么样的，架构师主要做什么

如何看待加班的问题

当然，一个完整的面试肯定不止上述的知识点，其他的诸如：Linux、数据结构、算法、逻辑思维题、系统设计题、职业规划等等都会有所涉及！总之，充分的准备是应对一切面试的不二法宝！