# java基础

<https://github.com/crossoverJie>

<https://github.com/Snailclimb/JavaGuide/blob/master/Java%E7%9B%B8%E5%85%B3/Java%E5%9F%BA%E7%A1%80%E7%9F%A5%E8%AF%86.md>

## 面向对象和面向过程的区别

### 面向过程

**优点：** 性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源;比如单片机、嵌入式开发、Linux/Unix等一般采用面向过程开发，性能是最重要的因素。

**缺点：** 没有面向对象易维护、易复用、易扩展

### 面向对象

**优点：** 易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护

**缺点：** 性能比面向过程低

## Java 语言有哪些特点

1. 简单易学；
2. 面向对象（封装，继承，多态）；
3. 平台无关性（ Java 虚拟机实现平台无关性）；
4. 可靠性；
5. 安全性；
6. 支持多线程（ C++ 语言没有内置的多线程机制，因此必须调用操作系统的多线程功能来进行多线程程序设计，而 Java 语言却提供了多线程支持）；
7. 支持网络编程并且很方便（ Java 语言诞生本身就是为简化网络编程设计的，因此 Java 语言不仅支持网络编程而且很方便）；
8. 编译与解释并存；

## 多态

在Java中有两种形式可以实现多态：继承（多个子类对同一方法的重写）和接口（实现接口并覆盖接口中同一方法）。

## 重载和重写的区别

**重载：** 发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同，发生在编译时。

**重写：** 发生在父子类中，方法名、参数列表必须相同，返回值范围小于等于父类，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类；如果父类方法访问修饰符为 private 则子类就不能重写该方法。

## 接口和抽象类的区别是什么

1. 接口的方法默认是 public，所有方法在接口中不能有实现(Java 8 开始接口方法可以有默认实现），抽象类可以有非抽象的方法
2. 接口中的实例变量默认是 final 类型的，而抽象类中则不一定
3. 一个类可以实现多个接口，但最多只能实现一个抽象类
4. 一个类实现接口的话要实现接口的所有方法，而抽象类不一定
5. 接口不能用 new 实例化，但可以声明，但是必须引用一个实现该接口的对象 从设计层面来说，抽象是对类的抽象，是一种模板设计，接口是行为的抽象，是一种行为的规范。

## 成员变量与局部变量的区别

1. 从语法形式上，看成员变量是属于类的，而局部变量是在方法中定义的变量或是方法的参数；成员变量可以被 public,private,static 等修饰符所修饰，而局部变量不能被访问控制修饰符及 static 所修饰；但是，成员变量和局部变量都能被 final 所修饰；
2. 从变量在内存中的存储方式来看，成员变量是对象的一部分，而对象存在于堆内存，局部变量存在于栈内存
3. 从变量在内存中的生存时间上看，成员变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而自动消失。
4. 成员变量如果没有被赋初值，则会自动以类型的默认值而赋值（一种情况例外被 final 修饰的成员变量也必须显示地赋值）；而局部变量则不会自动赋值。

## == 与 equals

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

**equals()** : 它的作用也是判断两个对象是否相等。但它一般有两种使用情况：

* 情况1：类没有覆盖 equals() 方法。则通过 equals() 比较该类的两个对象时，等价于通过“==”比较这两个对象。
* 情况2：类覆盖了 equals() 方法。一般，我们都覆盖 equals() 方法来两个对象的内容相等；若它们的内容相等，则返回 true (即，认为这两个对象相等)。

**说明：**

* String 中的 equals 方法是被重写过的，因为 object 的 equals 方法是比较的对象的内存地址，而 String 的 equals 方法比较的是对象的值。
* 当创建 String 类型的对象时，虚拟机会在常量池中查找有没有已经存在的值和要创建的值相同的对象，如果有就把它赋给当前引用。如果没有就在常量池中重新创建一个 String 对象。

## Java 中的异常处理

**Error（错误）:是程序无法处理的错误**，表示运行应用程序中较严重问题。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。例如，Java虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止。

这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时，如Java虚拟机运行错误（Virtual MachineError）、类定义错误（NoClassDefFoundError）等。这些错误是不可查的，因为它们在应用程序的控制和处理能力之 外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。在 Java中，错误通过Error的子类描述。

**Exception（异常）:是程序本身可以处理的异常**。Exception 类有一个重要的子类 **RuntimeException**。RuntimeException 异常由Java虚拟机抛出。**NullPointerException**（要访问的变量没有引用任何对象时，抛出该异常）、**ArithmeticException**（算术运算异常，一个整数除以0时，抛出该异常）和 **ArrayIndexOutOfBoundsException** （下标越界异常）。

## 非序列化的变量transient

对于不想进行序列化的变量，使用transient关键字修饰。

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

## 获取用键盘输入常用的的两种方法

方法1：通过 Scanner

Scanner input = new Scanner(System.in);String s = input.nextLine();

input.close();

方法2：通过 BufferedReader

BufferedReader input = new BufferedReader(new InputStreamReader(System.in)); String s = input.readLine();

## static 关键字

* **static 关键字主要有以下四种使用场景：**this、super不能用在static方法中。

1. **修饰成员变量和成员方法:** 被 static 修饰的成员属于类，不属于单个这个类的某个对象，被类中所有对象共享，可以并且建议通过类名调用。被static 声明的成员变量属于静态成员变量，静态变量 存放在 Java 内存区域的方法区。调用格式：类名.静态变量名 类名.静态方法名()
2. **静态代码块:** 静态代码块定义在类中方法外, 静态代码块在非静态代码块之前执行(静态代码块—>非静态代码块—>构造方法)。 该类不管创建多少对象，静态代码块只执行一次.
3. **静态内部类（static修饰类的话只能修饰内部类）：** 静态内部类与非静态内部类之间存在一个最大的区别: 非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围类，但是静态内部类却没有。没有这个引用就意味着：1. 它的创建是不需要依赖外围类的创建。2. 它不能使用任何外围类的非static成员变量和方法。
4. **静态导包(用来导入类中的静态资源，1.5之后的新特性):** 格式为：import static 这两个关键字连用可以指定导入某个类中的指定静态资源，并且不需要使用类名调用类中静态成员，可以直接使用类中静态成员变量和成员方法。

## hashCode 与 equals

### hashCode（）介绍

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个int整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。hashCode() 定义在JDK的Object.java中，这就意味着Java中的任何类都包含有hashCode() 函数。

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

### hashCode（）与equals（）的相关规定

1. 如果两个对象相等，则hashcode一定也是相同的
2. 两个对象相等,对两个对象分别调用equals方法都返回true
3. 两个对象有相同的hashcode值，它们也不一定是相等的
4. **因此，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖**
5. hashCode() 的默认行为是对堆上的对象产生独特值。如果没有重写 hashCode()，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）

## 常用类及方法

### String 和 StringBuffer、StringBuilder 的区别是什么 String 为什么是不可变的

**可变性**

简单的来说：String 类中使用 final 关键字字符数组保存字符串，private　final　char　value[]，所以 String 对象是不可变的。而StringBuilder 与 StringBuffer 都继承自 AbstractStringBuilder 类，在 AbstractStringBuilder 中也是使用字符数组保存字符串char[]value 但是没有用 final 关键字修饰，所以这两种对象都是可变的。

StringBuilder 与 StringBuffer 的构造方法都是调用父类构造方法也就是 AbstractStringBuilder 实现的，大家可以自行查阅源码。

AbstractStringBuilder.java

abstract class AbstractStringBuilder implements Appendable, CharSequence {

char[] value;

int count;

AbstractStringBuilder() {

}

AbstractStringBuilder(int capacity) {

value = new char[capacity];

}

**线程安全性**

String 中的对象是不可变的，也就可以理解为常量，线程安全。AbstractStringBuilder 是 StringBuilder 与 StringBuffer 的公共父类，定义了一些字符串的基本操作，如 expandCapacity、append、insert、indexOf 等公共方法。StringBuffer 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder 并没有对方法进行加同步锁，所以是非线程安全的。

**性能**

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

**对于三者使用的总结：**

1. 操作少量的数据 = String
2. 单线程操作字符串缓冲区下操作大量数据 = StringBuilder
3. 多线程操作字符串缓冲区下操作大量数据 = StringBuffer

### Comparable和Comparator区别比较

　　Comparable是排序接口，若一个类实现了Comparable接口，就意味着“该类支持排序”。而Comparator是比较器，我们若需要控制某个类的次序，可以建立一个“该类的比较器”来进行排序。

　　Comparable相当于“内部比较器”，而Comparator相当于“外部比较器”。

　　两种方法各有优劣， 用Comparable 简单， 只要实现Comparable 接口的对象直接就成为一个可以比较的对象，但是需要修改源代码。 用Comparator 的好处是不需要修改源代码， 而是另外实现一个比较器， 当某个自定义的对象需要作比较的时候，把比较器和对象一起传递过去就可以比大小了， 并且在Comparator 里面用户可以自己实现复杂的可以通用的逻辑，使其可以匹配一些比较简单的对象，那样就可以节省很多重复劳动了。

### hashCode:

会返回每个对象特有的序号（java是依据对象的内存地址计算出的此序号）

如果想要让两个不同的对象视为相等的，就必须覆盖Object继下来的hashCode方法和equals方法，因为Object hashCode方法返回的是该对象的内存地址，所以必须重写hashCode方法，才能保证两个不同的对象具有相同的hashCode，同时也需要两个不同对象比较equals方法会返回true

### final关键字

final可以修饰变量，方法和类，用于表示所修饰的内容一旦赋值之后就不会再被改变，比如String类就是一个final类型的类

**final关键字主要用在三个地方：变量、方法、类。**

**对于一个final变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。**

**当用final修饰一个类时，表明这个类不能被继承。final类中的所有成员方法都会被隐式地指定为final方法。**

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

下面总结了一些使用final关键字的好处

final关键字提高了性能。JVM和Java应用都会缓存final变量。

final变量可以安全的在多线程环境下进行共享，而不需要额外的同步开销。

使用final关键字，JVM会对方法、变量及类进行优化。

final关键字主要用在三个地方：变量、方法、类。

1. 对于一个final变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。
2. 当用final修饰一个类时，表明这个类不能被继承。final类中的所有成员方法都会被隐式地指定为final方法。
3. 使用final方法的原因有两个。第一个原因是把方法锁定，以防任何继承类修改它的含义；第二个原因是效率。在早期的Java实现版本中，会将final方法转为内嵌调用。但是如果方法过于庞大，可能看不到内嵌调用带来的任何性能提升（现在的Java版本已经不需要使用final方法进行这些优化了）。类中所有的private方法都隐式地指定为final。

### java序列化和反序列化

## 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。

序列化

 ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(new File("d:/flyPig.txt")));

 oos.writeObject(flyPig);

 System.out.println("FlyPig 对象序列化成功！");

反序列化

 ObjectInputStream ois = new ObjectInputStream(new FileInputStream(new File("d:/flyPig.txt")));

 FlyPig person = (FlyPig) ois.readObject();

 System.out.println("FlyPig 对象反序列化成功！");

## 集合容器

### 数组

默认排序：需要元素实现Comparable接口，重写compareTo方法

### List主要分为3类

ArrayList， LinkedList和Vector（父类Collection）

List是一个有序的集合，和set不同的是，List允许存储项的值为空，也允许存储相等值的存储项

### ArrayList

是一个数组实现的列表，由于数据是存入数组中的，所以它的特点也和数组一样，查询很快，但是中间部分的插入和删除很慢

### Vector

Vector就是ArrayList的线程安全版，它的方法前都加了synchronized锁，其他实现逻辑都相同。   
如果对线程安全要求不高的话，可以选择ArrayList，毕竟synchronized也很耗性能

### ****LinkedList****

LinkedList是一个双向链表,JDK1.6之前为循环链表，JDK1.7取消了循环。注意双向链表和双向循环链表的区别：

链表的数据结构：一个是当前链表的头节点，一个是尾部节点，还有是链表长度。

由于查询时候需要移动指针，所以性能偏低。但在插入删除方面只需要移动指针操作优于

ArrayList。因为ArrayList需要进行循环遍历，特别是插入到某一个位置时候需要将当前位置

以后的数据循环遍历下标+1，并且存放到新的数组里面。完成插入的操作。

### ArrayList和LinkedList的分析

可以理解List的3个特性   
1.是按顺序查找   
2.允许存储项为空   
3.允许多个存储项的值相等   
可以知其然知其所以然

然后对比LinkedList和ArrayList的实现方式不同，可以在不同的场景下使用不同的List   
ArrayList是由数组实现的，方便查找，返回数组下标对应的值即可，适用于多查找的场景   
LinkedList由链表实现，插入和删除方便，适用于多次数据替换的场景

### HashMap源码分析

<https://www.jianshu.com/p/b40fd341711e>

http://www.importnew.com/28263.html

散列表（Hash table，也叫哈希表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

HashMap 是一个散列表，它存储的内容是键值对 (key-value) 映射。

HashMap的键不重复：如果hashCode相同的话， 再去比较 equals()方法，如果也相同的话，JVM就认为数据已经存在了，就不会添加数据！

HashMap 实现原理：jdk1.7版本之前(包括1.7) 数组 + 链表

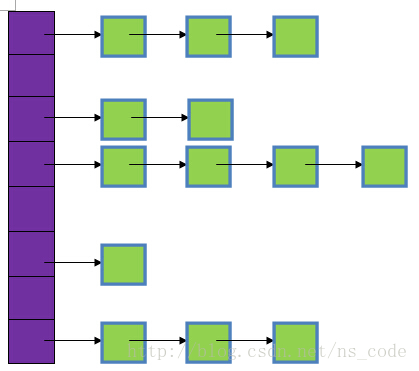
HashMap 里面是一个数组，然后数组中每个元素是一个单向链表。

HashMap的底层主要是基于数组和链表实现的，它之所以有相当快的查询速度主要是因为它是通过计算散列码来决定存储位置的。HashMap中主要是通过key的hashCode来计算hash值，然后通过hash值选择不同的数组来存储。

HashMap 采用一种所谓的“Hash 算法”来决定每个元素的存储位置

只要hashCode相同，计算出来的hash值就一样，如果存储对象多了，就有可能不同的对象计算出来的hash值是相同的，这就出现了所谓的hash冲突。

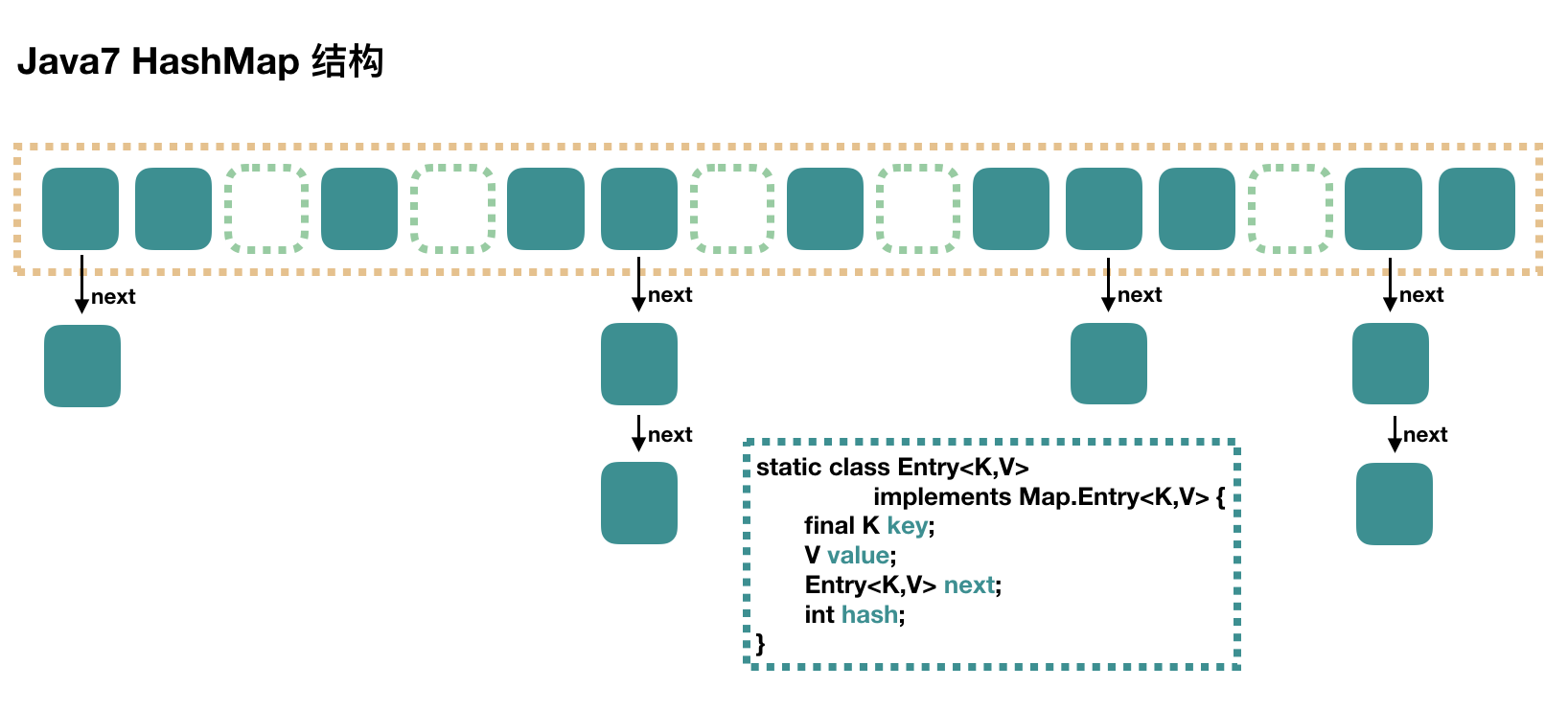
解决hash冲突：HashMap底层是通过链表来解决hash冲突的。



 图中，紫色部分即代表哈希表，也称为哈希数组，数组的每个元素都是一个单链表的头节

点，链表是用来解决冲突的，如果不同的key映射到了数组的同一位置处，就将其放入单链

表中。



这个仅仅是示意图，因为没有考虑到数组要扩容的情况，具体的后面再说。

大方向上，HashMap 里面是一个数组，然后数组中每个元素是一个单向链表。

上图中，每个绿色的实体是嵌套类 Entry 的实例，Entry 包含四个属性：key, value, hash 值和用于单向链表的 next。

capacity：当前数组容量，始终保持 2^n，可以扩容，扩容后数组大小为当前的 2 倍。

loadFactor：负载因子，默认为 0.75。

threshold：扩容的阈值，等于 capacity \* loadFactor

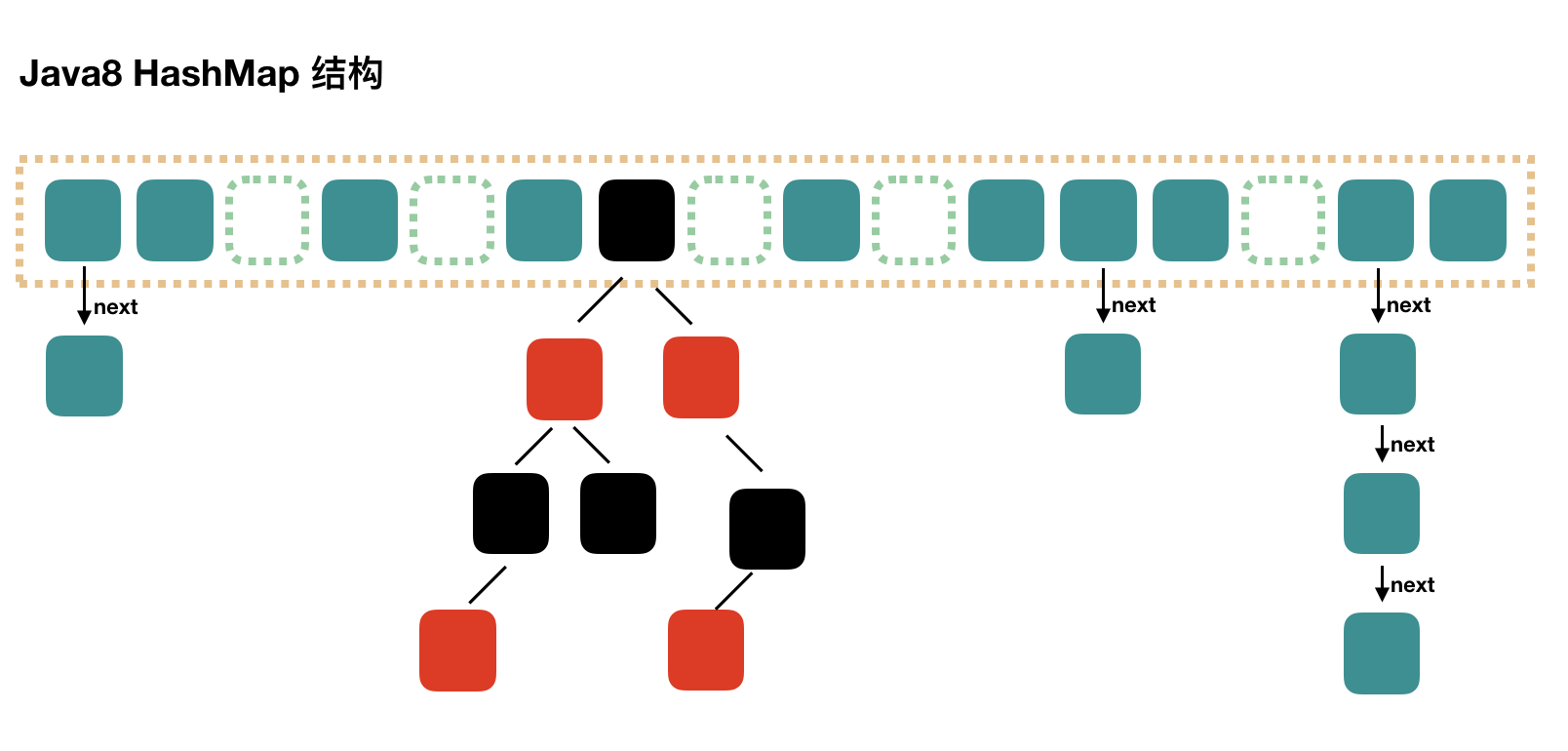
和 Java7 稍微有点不一样的地方就是，Java7 是先扩容后插入新值的，Java8 先插值再扩容，不过这个不重要。

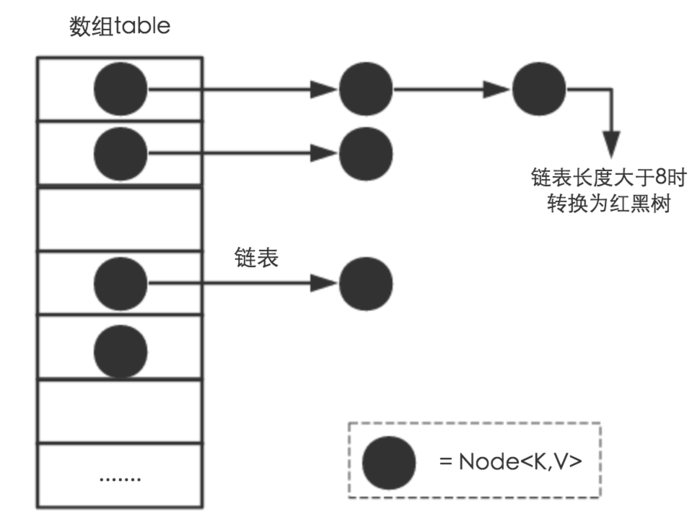
HashMap （JDK1.8 增加了红黑树部分）是数组 + 链表 + 红黑树实现的。

Java8 对 HashMap 进行了一些修改，最大的不同就是利用了红黑树，所以其由 数组+链表+红黑树 组成。

根据 Java7 HashMap 的介绍，我们知道，查找的时候，根据 hash 值我们能够快速定位到数组的具体下标，但是之后的话，需要顺着链表一个个比较下去才能找到我们需要的，时间复杂度取决于链表的长度，为 O(n)。

为了降低这部分的开销，在 Java8 中，当链表中的元素超过了 8 个以后，会将链表转换为红黑树，在这些位置进行查找的时候可以降低时间复杂度为 O(logN)。





### HashMap 特性

* Hash 相关的数据结构本质上就是键值对
* Hash 中不能存在重复的键
* HashMap 提供非常快速查找时间复杂度
* HashMap 具体实现中，null 可以作为 key 或者 value 存在
* HashMap 不是线程安全
* HashMap 非线程安全，即任一时刻可以有多个线程同时写 HashMap，可能会导致数据的不一致。如果需要满足线程安全，可以用 Collections 的 synchronizedMap 方法使 HashMap 具有线程安全的能力，或者使用 ConcurrentHashMap。
* Collections.synchronizedMap

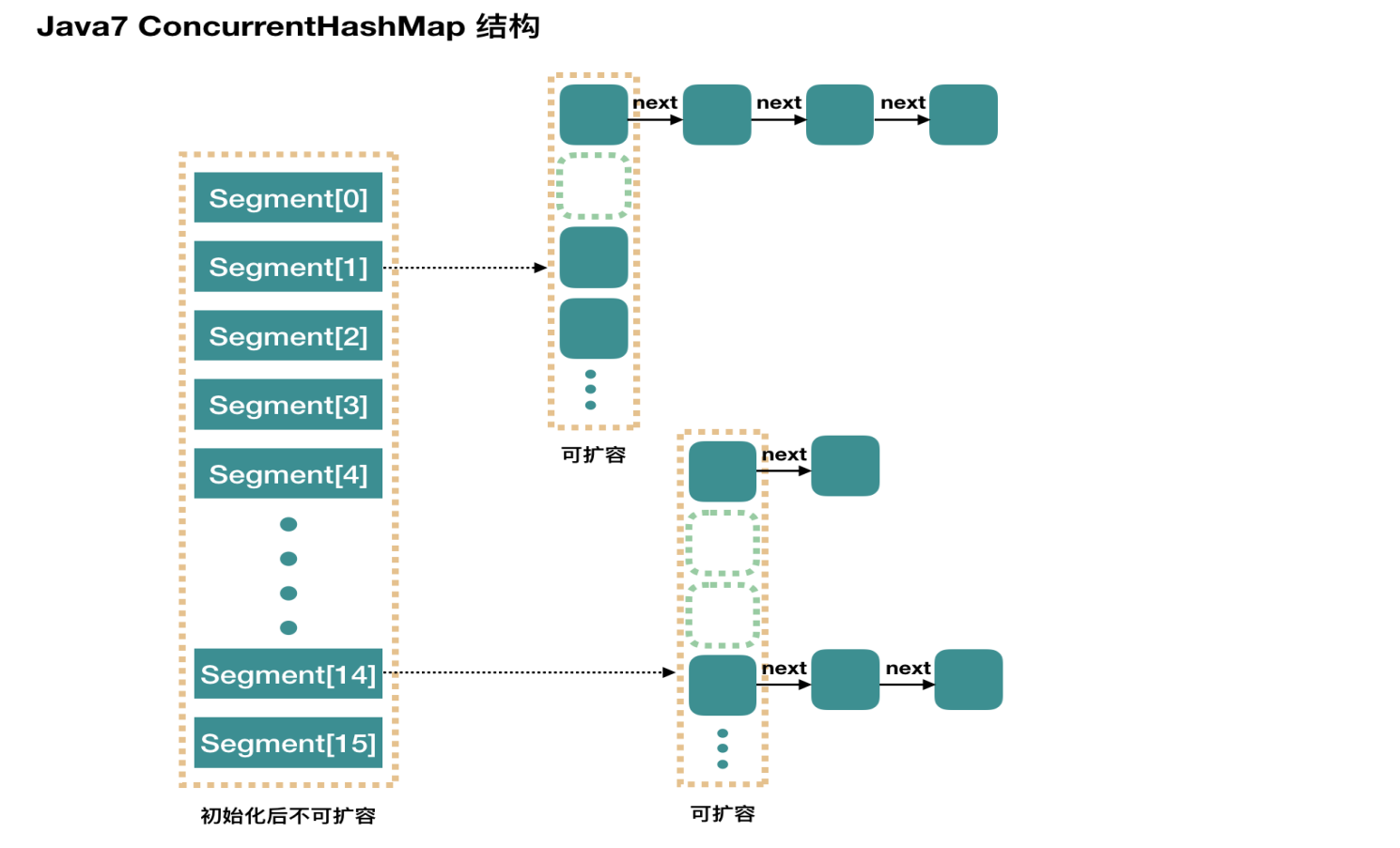
### ConcurrentHashMap

http://www.importnew.com/28263.html

ConcurrentHashMap 和 HashMap 思路是差不多的，但是因为它支持并发操作，所以要复杂一些。

整个 ConcurrentHashMap 由一个个 Segment 组成，Segment 代表”部分“或”一段“的意思，所以很多地方都会将其描述为分段锁。注意，行文中，我很多地方用了“槽”来代表一个 segment。

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



concurrencyLevel：并行级别、并发数、Segment 数，怎么翻译不重要，理解它。默认是 16，也就是说 ConcurrentHashMap 有 16 个 Segments，所以理论上，这个时候，最多可以同时支持 16 个线程并发写，只要它们的操作分别分布在不同的 Segment 上。这个值可以在初始化的时候设置为其他值，但是一旦初始化以后，它是不可以扩容的。

再具体到每个 Segment 内部，其实每个 Segment 很像之前介绍的 HashMap，不过它要保证线程安全，所以处理起来要麻烦些。

JDK1.8

ConcurrentHashMap取消了Segment分段锁，采用CAS和synchronized来保证并发安全。数据结构跟HashMap1.8的结构类似，数组+链表/红黑二叉树。

synchronized只锁定当前链表或红黑二叉树的首节点，这样只要hash不冲突，就不会产生并发，效率又提升N倍。

### TreeMap

### Set 继承自Collection

非线程安全

用于存储无序(存入和取出的顺序不一定相同)元素，值不能重复

TreeMap 是一个**有序的key-value集合**，它是通过[红黑树](http://www.cnblogs.com/skywang12345/p/3245399.html)实现的。

TreeMap基于**红黑树（Red-Black tree）实现**。该映射根据**其键的自然顺序进行排序**，或者根据**创建映射时提供的 Comparator 进行排序**，具体取决于使用的构造方法。  
TreeMap的基本操作 containsKey、get、put 和 remove 的时间复杂度是 log(n) 。O(n)

### HashSet

HashSet 底层就是基于 HashMap 实现的

HashSet实现了Set接口，它不允许集合中有重复的值，当我们提到HashSet时，第一件事情就是在将对象存储在HashSet之前，要先确保对象重写equals()和hashCode()方法，这样才能比较对象的值是否相等，以确保set中没有储存相等的对象。如果我们没有重写这两个方法，将会使用这个方法的默认实现。

public boolean add(Object o)方法用来在Set中添加元素，当元素值重复时则会立即返回false，如果成功添加的话会返回true

hashSet会通过元素的hashcode（）和equals方法进行判断元素师否重复。

HashSet到底是如何判断两个元素重复。

通过hashCode方法和equals方法来保证元素的唯一性，add()返回的是boolean类型

判断两个元素是否相同，先要判断元素的hashCode值是否一致，只有在该值一致的情况下，才会判断equals方法，如果存储在HashSet中的两个对象hashCode方法的值相同equals方法返回的结果是true，那么HashSet认为这两个元素是相同元素，只存储一个（重复元素无法存入）。

### TreeSet

红黑树(自平衡的排序二叉树。

通过compareTo或者compare方法中的来保证元素的唯一性。

添加的元素必须要实现Comparable接口。当compareTo()函数返回值为0时，说明两个对象相等，此时该对象不会添加进来。

### LinkedHashSet

会保存插入的顺序。

### HashTable

HashTable产生于JDK 1.1，而HashMap产生于JDK 1.2。从时间的维度上来看，HashMap要比HashTable出现得晚一些。

HashTable是同步的，HashMap不是，也就是说HashTable在多线程使用的情况下，不需要做额外的同步，而HashMap则不行。

HashMap是支持null键和null值的，而HashTable在遇到null时，会抛出NullPointerException异常。这并不是因为HashTable有什么特殊的实现层面的原因导致不能支持null键和null值，这仅仅是因为HashMap在实现时对null做了特殊处理，将null的hashCode值定为了0，从而将其存放在哈希表的第0个bucket中。

HashMap和HashTable都使用哈希表来存储键值对。在数据结构上是基本相同的，都创建了一个继承自Map.Entry的私有的内部类Entry，每一个Entry对象表示存储在哈希表中的一个键值对。

就是公开的方法比如get都使用了synchronized描述符。而遍历视图比如keySet都使用了Collections.synchronizedXXX进行了同步包装。

HashTable已经被淘汰了，不要在代码中再使用它。

简单来说就是，如果你不需要线程安全，那么使用HashMap，如果需要线程安全，那么使用ConcurrentHashMap。HashTable已经被淘汰了，不要在新的代码中再使用它。

## 多线程

### 同步和异步

同步和异步通常用来形容一次方法调用。同步方法调用一旦开始，调用者必须等到方法调用返回后，才能继续后续的行为。异步方法调用更像一个消息传递，一旦开始，

方法调用就会立即返回，调用者可以继续后续的操作。

关于异步目前比较经典以及常用的实现方式就是消息队列：在不使用消息队列服务器的时候，用户的请求数据直接写入数据库，

在高并发的情况下数据库压力剧增，使得响应速度变慢。但是在使用消息队列之后，用户的请求数据发送给消息队列之后立即 返回，再由消息队列的消费者进程从消息队列中获取数据，

异步写入数据库。由于消息队列服务器处理速度快于数据库（消息队列也比数据库有更好的伸缩性），因此响应速度得到大幅改善。

### 并发(Concurrency)和并行(Parallelism)

并发编程的目的就是为了能提高程序的执行效率提高程序运行速度，但是并发编程并不总是能提高程序运行速度的，而且并发编程可能会遇到很多问题，比如：内存泄漏、上下文切换、死锁还有受限于硬件和软件的资源闲置问题。

### **上下文切换**

当前任务在执行完CPU时间片切换到另一个任务之前会先保存自己的状态，以便下次再切换会这个任务时，可以再加载这个任务的状态。****任务从保存到再加载的过程就是一次上下文切换****。

并发和并行是两个非常容易被混淆的概念。它们都可以表示两个或者多个任务一起执行，但是偏重点有些不同。并发偏重于多个任务交替执行，而多个任务之间有可能还是串行的。而并行是真正意义上的“同时执行”。

多线程在单核CPU的话是顺序执行，也就是交替运行（并发）。多核CPU的话，因为每个CPU有自己的运算器，所以在多个CPU中可以同时运行（并行）。

### 简述线程，程序、进程的基本概念。以及他们之间关系是什么

**线程**与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享同一块内存空间和一组系统资源，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

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

**进程**是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。简单来说，一个进程就是一个执行中的程序，它在计算机中一个指令接着一个指令地执行着，同时，每个进程还占有某些系统资源如CPU时间，内存空间，文件，文件，输入输出设备的使用权等等。换句话说，当程序在执行时，将会被操作系统载入内存中。 线程是进程划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。从另一角度来说，进程属于操作系统的范畴，主要是同一段时间内，可以同时执行一个以上的程序，而线程则是在同一程序内几乎同时执行一个以上的程序段。

### 线程创建方式

 通过继承Thread类，重写run方法；

 通过实现runable接口；

 通过实现callable接口这三种方式

****Runnable接口****不会返回结果但是****Callable接口****可以返回结果。后面介绍****Executors类****的一些方法的时候会介绍到两者的相互转换。

public class CreateThreadDemo {

public static void main(String[] args) {

//1.继承Thread

Thread thread = new Thread() {

@Override

public void run() {

System.out.println("继承Thread");

super.run();

}

};

thread.start();

//2.实现runable接口

Thread thread1 = new Thread(new Runnable() {

@Override

public void run() {

System.out.println("实现runable接口");

}

});

thread1.start();

//3.实现callable接口

ExecutorService service = Executors.newSingleThreadExecutor();

Future<String> future = service.submit(new Callable() {

@Override

public String call() throws Exception {

return "通过实现Callable接口";

}

});

try {

String result = future.get();

System.out.println(result);

} catch (InterruptedException e) {

e.printStackTrace();

} catch (ExecutionException e) {

e.printStackTrace();

}

}

}

### 线程关闭

Interrupt 、this.isInterrupted()、return;

### 多线程分类

用户线程：运行在前台，执行具体的任务，如程序的主线程、连接网络的子线程等都是用户线程

守护线程：运行在后台，为其他前台线程服务.也可以说守护线程是JVM中非守护线程的 “佣人”。

可以通过调用Thead类的setDaemon(true)方法设置当前的线程为守护线程

特点：一旦所有用户线程都结束运行，守护线程会随JVM一起结束工作

应用：数据库连接池中的检测线程，JVM虚拟机启动后的检测线程

最常见的守护线程：垃圾回收线程

### 线程的状态

新建、运行、阻塞、等待、带超时的等待、终止

### **避免死锁**

在线程中，如果两个线程同时等待对方释放锁，相互竞争资源也会产生死锁。

* 避免一个线程同时获得多个锁
* 避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源
* 尝试使用定时锁，使用lock.tryLock(timeout)来替代使用内部锁机制
* 对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败的情况

### 彻底理解synchronized

synchronized可以用在**方法**上也可以使用在**代码块**中

它最大的特征就是在同一时刻只有一个线程能够获得对象的监视器（monitor），从而进入到同步代码块或者同步方法之中，即表现为**互斥性（排它性）**。

synchronized

关键字synchronized可以用来修饰方法和代码块，修饰方法也就相当于将方法放在代码块中。synchronized修饰实例方法时（对象锁），相当于synchronized (this) {}。修饰静态方法时（类锁，也叫静态锁），相当于synchronized (ClassName.class) {}。

对象有锁计数器，一个任务可以多次获得对象的锁，每获取一次，计数加1，每释放一次，计数减1，当计数为0时，其他任务才能获取该对象的锁（可重入性）。

被synchronized修饰的代码块，如果有一个线程获得了对应的锁，其他线程只能一直等待，直到锁被释放。原线程释放锁的情况有：

执行完synchronized修饰的代码块，然后释放锁。

线程执行发生异常，JVM会让线程自动释放锁。

其他时候，如果代码块执行时间很长，或者等待IO、sleep等发生阻塞，一直没有释放锁，其他线程只能一直等待。

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

### **synchronized和 volatile关键字的区别**

synchronized关键字和volatile关键字比较

* ****volatile关键字****是线程同步的****轻量级实现****，所以****volatile性能肯定比synchronized关键字要好****。但是****volatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块****。synchronized关键字在JavaSE1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，****实际开发中使用 synchronized 关键字的场景还是更多一些****。
* ****多线程访问volatile关键字不会发生阻塞，而synchronized关键字可能会发生阻塞****
* ****volatile关键字能保证数据的可见性，但不能保证数据的原子性。synchronized关键字两者都能保证。****
* ****volatile关键字主要用于解决变量在多个线程之间的可见性，而 synchronized关键字解决的是多个线程之间访问资源的同步性。****

### Lock

**可以让尝试获得锁的其他线程只等待一定时间。**

**读操作和写操作、写操作和写操作会发生冲突，但是读操作和读操作不发生冲突，Lock就可以允许这种可以同时进行的操作。（读写锁）**

**Lock可以知道线程有没有成功获取到锁。**

**当竞争资源非常激烈时，Lock的性能会更好。**

**sychronized是不可中断锁，Lock可中断。（可中断锁）**

**Lock可设置为公平锁，按照请求锁的顺序分配锁。（公平锁）**

**可以和条件变量配合使用，对共享数据的多种状态进行监控。**

### JDK1.8并发之synchronized和Lock

2018年06月23日 19:25:36 [mindle1995](https://me.csdn.net/weixin_40255793" \t "https://blog.csdn.net/weixin_40255793/article/details/_blank) 阅读数：843

版权声明：码字不易，欢迎点赞。 https://blog.csdn.net/weixin\_40255793/article/details/80786249

### 什么是线程安全？

线程安全是指保证多线程环境下共享的、可修改的状态的正确性。

保证线程安全的两个办法：

* 封装：将对象的内部状态隐藏、保护起来。
* 不可变：final变量产生了某种程度地不可变（immutable）效果，可以用于保护只读数据。

线程安全需要保证几个基本特性：

* 原子性：相关操作不会中途被其他线程干扰，一般通过同步机制实现。
* 可见性：一个线程修改了某个共享变量，其状态能够立即被其他线程知晓，通常被解释为将线程本地状态反映到主内存上，volatile就是负责保证可见性的。
* 有序性：保证线程内串行语义，避免指令重排。

### Synchronized

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

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

关键字synchronized可以用来修饰方法和代码块，修饰方法也就相当于将方法放在代码块中。synchronized修饰实例方法时（对象锁），相当于synchronized (this) {}。修饰静态方法时（类锁，也叫静态锁），相当于synchronized (ClassName.class) {}。

对象有锁计数器，一个任务可以多次获得对象的锁，每获取一次，计数加1，每释放一次，计数减1，当计数为0时，其他任务才能获取该对象的锁（可重入性）。

### **Synchronized 和 ReenTrantLock 的对比**

****① 两者都是可重入锁****

****② synchronized 依赖于 JVM 而 ReenTrantLock 依赖于 API****

****③ ReenTrantLock 比 synchronized 增加了一些高级功能****

****④ 性能已不是选择标准****

### Volatile 变量

https://blog.csdn.net/qq\_34337272/article/details/79680771

java 内存模型下，线程可以把变量保存**本地内存**（比如机器的寄存器）中，而不是直接在主存中进行读写。这就可能造成一个线程在主存中修改了一个变量的值，而另外一个线程还继续使用它在寄存器中的变量值的拷贝，造成**数据的不一致**。

volatile就可以说是java虚拟机提供的最轻量级的同步机制

线程能够自动发现 volatile 变量的最新值

针对volatile修饰的变量给java虚拟机特殊的约定，线程对volatile变量的修改会立刻被其他线程所感知，即不会出现数据脏读的现象，从而保证数据的“可见性”。

volatile 操作不会像锁一样造成阻塞，因此，在能够安全使用 volatile 的情况下，volatile 可以提供一些优于锁的可伸缩特性。如果读操作的次数要远远超过写操作，与锁相比，volatile 变量通常能够减少同步的性能开销。

synchronized关键字和volatile关键字比较

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

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

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

volatile关键字用于解决变量在多个线程之间的可见性，而ynchronized关键字解决的是多个线程之间访问资源的同步性。

### 什么是等待wait /通知机制nofity

**当线程执行wait()时，会把当前的锁释放，然后让出CPU，进入等待状态。**

**当执行notify/notifyAll方法时，会唤醒一个处于等待该 对象锁 的线程，然后继续往下执行，直到执行完退出对象锁锁住的区域（synchronized修饰的代码块）后再释放锁。**

**notify/notifyAll()执行后，并不立即释放锁**

**notify/notifyAll()执行后，并不立即释放锁，而是要等到执行完临界区中代码后，再释放。**

通俗来讲：

等待/通知机制在我们生活中比比皆是，一个形象的例子就是厨师和服务员之间就存在等待/通知机制。

1. 厨师做完一道菜的时间是不确定的，所以菜到服务员手中的时间是不确定的；

2. 服务员就需要去“等待（wait）”；

3. 厨师把菜做完之后，按一下铃，这里的按铃就是“通知（nofity）”；

4. 服务员听到铃声之后就知道菜做好了，他可以去端菜了。

用专业术语讲：

等待/通知机制，是指一个线程A调用了对象O的wait()方法进入等待状态，而另一个线程B调用了对象O的notify()/notifyAll()方法，线程A收到通知后退出等待队列，进入可运行状态，进而执行后续操作。上诉两个线程通过对象O来完成交互，而对象上的wait()方法和notify()/notifyAll()方法的关系就如同开关信号一样，用来完成等待方和通知方之间的交互工作。

1.3 等待/通知机制的相关方法

方法名称 描述

notify() 随机唤醒等待队列中等待同一共享资源的 “一个线程”，并使该线程退出等待队列，进入可运行状态，也就是notify()方法仅通知“一个线程”

notifyAll() 使所有正在等待队列中等待同一共享资源的 “全部线程” 退出等待队列，进入可运行状态。此时，优先级最高的那个线程最先执行，但也有可能是随机执行，这取决于JVM虚拟机的实现

wait() 使调用该方法的线程释放共享资源锁，然后从运行状态退出，进入等待队列，直到被再次唤醒

wait(long) 超时等待一段时间，这里的参数时间是毫秒，也就是等待长达n毫秒，如果没有通知就超时返回

wait(long，int) 对于超时时间更细力度的控制，可以达到纳秒

---------------------

作者：SnailClimb在CSDN

来源：CSDN

原文：https://blog.csdn.net/qq\_34337272/article/details/79690279

版权声明：本文为博主原创文章，转载请附上博文链接！

①wait() 与 notify/notifyAll 方法必须在同步代码块中使用

②wait() 与  notify/notifyAll() 的执行过程

③中断 调用wait()方法进入等待队列的 线程

④notify 通知的顺序不能错

⑤多线程中测试某个条件的变化用 if 还是用 while？

### join

Thread类中的join方法的主要作用就是同步，它可以使得线程之间的并行执行变为串行执行。调用join方法后需要等待当前线程执行完毕才能继续执行

在A线程中调用了B线程的join()方法时，表示只有当B线程执行完毕时，A线程才能继续执行。

### Lock

Lock的简单使用

Lock lock=new ReentrantLock()；

lock.lock();

try{

}finally{

lock.unlock();

}

因为Lock是接口所以使用时要结合它的实现类，另外在finall语句块中释放锁的目的是保证获取到锁之后，最终能够被释放。

****注意：**** 最好不要把获取锁的过程写在try语句块中，因为如果在获取锁时发生了异常，异常抛出的同时也会导致锁无法被释放。

锁是用于通过多个线程控制对共享资源的访问的工具。通常，锁提供对共享资源的独占访问：一次只能有一个线程可以获取锁，并且对共享资源的所有访问都要求首先获取锁。 但是，一些锁可能允许并发访问共享资源，如ReadWriteLock的读写锁。

在Lock接口出现之前，Java程序是靠synchronized关键字实现锁功能的。JDK1.5之后并发包中新增了Lock接口以及相关实现类来实现锁功能。

虽然synchronized方法和语句的范围机制使得使用监视器锁更容易编程，并且有助于避免涉及锁的许多常见编程错误，但是有时您需要以更灵活的方式处理锁。例如，用于遍历并发访问的数据结构的一些算法需要使用“手动”或“链锁定”：您获取节点A的锁定，然后获取节点B，然后释放A并获取C，然后释放B并获得D等。在这种场景中synchronized关键字就不那么容易实现了，使用Lock接口容易很多。

****Lock接口的实现类：****   
ReentrantLock ， ReentrantReadWriteLock.ReadLock ， ReentrantReadWriteLock.WriteLock

ReentrantLock和synchronized关键字一样可以用来实现线程之间的同步互斥，但是在功能是比synchronized关键字更强大而且更灵

java.util.concurrent.locks.Lock

被synchronized修饰的代码块，如果有一个线程获得了对应的锁，其他线程只能一直等待，直到锁被释放。原线程释放锁的情况有：

* 执行完synchronized修饰的代码块，然后释放锁。
* 线程执行发生异常，JVM会让线程自动释放锁。

其他时候，如果代码块执行时间很长，或者等待IO、sleep等发生阻塞，一直没有释放锁，其他线程只能一直等待。

本文学习完毕之后，你将能够理解Lock的优势包括：

* 可以让尝试获得锁的其他线程只等待一定时间。
* 读操作和写操作、写操作和写操作会发生冲突，但是读操作和读操作不发生冲突，Lock就可以允许这种可以同时进行的操作。（读写锁）
* Lock可以知道线程有没有成功获取到锁。
* 当竞争资源非常激烈时，Lock的性能会更好。
* sychronized是不可中断锁，Lock可中断。（可中断锁）
* Lock可设置为公平锁，按照请求锁的顺序分配锁。（公平锁）
* 可以和条件变量配合使用，对共享数据的多种状态进行监控。

但是

* Lock对象必须被显式地创建、锁定和释放。

private Lock lock = new ReentrantLock();

public void lockedMethod() {

lock.lock();;

try{

*// do something*

return; *// return必须在try子句中，确保unlock不会过早发生*

} finally {

lock.unlock();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

### Lock接口

package java.util.concurrent.locks;public interface Lock {

void lock();

*// 尝试获取锁，如果失败，等待的过程中可以响应中断（threadWait.interrupt()）*

void lockInterruptibly() throws InterruptedException;

*// 尝试获取锁，如果获取成功，就马上返回true，否则（锁已经被其他线程获取）马上返回false*

boolean tryLock();

*// 尝试获取锁，如果获取失败，会等待unit时间，等待期间还拿不到锁就马上返回false*

boolean tryLock(long time, TimeUnit unit) throws InterruptedException;

void unlock();

Condition newCondition();

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

ReentrantLock，重入锁，Lock接口的实现类，提供了更多的方法。

* isFair()，判断锁是否是公平锁
* isLocked()，判断锁是否被任何线程获取了
* isHeldByCurrentThread()，判断锁是否被当前线程获取了
* hasQueuedThreads()，判断是否有线程在等待该锁

### ReadWriteLock接口

public interface ReadWriteLock {

*// 获取读锁*

Lock readLock();

*// 获取写锁*

Lock writeLock();

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7

ReentrantReadWriteLock，重入读写锁，实现了ReadWriteLock接口。

* 如果一个线程已经占用了读锁，其他线程可以马上获得读锁，但需要等待才能获取写锁。
* 如果一个线程已经占用了写锁，其他线程要获取读锁或写锁都需要等待。

private ReadWriteLock rwl = new ReentrantReadWriteLock();

public static void main(String[] args) {

final Main main = new Main();

new Thread(() -> main.testRWL(Thread.currentThread())).start();

new Thread(() -> main.testRWL(Thread.currentThread())).start();

*// 输出的结果是两个thread交替输出“正在读”*

}

public void testRWL(Thread thread) {

rwl.readLock().lock();

try {

long finish = System.currentTimeMillis() + 1;

while (System.currentTimeMillis() <= finish) {

System.out.println(thread.getName() + "正在读");

}

System.out.println(thread.getName() + "读结束");

} finally {

rwl.readLock().unlock();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

### ThreadLocal

http://www.importnew.com/22039.html

把ThreadLocal比做一个存放线程的容器，每一个线程都有自己的共享变量，存放自己的私有数据。提供了线程独有的局部变量，可以在整个线程存活的过程中随时取用

ThreadLocalMap使用ThreadLocal的弱引用作为key，如果一个ThreadLocal没有外部强引用来引用它，那么系统 GC 的时候，这个ThreadLocal势必会被回收，这样一来，ThreadLocalMap中就会出现key为null的Entry，就没有办法访问这些key为null的Entry的value，如果当前线程再迟迟不结束的话，这些key为null的Entry的value就会一直存在一条强引用链：Thread Ref -> Thread -> ThreaLocalMap -> Entry -> value永远无法回收，造成内存泄漏。

* 每次使用完ThreadLocal，都调用它的remove()方法，清除数据。

其实，ThreadLocalMap的设计中已经考虑到这种情况，也加上了一些防护措施：在ThreadLocal的get(),set(),remove()的时候都会清除线程ThreadLocalMap里所有key为null的value。

但是这些被动的预防措施并不能保证不会内存泄漏：

### Lock和条件变量

Condition将Object类的wait、notify、notifyAll等操作转化为相应的条件对象操作await、signal、signalAll，将复杂而晦涩的同步操作转变为直观可控的对象行为。看JDK源码注释中的一个例子：

class BoundedBuffer {

final Lock lock = new ReentrantLock();

final Condition notFull = lock.newCondition(); *// 非满条件对象*

final Condition notEmpty = lock.newCondition(); *// 非空条件对象*

final Object[] items = new Object[100];

int putptr, takeptr, count;

public void put(Object x) throws InterruptedException {

lock.lock();

try {

*// 当数组已满，等待非满条件*

while (count == items.length)

notFull.await();

items[putptr] = x;

if (++putptr == items.length) putptr = 0;

++count;

*// 添加元素，就发送一个非空条件信号*

notEmpty.signal();

} finally {

lock.unlock();

}

}

public Object take() throws InterruptedException {

lock.lock();

try {

*// 当数组已空，等待非空条件*

while (count == 0)

notEmpty.await();

Object x = items[takeptr];

if (++takeptr == items.length) takeptr = 0;

--count;

*// 添加元素，就发送一个非满条件信号*

notFull.signal();

return x;

} finally {

lock.unlock();

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41

对象o的wait方法是让执行线程放弃o的锁并阻塞，而Condition对象的await方法放弃的是Lock锁，因此，一个Lock锁可以同时调整多个条件对象的状态，而synchronized锁只能调整锁隶属的对象的状态。

### 锁的名词解释

#### 排他锁

ReentrantLock（排他锁）具有完全互斥排他的效果，即同一时刻只允许一个线程访问，这样做虽然虽然保证了实例变量的线程安全性，但效率非常低下。ReadWriteLock接口的实现类-ReentrantReadWriteLock读写锁就是为了解决这个问题。

#### 重入锁

重入锁的分配机制是：基于线程的分配，而不是基于方法调用的分配。同一个线程获得锁之后，在其他方法中，不需要再次申请就可以获取锁。synchronized是重入锁，它锁定的是实例的this对象或者静态类的类对象，当锁方法A中又调用同类中的锁方法B时，锁计数增加1，而不用重新申请锁。加锁的ReentrantLock是同一个对象时，Lock锁也满足重入性。

#### 可中断锁

线程A如果在等待sychronized的锁，那么其他线程中用A.interrupt()来中断线程A时，A是不会响应的。这种就是不可中断锁。ReentrantLock使用lockInterrupt方法加锁时，此线程是可中断锁。

#### 公平锁

如果同时有多个线程在等待一个锁时，当锁被释放时，等待时间最长的线程（FIFO）会获得该锁。满足这个条件的就是公平锁。非公平锁无法保证按请求顺序来分配锁。synchronized是非公平锁。ReentrantLock和ReentrantReadWriteLock默认是非公平锁，但是在构造时可以设置为公平锁。

public ReentrantLock() { *// 默认非公平锁*

sync = new NonfairSync();

}

public ReentrantLock(boolean fair) { *// true表示公平锁*

sync = fair ? new FairSync() : new NonfairSync();

}

* 1
* 2
* 3
* 4
* 5
* 6

#### 读写锁

private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

****读读共享**** lock.readLock().lock();

****写写互斥****lock.writeLock().lock();

****读写互斥****

****A线程读：****lock.readLock().lock();

****B线程写：****lock.writeLock().lock();

运行两个使用同一个Service对象实例的线程a,b，线程a执行上面的read方法，线程b执行上面的write方法。你会发现同一时间只允许一个线程执行lock()方法后面的代码。记住：只要出现写操作的过程就是互斥的。

读写锁维护了两个锁，一个是读操作相关的锁也成为共享锁，一个是写操作相关的锁 也称为排他锁。通过分离读锁和写锁，其并发性比一般排他锁有了很大提升。

多个读锁之间不互斥，读锁与写锁互斥，写锁与写锁互斥（只要出现写操作的过程就是互斥的。）。在没有线程Thread进行写入操作时，进行读取操作的多个Thread都可以获取读锁，而进行写入操作的Thread只有在获取写锁后才能进行写入操作。即多个Thread可以同时进

使得多个线程之间的读操作不会发生冲突。比如ReadWriteLock接口就是读写锁。

#### 公平锁与非公平锁

lock = new ReentrantLock(true);// true/false

**Lock锁分为：公平锁 和 非公平锁。公平锁表示线程获取锁的顺序是按照线程加锁的顺序来分配的，即先来先得的FIFO先进先出顺序。而非公平锁就是一种获取锁的抢占机制，是随机获取锁的，和公平锁不一样的就是先来的不一定先的到锁，这样可能造成某些线程一直拿不到锁，结果也就是不公平的了。**

**---------------------**

**作者：SnailClimb在CSDN**

**来源：CSDN**

**原文：https://blog.csdn.net/qq\_34337272/article/details/79714196**

**版权声明：本文为博主原创文章，转载请附上博文链接！**

### **悲观锁与乐观锁**

**悲观锁:** 每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁

**乐观锁:** 每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据

可以使用版本号机制和CAS算法实现。提交版本必须大于记录当前版本才能执行更新

若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

### 什么是CAS

期望值-原值-新值

使用锁时，线程获取锁是一种**悲观锁策略**，即假设每一次执行临界区代码都会产生冲突，所以当前线程获取到锁的时候同时也会阻塞其他线程获取该锁。而CAS操作（又称为无锁操作）是一种**乐观锁策略**，它假设所有线程访问共享资源的时候不会出现冲突，既然不会出现冲突自然而然就不会阻塞其他线程的操作。因此，线程就不会出现阻塞停顿的状态。那么，如果出现冲突了怎么办？无锁操作是使用\*\*CAS(compare and swap)\*\*又叫做比较交换来鉴别线程是否出现冲突，出现冲突就重试当前操作直到没有冲突为止。

### 多线程join

多线程Join方法的作用就是把指定的线程加入到当前线程，让主线程等待子线程结束之后才能继续运行，从而完成同步操作。

join方法是先将线程池中的其它线程wait状态，join执行完毕，又调用了notifyAll()唤醒所有线程中的线程。

强行进入使用join方法的线程，其他线程等待该线程完全执行完后才会进来

### sleep

线程按照指定的时间休眠, sleep方法并不会失去锁。

### sleep() VS wait()

两者主要的区别：

1. sleep()方法是Thread的静态方法，而wait是Object实例方法
2. wait()方法必须要在同步方法或者同步块中调用，也就是必须已经获得对象锁。而sleep()方法没有这个限制可以在任何地方种使用。另外，wait()方法会释放占有的对象锁，使得该线程进入等待池中，等待下一次获取资源。而sleep()方法只是会让出CPU并不会释放掉对象锁；
3. sleep()方法在休眠时间达到后如果再次获得CPU时间片就会继续执行，而wait()方法必须等待Object.notift/Object.notifyAll通知后，才会离开等待池，并且再次获得CPU时间片才会继续执行。

### 守护线程

守护线程是一种特殊的线程，就和它的名字一样，它是系统的守护者，在后台默默地守护一些系统服务，比如垃圾回收线程，JIT线程就可以理解守护线程。与之对应的就是用户线程，用户线程就可以认为是系统的工作线程，它会完成整个系统的业务操作。用户线程完全结束后就意味着整个系统的业务任务全部结束了，因此系统就没有对象需要守护的了，守护线程自然而然就会退。当一个Java应用，只有守护线程的时候，虚拟机就会自然退出。

### 避免死锁

避免一个线程同时获得多个锁；互相竞争资源，并且不释放。

避免一个线程在锁内部占有多个资源，尽量保证每个锁只占用一个资源；

尝试使用定时锁，使用lock.tryLock(timeOut)，当超时等待时当前线程不会阻塞；

对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败的情况

### 同步VS异步

同步和异步通常用来形容一次方法调用。同步方法调用一开始，调用者必须等待被调用的方法结束后，调用者后面的代码才能执行。而异步调用，指的是，调用者不用管被调用方法是否完成，都会继续执行后面的代码，当被调用的方法完成后会通知调用者。比如，在超时购物，如果一件物品没了，你得等仓库人员跟你调货，直到仓库人员跟你把货物送过来，你才能继续去收银台付款，这就类似同步调用。而异步调用了，就像网购，你在网上付款下单后，什么事就不用管了，该干嘛就干嘛去了，当货物到达后你收到通知去取就好。

### 并发与并行

并发和并行是十分容易混淆的概念。并发指的是多个任务交替进行，而并行则是指真正意义上的“同时进行”。实际上，如果系统内只有一个CPU，而使用多线程时，那么真实系统环境下不能并行，只能通过切换时间片的方式交替进行，而成为并发执行任务。真正的并行也只能出现在拥有多个CPU的系统中。

### 线程池与Executor框架

使用线程池的好处

线程池提供了一种限制和管理资源（包括执行一个任务）。 每个线程池还维护一些基本统计信息，例如已完成任务的数量。

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

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

1.降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

2.提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。

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

**Executor 框架结构(主要由三大部分组成**

1 任务。

执行任务需要实现的Runnable接口或Callable接口。

Runnable接口或Callable接口实现类都可以被ThreadPoolExecutor或ScheduledThreadPoolExecutor执行。

两者的区别：

Runnable接口不会返回结果但是Callable接口可以返回结果。后面介绍Executors类的一些方法的时候会介绍到两者的相互转换。

2 任务的执行

以及继承自Executor 接口的ExecutorService接口。ScheduledThreadPoolExecutor和ThreadPoolExecutor这两个关键类实现了ExecutorService接口。

注意： 通过查看ScheduledThreadPoolExecutor源代码我们发现ScheduledThreadPoolExecutor实际上是继承了ThreadPoolExecutor并实现了ScheduledExecutorService ，而ScheduledExecutorService又实现了ExecutorService，正如我们下面给出的类关系图显示的一样。

3.异步计算的结果

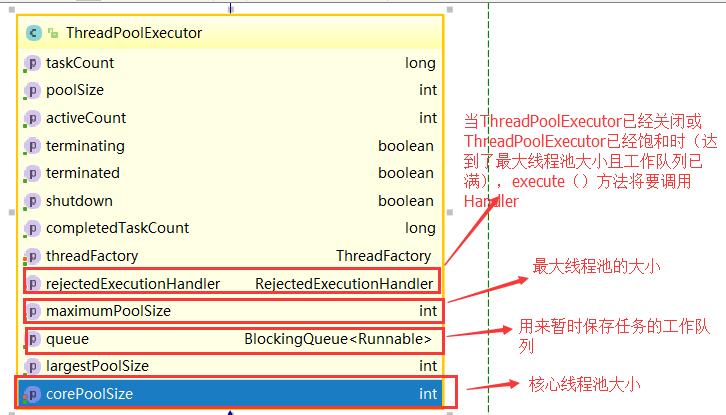
Future接口以及Future接口的实现类FutureTask类。

当我们把Runnable接口或Callable接口的实现类提交（调用submit方法）给ThreadPoolExecutor或ScheduledThreadPoolExecutor时，会返回一个FutureTask对象。

**ThreadPoolExecutor详解**

线程池实现类ThreadPoolExecutor是Executor 框架最核心的类，先来看一下这个类中比较重要的四个属性

**3.1 ThreadPoolExecutor类的四个比较重要的属性**





各种线程池的适用场景介绍

FixedThreadPool： 适用于为了满足资源管理需求，而需要限制当前线程数量的应用场景。它适用于负载比较重的服务器；

SingleThreadExecutor： 适用于需要保证顺序地执行各个任务并且在任意时间点，不会有多个线程是活动的应用场景。

CachedThreadPool： 适用于执行很多的短期异步任务的小程序，或者是负载较轻的服务器；

ScheduledThreadPoolExecutor： 适用于需要多个后台执行周期任务，同时为了满足资源管理需求而需要限制后台线程的数量的应用场景，

SingleThreadScheduledExecutor： 适用于需要单个后台线程执行周期任务，同时保证顺序地执行各个任务的应用场景。

**执行execute()方法和submit()方法的区别是什么呢？**

1)**execute()**方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功与否；****

2)****submit()方法用于提交需要返回值的任务。线程池会返回一个future类型的对象，通过这个future对象可以判断任务是否执行成功****，并且可以通过future的get()方法来获取返回值，get()方法会阻塞当前线程直到任务完成，而使用 get（long timeout，TimeUnit unit）方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

## 常用设计模式

23种常见设计模式

### 工厂模式

我们没有创建逻辑暴露给客户端创建对象，并使用一个通用的接口引用新创建的对象。

### 抽象工厂模式

是一个超级工厂，用来创建其他工厂。 这个工厂也被称为工厂的工厂。 这种类型的设计模式属于创建模式，因为此模式提供了创建对象的最佳方法之一。

在抽象工厂模式中，接口负责创建相关对象的工厂，而不明确指定它们的类。 每个生成的工厂可以按照工厂模式提供对象。

### 单例（Singleton）模式

是Java中最简单的设计模式之一。这种类型的设计模式属于创建模式，因为此模式提供了创建对象的最佳方法之一。

这种模式涉及一个类，它负责创建一个对象，同时确保只创建一个对象。这个类提供了一种方法来访问它的唯一对象，可以直接访问，而不需要实例化类的对象。

### 适配器模式

作为两个不兼容接口之间的桥梁。 这种类型的设计模式属于结构模式，因为该模式组合了两个独立接口。

在这其中InputStreamReader和OutputStreamWriter就在字节流与字符流中起到了桥梁作用， 如：

BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(new File("text.txt"))));

### Java mvc模式

如框架ssh、ssm

* **模型（Model）** - 模型表示携带数据的对象或JAVA POJO。如果其数据改变它也可以具有逻辑来更新控制器。
* **视图（View）** - 视图表示模型包含的数据的可视化层。
* **控制器（Controller）** - 控制器对模型和视图都有起作用。它控制数据流进入模型对象，并在数据更改时更新视图。它保持视图和模型分开，也是视图和模型的中间层。

**Java观察者模式**

activeMQ消息队列

观察者模式在对象之间存在一对多关系时使用，例如，如果一个对象被修改，它的依赖对象将被自动通知。 观察者模式属于行为模式类别。

### 责任链模式：

通过把请求从一个对象传递到链条中下一个对象的方式，直到请求被处理完毕，以实现对象间的解耦。如 javax.servlet.Filter#doFilter()。

### 代理模式（Proxy）

为其他对象提供一种代理以控制对这个对象的访问。

ActiveMQ 是Apache出品，最流行的，能力强劲的开源消息总线。ActiveMQ 是一个完全支持JMS1.1和J2EE 1.4规范的JMS Provider实现,尽管JMS规范出台已经是很久的事情了,但是JMS在当今的J2EE应用中间仍然扮演着特殊的地位。

### 观察者模式

<https://www.cnblogs.com/luohanguo/p/7825656.html>

观察者模式的定义：

在对象之间定义了一对多的依赖，这样一来，当一个对象改变状态，依赖它的对象会收到通知并自动更新。

大白话：

　　其实就是发布订阅模式，发布者发布信息，订阅者获取信息，订阅了就能收到信息，没订阅就收不到信息。

### 事件驱动模型简介

事件驱动模型也就是我们常说的观察者，或者发布-订阅模型；理解它的几个关键点：

1. 首先是一种对象间的一对多的关系；最简单的如交通信号灯，信号灯是目标（一方），行人注视着信号灯（多方）；
2. 当目标发送改变（发布），观察者（订阅者）就可以接收到改变；
3. 观察者如何处理（如行人如何走，是快走/慢走/不走，目标不会管的），目标无需干涉；所以就松散耦合了它们之间的关系。

### 门面模式

slf4j是门面模式的典型应用，因此在讲slf4j前，我们先简单回顾一下门面模式，

门面模式，其核心为外部与一个子系统的通信必须通过一个统一的外观对象进行，使得子系统更易于使用。

## 加解密

### 参考资料

该文中的内容来源于 Oracle 的官方文档 [Java SE Tools Reference](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/index.html" \t "_blank) 。Oracle 在 Java 方面的文档是非常完善的。对 Java 8 感兴趣的朋友，可以直接找到这个总入口 [Java SE 8 Documentation](http://docs.oracle.com/javase/8/" \t "_blank) ，想阅读什么就点什么。本博客不定期从 Oracle 官网搬砖。这里介绍的工具是 [keytool](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html" \t "_blank) 。

### 网络安全概论

在 Web 世界里，安全是一个重之又重的课题，甚至是美国政府都禁止某些加密解密算法的出口。 Java 和 Linux 都是 Web 领域的领头羊，各种加解密的算法和管理工具一应俱全，例如 GnuPG 就是一个不错的安全套件。但是在这里，只展示一下 JDK 中的 keytool 工具的使用方法。

先来说一下加解密算法的分类。基本上可以分为三类，它们分别是对称性解密算法、非对称性加密算法和消息摘要算法。每一类算法中又有多个不同的具体算法。对于这些算法，我们不要求完全掌握，但是对于各类算法的特点和用途是一定要了解的，对于一些常见的名字，如 DES、AES、RSA、DSA、MD5、SHA1 等，一定要熟悉，至少要知道它们分别属于哪类算法。

### 对称加密

对称性加密算法使用同一个密钥对信息进行加密和解密，其信息的安全性一部分取决于加密算法和密钥的长度，另一部分取决于密钥在传递过程中是否会被截获或盗取。

常用的对称性加密算法有DES算法、AES算法、3DES算法、TDEA算法、Blowfish算法、RC5算法、IDEA算法。常用的非对称性加密算法有DSA算法、RSA算法、Elgamal算法、背包算法、Rabin算法、D-H算法、ECC算法（椭圆曲线加密算法）。

### 非对称加密

非对称性加密算法使用两个密钥分别对信息进行加密和解密，这两个密钥称为私钥/公钥对。使用私钥加密的信息必须使用公钥解密，反之亦然。公钥可以公开发布，私钥由加密方保存，绝对不公开，将私钥被截获或窃取的可能性降到最低，因此非对称性加密算法的安全性比对称性加密算法的安全性更高。既然非对称性加密算法比对称性加密算法安全性更高，那对称性加密算法有什么存在的必要呢？这是因为对称性加密算法的运算速度更快。

现实中，往往将对称性加密算法和非对称性加密算法结合使用，对于要传输的大块数据使用对称性加密算法加密，然后对加密使用的密钥使用非对称性加密算法进行加密，这样既可以获得更高的安全性，又可以获得更高的加解密运算速度。

### (消息摘要)散列型加密

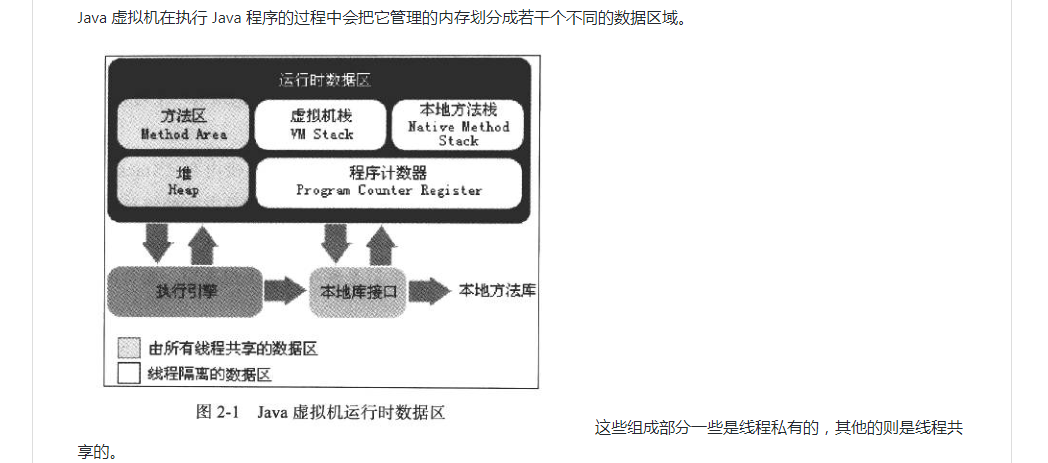
消息摘要算法的主要目的是对数据生成摘要。消息摘要算法不需要密钥，只有输入相同的数据才能得到相同的摘要，而且不可能从摘要反过来推算出数据。常用的消息摘要算法有MD5算法和SHA-1算法及其大量的变体。它们可以用来保证数据的完整性，在网络上发布文件时，常同时提供该文件的MD5值就是利用的消息摘要算法的这个特点，一旦该文件被篡改或者在网络传输中出现数据错误，再对其进行摘要运算就得不到相同的MD5值。

# jvm虚拟机

## Java 主流的虚拟机

JRockit VM和HotSpot

## Java内存模型：



****线程私有的：****

* 程序计数器
* 虚拟机栈
* 本地方法栈

****线程共享的：****

* 堆
* 方法区
* 直接内存

程序计数器：是一块较小的内存空间，是当前线程所执行字节码的行号的指示器，通过改变这个计数器的值来选取需要执行的字节码指令。比如多线程的数据状态，且该区域为线程私有。Java多线程通过空间切换时间，分配处理器时间。为了线程切换后能回复到正确的位置，每条线程都需要独立的程序计数器。

java虚拟机栈：

本地方法 栈：

java虚拟机堆：

常量池：

## jvm 重排序：

## Jvm参数：

**-vmargs -Xms128M -Xmx512M -XX:PermSize=64M -XX:MaxPermSize=128M  
-vmargs 说明后面是VM的参数，所以后面的其实都是JVM的参数了  
-Xms128m JVM初始分配的堆内存  
-Xmx512m JVM最大允许分配的堆内存，按需分配  
-XX:PermSize=64M JVM初始分配的非堆内存  
-XX:MaxPermSize=128M JVM最大允许分配的非堆内存，按需分配**

# 计算机网络模型

## http

## https

## tcp

## udp

## socket

## cookie

cookie的内容主要包括：名字，值，过期时间，路径和域。路径与域一起构成cookie的作用范围。若不设置过期时间，则表示这

个cookie的生命期为浏览器会话期间，关闭浏览器窗口，cookie就消失。这种生命期为浏览器会话期的cookie被称为会话cookie。

会话cookie一般不存储在硬盘上而是保存在内存里，当然这种行为并不是规范规定的。若设置了过期时间，浏览器就会把cookie

保存到硬盘上，关闭后再次打开浏览器，这些cookie仍然有效直到超过设定的过期时间。存储在硬盘上的cookie可以在不同的浏

览器进程间共享，比如两个IE窗口。而对于保存在内存里的cookie，不同的浏览器有不同的处理方式

## session

session机制。session机制是一种服务器端的机制，服务器使用一种类似于散列表的结构（也可能就是使用散列表）来保存信息。

          当程序需要为某个客户端的请求创建一个session时，服务器首先检查这个客户端的请求里是否已包含了一个session标识

（称为session id），如果已包含则说明以前已经为此客户端创建过session，服务器就按照session id把这个session检索出来

使用（检索不到，会新建一个），如果客户端请求不包含session id，则为此客户端创建一个session并且生成一个与此session相

关联的session id，session id的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串，这个session id将被在本次响应

中返回给客户端保存。保存这个session id的方式可以采用cookie，这样在交互过程中浏览器可以自动的按照规则把这个标识发送给

服务器。一般这个cookie的名字都是类似于SEEESIONID。但cookie可以被人为的禁止，则必须有其他机制以便在cookie被禁止时

仍然能够把session id传递回服务器。

## cookie和session的区别

cookie机制采用的是在客户端保持状态的方案，而session机制采用的是在服务器端保持状态的方案。

1、cookie数据存放在客户的浏览器上，session数据放在服务器上。

2、cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗  
   考虑到安全应当使用session。

3、session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能  
   考虑到减轻服务器性能方面，应当使用COOKIE。

4、单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。

5、所以个人建议：  
   将登陆信息等重要信息存放为SESSION  
   其他信息如果需要保留，可以放在COOKIE中

## get和post请求的区别

①get请求用来从服务器上获得资源，而post是用来向服务器提交数据；

②get将表单中数据按照name=value的形式，添加到action 所指向的URL 后面，并且两者使用"?"连接，而各个变量之间使用"&"连接；post是将表单中的数据放在HTTP协议的请求头或消息体中，传递到action所指向URL；

③get传输的数据要受到URL长度限制（1024字节即256个字符）；而post可以传输大量的数据，上传文件通常要使用post方式；

④使用get时参数会显示在地址栏上，如果这些数据不是敏感数据，那么可以使用get；对于敏感数据还是应用使用post；

⑤get使用MIME类型application/x-www-form-urlencoded的URL编码（也叫百分号编码）文本的格式传递参数，保证被传送的参数由遵循规范的文本组成，例如一个空格的编码是"%20"。

补充：GET方式提交表单的典型应用是搜索引擎。GET方式就是被设计为查询用的。

## 转发(Forward)和重定向(Redirect)的区别

**转发是服务器行为，重定向是客户端行为。**

## BIO与NIO、AIO的区别

**https://www.jianshu.com/p/ef418ccf2f7d**

一、BIO

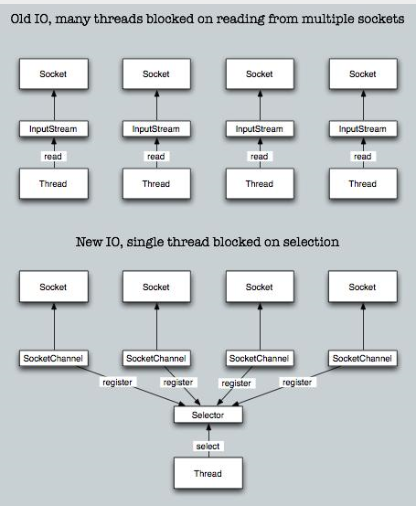
     在JDK1.4出来之前，我们建立网络连接的时候采用BIO模式，需要先在服务端启动一个ServerSocket，然后在客户端启动Socket来对服务端进行通信，默认情况下服务端需要对每个请求建立一堆线程等待请求，而客户端发送请求后，先咨询服务端是否有线程相应，如果没有则会一直等待或者遭到拒绝请求，如果有的话，客户端会线程会等待请求结束后才继续执行。

二、NIO

    NIO本身是基于事件驱动思想来完成的，其主要想解决的是BIO的大并发问题： 在使用同步I/O的网络应用中，如果要同时处理多个客户端请求，或是在客户端要同时和多个服务器进行通讯，就必须使用多线程来处理。也就是说，将每一个客户端请求分配给一个线程来单独处理。这样做虽然可以达到我们的要求，但同时又会带来另外一个问题。由于每创建一个线程，就要为这个线程分配一定的内存空间（也叫工作存储器），而且操作系统本身也对线程的总数有一定的限制。如果客户端的请求过多，服务端程序可能会因为不堪重负而拒绝客户端的请求，甚至服务器可能会因此而瘫痪。

    NIO基于Reactor，当socket有流可读或可写入socket时，操作系统会相应的通知引用程序进行处理，应用再将流读取到缓冲区或写入操作系统。  也就是说，这个时候，已经不是一个连接就要对应一个处理线程了，而是有效的请求，对应一个线程，当连接没有数据时，是没有工作线程来处理的。

   BIO与NIO一个比较重要的不同，是我们使用BIO的时候往往会引入多线程，每个连接一个单独的线程；而NIO则是使用单线程或者只使用少量的多线程，每个连接共用一个线程。



      NIO的最重要的地方是当一个连接创建后，不需要对应一个线程，这个连接会被注册到多路复用器上面，所以所有的连接只需要一个线程就可以搞定，当这个线程中的多路复用器进行轮询的时候，发现连接上有请求的话，才开启一个线程进行处理，也就是一个请求一个线程模式。

      在NIO的处理方式中，当一个请求来的话，开启线程进行处理，可能会等待后端应用的资源(JDBC连接等)，其实这个线程就被阻塞了，当并发上来的话，还是会有BIO一样的问题。

　　HTTP/1.1出现后，有了Http长连接，这样除了超时和指明特定关闭的http header外，这个链接是一直打开的状态的，这样在NIO处理中可以进一步的进化，在后端资源中可以实现资源池或者队列，当请求来的话，开启的线程把请求和请求数据传送给后端资源池或者队列里面就返回，并且在全局的地方保持住这个现场(哪个连接的哪个请求等)，这样前面的线程还是可以去接受其他的请求，而后端的应用的处理只需要执行队列里面的就可以了，这样请求处理和后端应用是异步的.当后端处理完，到全局地方得到现场，产生响应，这个就实现了异步处理。

三、AIO

     与NIO不同，当进行读写操作时，只须直接调用API的read或write方法即可。这两种方法均为异步的，对于读操作而言，当有流可读取时，操作系统会将可读的流传入read方法的缓冲区，并通知应用程序；对于写操作而言，当操作系统将write方法传递的流写入完毕时，操作系统主动通知应用程序。  即可以理解为，read/write方法都是异步的，完成后会主动调用回调函数。  在JDK1.7中，这部分内容被称作NIO.2，主要在java.nio.channels包下增加了下面四个异步通道：

* AsynchronousSocketChannel
* AsynchronousServerSocketChannel
* AsynchronousFileChannel
* AsynchronousDatagramChannel

其中的read/write方法，会返回一个带回调函数的对象，当执行完读取/写入操作后，直接调用回调函数。

BIO是一个连接一个线程。

NIO是一个请求一个线程。

AIO是一个有效请求一个线程。

Java对BIO、NIO、AIO的支持：

Java BIO ： 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。

Java NIO ： 同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

Java AIO(NIO.2) ： 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理，

BIO、NIO、AIO适用场景分析:

BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。

NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。

AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。

## 跨域

[探讨跨域请求资源的几种方式](https://www.cnblogs.com/dojo-lzz/p/4265637.html)

什么是跨域

JSONP

proxy代理

cors

xdr

　　由于浏览器同源策略，凡是发送请求url的协议、域名、端口三者之间任意一与当前页面地址不同即为跨域。

# javaWeb框架

## JSP

 静态包含 <%@ include file="文件相对 url 地址" %>

include指令是不能带参数，include指令也不支持Servlet路径。

动态包含：<jsp:include page="相对 URL 地址" flush="true" />

## Spring

### IOC控制反转 DI

### AOP面向切面编程

### spring注解@CrossOrigin解决跨域

### 注解

注解本身不做任何事情，只是像xml文件一样起到配置作用。注解代表的是某种业务意义，注解背后处理器的工作原理如上源码实现：首先解析所有属性，判断属性上是否存在指定注解，如果存在则根据搜索规则取得bean，然后利用反射原理注入。如果标注在字段上面，也可以通过字段的反射技术取得注解，根据搜索规则取得bean，然后利用反射技术注入。

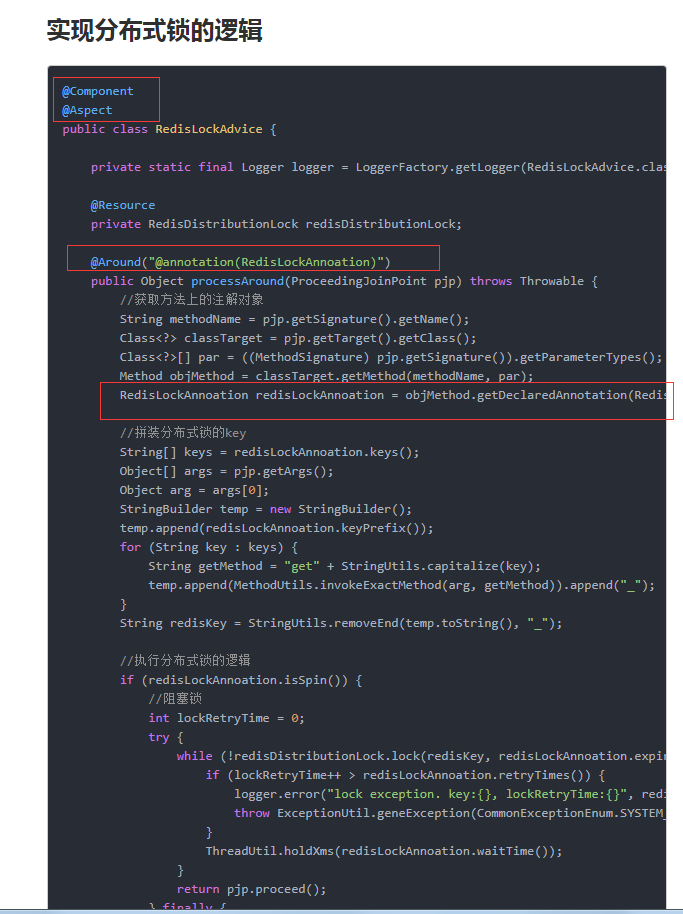
https://www.cnblogs.com/fanheyan/articles/5691659.html

<https://www.jianshu.com/p/e72baf5e5617>

* @Configuration把一个类作为一个IoC容器，它的某个方法头上如果注册了@Bean，就会作为这个Spring容器中的Bean。
* @Scope注解 作用域
* @Lazy(true) 表示延迟初始化
* @Service用于标注业务层组件、
* @Controller用于标注控制层组件（如struts中的action）
* @Repository用于标注数据访问组件，即DAO组件。
* @Component泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。
* @Scope用于指定scope作用域的（用在类上）
* @PostConstruct用于指定初始化方法（用在方法上）
* @PreDestory用于指定销毁方法（用在方法上）
* @Resource 默认按名称装配，当找不到与名称匹配的bean才会按类型装配。
* @DependsOn：定义Bean初始化及销毁时的顺序
* @Primary：自动装配时当出现多个Bean候选者时，被注解为@Primary的Bean将作为首选者，否则将抛出异常
* @Autowired 默认按类型装配，如果我们想使用按名称装配，可以结合@Qualifier注解一起使用
* @Autowired @Qualifier("personDaoBean") 存在多个实例配合使用



注解的实现类



## SpringMvc

## SpringBoot

## MyBatis

# 数据库

## mysql的常用引擎

在MySQL数据库中，常用的引擎主要就是2个：Innodb和MyIASM。

首先：

1.简单介绍这两种引擎，以及该如何去选择。

2.这两种引擎所使用的数据结构是什么。

a.Innodb引擎，Innodb引擎提供了对数据库ACID事务的支持。并且还提供了行级锁和外键的约束。它的设计的目标就是处理大数据容量的数据库系统。它本身实际上是基于Mysql后台的完整的系统。Mysql运行的时候，Innodb会在内存中建立缓冲池，用于缓冲数据和索引。但是，该引擎是不支持全文搜索的。同时，启动也比较的慢，它是不会保存表的行数的。当进行Select count(\*) from table指令的时候，需要进行扫描全表。所以当需要使用数据库的事务时，该引擎就是首选。由于锁的粒度小，写操作是不会锁定全表的。所以在并发度较高的场景下使用会提升效率的。

b.MyIASM引擎，它是MySql的默认引擎，但不提供事务的支持，也不支持行级锁和外键。因此当执行Insert插入和Update更新语句时，即执行写操作的时候需要锁定这个表。所以会导致效率会降低。不过和Innodb不同的是，MyIASM引擎是保存了表的行数，于是当进行Select count(\*) from table语句时，可以直接的读取已经保存的值而不需要进行扫描全表。所以，如果表的读操作远远多于写操作时，并且不需要事务的支持的。可以将MyIASM作为数据库引擎的首先。

补充2点：

c.大容量的数据集时趋向于选择Innodb。因为它支持事务处理和故障的恢复。Innodb可以利用数据日志来进行数据的恢复。主键的查询在Innodb也是比较快的。

d.大批量的插入语句时（这里是INSERT语句）在MyIASM引擎中执行的比较的快，但是UPDATE语句在Innodb下执行的会比较的快，尤其是在并发量大的时候。

2.两种引擎所使用的索引的数据结构是什么？

答案:都是B+树!

MyIASM引擎，B+树的数据结构中存储的内容实际上是实际数据的地址值。也就是说它的索引和实际数据是分开的，只不过使用索引指向了实际数据。这种索引的模式被称为非聚集索引。

Innodb引擎的索引的数据结构也是B+树，只不过数据结构中存储的都是实际的数据，这种索引有被称为聚集索引。

三、 关于Mysql数据库默认的存储引擎：

MyISAM和InnoDB是MySQL的两种存储引擎。从MySQL5.5.5以后，InnoDB是默认引擎。

如果是默认安装，那就应该是InnoDB，你可以在my.cnf文件中找到default-storage-engine=INNODB；

InnoDB优点：支持事务，支持外键，并发量较大，适合大量update。

缺点：查询数据相对较快，不适合大量的select。

## 事务

### 3.1、原子性（A）

所谓的原子性就是说，在整个事务中的所有操作，要么全部完成，要么全部不做，没有中间状态。对于事务在执行中发生错误，所有的操作都会被回滚，整个事务就像从没被执行过一样。

### 3.2、一致性（C）

事务的执行必须保证系统的一致性，就拿转账为例，A有500元，B有300元，如果在一个事务里A成功转给B50元，那么不管并发多少，不管发生什么，只要事务执行成功了，那么最后A账户一定是450元，B账户一定是350元。

### 3.3、隔离性（I）

所谓的隔离性就是说，事务与事务之间不会互相影响，一个事务的中间状态不会被其他事务感知。

### 3.4、持久性（D）

所谓的持久性，就是说一单事务完成了，那么事务对数据所做的变更就完全保存在了数据库中，即使发生停电，系统宕机也是如此。

## 数据库性能优化

# 分布式

## 分布式原理

https://juejin.im/entry/59b9e125f265da064703d1da

相对于单体系统，分布式是指将一个业务拆分不同的子业务，分布在不同的服务器上执行，不同服务器之间可以互相通信和协调。

## 集群

* 是指多台服务器集中在一起，实现同一业务（同一个业务代码部署多套到不同机子上），集合起来提供服务的（一般会配上负载均衡）。理论上，分布式的系统一般都是带集群的。

## 负载均衡

* 高效分发请求

Nginx

## 微服务

* 作者有关 Microservice 的原文：<https://martinfowler.com/articles/microservices.html>
* 中文翻译：<http://insights.thoughtworks.cn/microservices-2/>
* 它不是框架，也不是系统，只是一种架构风格。所以我们用的 dubbo [‘dʌbəu]、Spring Cloud 等框架，都是分布式服务框架。只是这种分布式服务框架都是微服务架构必不可少的基础能力，微服务一定是分布式的。分布式服务的概念比较模糊，只是解决了网站的高并发问题，很多细节问题是微服务帮其明确的。微服务更加强调敏捷和健壮，强调服务的粒度，一个服务只需完成一个单一的、独立的功能，多个微服务组合完成相对复杂的业务系统，以满足需求。而且微服务注重借助于各种中间件进行业务解耦和提高性能，以及提高服务的容错性。
* 所以，理论上：微服务的架构都是带：模块拆分、服务拆分、分布式、集群。
* 微服务其实在很早以前就有人/公司在做，只是过去没人去给它做总结和思考它本身，然后 Martin Fowler 就帮我们帮我们做了。
* 单体系统在早期也没啥不好，但是随着 AWS 这种可弹性扩展的云诞生之后，这种单体架构就显得不够时尚了，只有微服务的架构才能合理榨干这些各种类型的云服务。  
  微服务的目的是有效的拆分应用，实现敏捷开发和部署 。

### 引入微服务带来的技术问题

* 既然微服务的目的是有效的拆分应用，实现敏捷开发和部署，那拆分后的新模式肯定会带来新的问题。
  + 客户端怎么访问 N 个服务？
  + N 个服务怎么管理？
  + N 个服务如何通信？
  + 服务挂了怎么处理？
  + 服务变成无状态？
  + 网络开销？

### RPC原理

### dubbo

### springCloud

## 中间件

### 缓存

**redis**

https://blog.csdn.net/u013256816/article/details/51146314

### 搜索引擎

MQ消息队列

### JMS介绍

JMS的全称是Java Message Service，即Java消息服务。用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。

它主要用于在生产者和消费者之间进行消息传递，生产者负责产生消息，而消费者负责接收消息。把它应用到实际的业务需求中的话我们可以在特定的时候利用生产者生成一消息，并进行发送，对应的消费者在接收到对应的消息后去完成对应的业务逻辑。

对于消息的传递有两种类型：

**一种是点对点的**，即一个生产者和一个消费者一一对应；

**另一种是发布/订阅模式**，即一个生产者产生消息并进行发送后，可以由多个消费者进行接收。

## 分布式锁

https://github.com/yujiasun/Distributed-Kit

[http://www.importnew.com/27477.html#comment-662045](http://www.importnew.com/27477.html" \l "comment-662045)

### 前言

分布式锁一般有三种实现方式：1. 数据库乐观锁；2. 基于Redis的分布式锁；3. 基于ZooKeeper的分布式锁。本篇博客将介绍第二种方式，基于Redis实现分布式锁。虽然网上已经有各种介绍Redis分布式锁实现的博客，然而他们的实现却有着各种各样的问题，为了避免误人子弟，本篇博客将详细介绍如何正确地实现Redis分布式锁。

### 可靠性

首先，为了确保分布式锁可用，我们至少要确保锁的实现同时满足以下四个条件：

1. 互斥性。在任意时刻，只有一个客户端能持有锁。
2. 不会发生死锁。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。
3. 具有容错性。只要大部分的Redis节点正常运行，客户端就可以加锁和解锁。
4. 解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。

### 代码实现

### 组件依赖

首先我们要通过Maven引入Jedis开源组件，在pom.xml文件加入下面的代码：

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>redis.clients</groupId>      <artifactId>jedis</artifactId>      <version>2.9.0</version>  </dependency> |

### 加锁代码

**正确姿势**

Talk is cheap, show me the code。先展示代码，再带大家慢慢解释为什么这样实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | public class RedisTool {        private static final String LOCK\_SUCCESS = "OK";      private static final String SET\_IF\_NOT\_EXIST = "NX";      private static final String SET\_WITH\_EXPIRE\_TIME = "PX";        /\*\*       \* 尝试获取分布式锁       \* @param jedis Redis客户端       \* @param lockKey 锁       \* @param requestId 请求标识       \* @param expireTime 超期时间       \* @return 是否获取成功       \*/      public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String requestId, int expireTime) {            String result = jedis.set(lockKey, requestId, SET\_IF\_NOT\_EXIST, SET\_WITH\_EXPIRE\_TIME, expireTime);            if (LOCK\_SUCCESS.equals(result)) {              return true;          }          return false;        }    } |

可以看到，我们加锁就一行代码：jedis.set(String key, String value, String nxxx, String expx, int time)，这个set()方法一共有五个形参：

* 第一个为key，我们使用key来当锁，因为key是唯一的。
* 第二个为value，我们传的是requestId，很多童鞋可能不明白，有key作为锁不就够了吗，为什么还要用到value？原因就是我们在上面讲到可靠性时，分布式锁要满足第四个条件解铃还须系铃人，通过给value赋值为requestId，我们就知道这把锁是哪个请求加的了，在解锁的时候就可以有依据。requestId可以使用UUID.randomUUID().toString()方法生成。
* 第三个为nxxx，这个参数我们填的是NX，意思是SET IF NOT EXIST，即当key不存在时，我们进行set操作；若key已经存在，则不做任何操作；
* 第四个为expx，这个参数我们传的是PX，意思是我们要给这个key加一个过期的设置，具体时间由第五个参数决定。
* 第五个为time，与第四个参数相呼应，代表key的过期时间。

总的来说，执行上面的set()方法就只会导致两种结果：1. 当前没有锁（key不存在），那么就进行加锁操作，并对锁设置个有效期，同时value表示加锁的客户端。2. 已有锁存在，不做任何操作。

心细的童鞋就会发现了，我们的加锁代码满足我们可靠性里描述的三个条件。首先，set()加入了NX参数，可以保证如果已有key存在，则函数不会调用成功，也就是只有一个客户端能持有锁，满足互斥性。其次，由于我们对锁设置了过期时间，即使锁的持有者后续发生崩溃而没有解锁，锁也会因为到了过期时间而自动解锁（即key被删除），不会发生死锁。最后，因为我们将value赋值为requestId，代表加锁的客户端请求标识，那么在客户端在解锁的时候就可以进行校验是否是同一个客户端。由于我们只考虑Redis单机部署的场景，所以容错性我们暂不考虑。

### Redis 分布式锁的正确实现方式（ Java 版 ）

2017/12/02 | 分类： [基础技术](http://www.importnew.com/cat/basic" \o "查看 基础技术 中的全部文章) | [11 条评论](http://www.importnew.com/27477.html" \l "comments" \o "《Redis 分布式锁的正确实现方式（ Java 版 ）》上的评论) | 标签： [redis](http://www.importnew.com/tag/redis), [分布式](http://www.importnew.com/tag/%e5%88%86%e5%b8%83%e5%bc%8f)

分享到：

原文出处： [吴兆锋](http://wudashan.cn/2017/10/23/Redis-Distributed-Lock-Implement/" \t "_blank)

### 前言

分布式锁一般有三种实现方式：1. 数据库乐观锁；2. 基于Redis的分布式锁；3. 基于ZooKeeper的分布式锁。本篇博客将介绍第二种方式，基于Redis实现分布式锁。虽然网上已经有各种介绍Redis分布式锁实现的博客，然而他们的实现却有着各种各样的问题，为了避免误人子弟，本篇博客将详细介绍如何正确地实现Redis分布式锁。

### 可靠性

首先，为了确保分布式锁可用，我们至少要确保锁的实现同时满足以下四个条件：

1. 互斥性。在任意时刻，只有一个客户端能持有锁。
2. 不会发生死锁。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。
3. 具有容错性。只要大部分的Redis节点正常运行，客户端就可以加锁和解锁。
4. 解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。

### 代码实现

### 组件依赖

首先我们要通过Maven引入Jedis开源组件，在pom.xml文件加入下面的代码：

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>redis.clients</groupId>      <artifactId>jedis</artifactId>      <version>2.9.0</version>  </dependency> |

### 加锁代码

**正确姿势**

Talk is cheap, show me the code。先展示代码，再带大家慢慢解释为什么这样实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | public class RedisTool {        private static final String LOCK\_SUCCESS = "OK";      private static final String SET\_IF\_NOT\_EXIST = "NX";      private static final String SET\_WITH\_EXPIRE\_TIME = "PX";        /\*\*       \* 尝试获取分布式锁       \* @param jedis Redis客户端       \* @param lockKey 锁       \* @param requestId 请求标识       \* @param expireTime 超期时间       \* @return 是否获取成功       \*/      public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String requestId, int expireTime) {            String result = jedis.set(lockKey, requestId, SET\_IF\_NOT\_EXIST, SET\_WITH\_EXPIRE\_TIME, expireTime);            if (LOCK\_SUCCESS.equals(result)) {              return true;          }          return false;        }    } |

可以看到，我们加锁就一行代码：jedis.set(String key, String value, String nxxx, String expx, int time)，这个set()方法一共有五个形参：

* 第一个为key，我们使用key来当锁，因为key是唯一的。
* 第二个为value，我们传的是requestId，很多童鞋可能不明白，有key作为锁不就够了吗，为什么还要用到value？原因就是我们在上面讲到可靠性时，分布式锁要满足第四个条件解铃还须系铃人，通过给value赋值为requestId，我们就知道这把锁是哪个请求加的了，在解锁的时候就可以有依据。requestId可以使用UUID.randomUUID().toString()方法生成。
* 第三个为nxxx，这个参数我们填的是NX，意思是SET IF NOT EXIST，即当key不存在时，我们进行set操作；若key已经存在，则不做任何操作；
* 第四个为expx，这个参数我们传的是PX，意思是我们要给这个key加一个过期的设置，具体时间由第五个参数决定。
* 第五个为time，与第四个参数相呼应，代表key的过期时间。

总的来说，执行上面的set()方法就只会导致两种结果：1. 当前没有锁（key不存在），那么就进行加锁操作，并对锁设置个有效期，同时value表示加锁的客户端。2. 已有锁存在，不做任何操作。

心细的童鞋就会发现了，我们的加锁代码满足我们可靠性里描述的三个条件。首先，set()加入了NX参数，可以保证如果已有key存在，则函数不会调用成功，也就是只有一个客户端能持有锁，满足互斥性。其次，由于我们对锁设置了过期时间，即使锁的持有者后续发生崩溃而没有解锁，锁也会因为到了过期时间而自动解锁（即key被删除），不会发生死锁。最后，因为我们将value赋值为requestId，代表加锁的客户端请求标识，那么在客户端在解锁的时候就可以进行校验是否是同一个客户端。由于我们只考虑Redis单机部署的场景，所以容错性我们暂不考虑。

**错误示例1**

比较常见的错误示例就是使用jedis.setnx()和jedis.expire()组合实现加锁，代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public static void wrongGetLock1(Jedis jedis, String lockKey, String requestId, int expireTime) {        Long result = jedis.setnx(lockKey, requestId);      if (result == 1) {          // 若在这里程序突然崩溃，则无法设置过期时间，将发生死锁          jedis.expire(lockKey, expireTime);      }    } |

setnx()方法作用就是SET IF NOT EXIST，expire()方法就是给锁加一个过期时间。乍一看好像和前面的set()方法结果一样，然而由于这是两条Redis命令，不具有原子性，如果程序在执行完setnx()之后突然崩溃，导致锁没有设置过期时间。那么将会发生死锁。网上之所以有人这样实现，是因为低版本的jedis并不支持多参数的set()方法。

**错误示例2**

这一种错误示例就比较难以发现问题，而且实现也比较复杂。实现思路：使用jedis.setnx()命令实现加锁，其中key是锁，value是锁的过期时间。执行过程：1. 通过setnx()方法尝试加锁，如果当前锁不存在，返回加锁成功。2. 如果锁已经存在则获取锁的过期时间，和当前时间比较，如果锁已经过期，则设置新的过期时间，返回加锁成功。代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public static boolean wrongGetLock2(Jedis jedis, String lockKey, int expireTime) {        long expires = System.currentTimeMillis() + expireTime;      String expiresStr = String.valueOf(expires);        // 如果当前锁不存在，返回加锁成功      if (jedis.setnx(lockKey, expiresStr) == 1) {          return true;      }        // 如果锁存在，获取锁的过期时间      String currentValueStr = jedis.get(lockKey);      if (currentValueStr != null && Long.parseLong(currentValueStr) < System.currentTimeMillis()) {          // 锁已过期，获取上一个锁的过期时间，并设置现在锁的过期时间          String oldValueStr = jedis.getSet(lockKey, expiresStr);          if (oldValueStr != null && oldValueStr.equals(currentValueStr)) {              // 考虑多线程并发的情况，只有一个线程的设置值和当前值相同，它才有权利加锁              return true;          }      }        // 其他情况，一律返回加锁失败      return false;    } |

那么这段代码问题在哪里？1. 由于是客户端自己生成过期时间，所以需要强制要求分布式下每个客户端的时间必须同步。 2. 当锁过期的时候，如果多个客户端同时执行jedis.getSet()方法，那么虽然最终只有一个客户端可以加锁，但是这个客户端的锁的过期时间可能被其他客户端覆盖。3. 锁不具备拥有者标识，即任何客户端都可以解锁。

### 解锁代码

**正确姿势**

还是先展示代码，再带大家慢慢解释为什么这样实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class RedisTool {        private static final Long RELEASE\_SUCCESS = 1L;        /\*\*       \* 释放分布式锁       \* @param jedis Redis客户端       \* @param lockKey 锁       \* @param requestId 请求标识       \* @return 是否释放成功       \*/      public static boolean releaseDistributedLock(Jedis jedis, String lockKey, String requestId) {            String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";          Object result = jedis.eval(script, Collections.singletonList(lockKey), Collections.singletonList(requestId));            if (RELEASE\_SUCCESS.equals(result)) {              return true;          }          return false;        }    } |

可以看到，我们解锁只需要两行代码就搞定了！第一行代码，我们写了一个简单的Lua脚本代码，上一次见到这个编程语言还是在《[黑客与画家](http://www.amazon.cn/gp/product/B004WHZGZQ/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&tag=importnew-23&linkCode=as2&camp=536&creative=3200&creativeASIN=B004WHZGZQ" \t "_blank" \o "黑客与画家:硅谷创业之父Paul Graham文集)》里，没想到这次居然用上了。第二行代码，我们将Lua代码传到jedis.eval()方法里，并使参数KEYS[1]赋值为lockKey，ARGV[1]赋值为requestId。eval()方法是将Lua代码交给Redis服务端执行。

那么这段Lua代码的功能是什么呢？其实很简单，首先获取锁对应的value值，检查是否与requestId相等，如果相等则删除锁（解锁）。那么为什么要使用Lua语言来实现呢？因为要确保上述操作是原子性的。关于非原子性会带来什么问题，可以阅读【[解锁代码-错误示例2](http://wudashan.cn/2017/10/23/Redis-Distributed-Lock-Implement/" \l "releaseLock-wrongDemo2" \t "_blank)】 。那么为什么执行eval()方法可以确保原子性，源于Redis的特性，下面是官网对eval命令的部分解释：

## 单点登录

https://www.cnblogs.com/ywlaker/p/6113927.html

# 性能优化

# 安全攻防

## XSS

## CSRF防御

* 通过 referer、token 或者 验证码 来检测用户提交。
* 尽量不要在页面的链接中暴露用户隐私信息。
* 对于用户修改删除等操作最好都使用post 操作 。
* 避免全站通用的cookie，严格设置cookie的域。

# 需要看的相关书籍

jvm、多线程、redis、mq