## java基础语法

### 方法

方法包括方法头 + 方法体

方法头：又可以分成修饰符、返回值类型、方法名和参数列表

方法名和参数表共同构成**方法签名**。

* 方法名：方法的实际名称。
* 参数列表：定义在方法头中的变量称为形式参数或参数，简称形参。

当调用方法时，需要给参数传递一个值，称为实际参数，简称实参。

参数列表指明方法中的参数类型、次序和数量。

参数是可选的，方法可以不包含参数。

#### 方法重载

方法的重载是指在同一个类中的多个方法有相同的名称，但是方法签名不同。

方法的重载等同于多个方法有相同的名称和不同的参数列表。

方法的重载可以增加程序的可读性，执行相似操作的方法应该有相同的名称。

注意点：

* 被重载的方法必须具有不同的参数列表，而不能通过不同的修饰符和返回值类型进行方法的重载。
* 如果一个方法调用有多个可能的匹配，则编译器会调用最合适的匹配方法，如果编译器无法判断哪个方法最匹配，则称为**歧义调用**，会导致编译错误

#### 方法重写

子类从父类中继承方法。如果子类修改了父类中定义的方法，则称为方法重写。

注意点：

* 要求子类的方法和父类的方法的签名相同
* 如果方法的返回值类型是基本数据类型或者 void，则要求子类的方法的返回值类型和父类的方法的返回值类型相同
* 如果方法的返回值类型是引用类型，则要求返回值类型相同或者子类的方法的返回值类型是父类的方法的返回值类型的子类。
* 实例方法只有当可访问时才能被重写。
* 私有方法不能在定义该方法的类外访问，因此私有方法不能被重写。
* 静态方法可以被继承，但是不能被重写。

重写和重载的区别

* 重载指在同一个类中定义多个方法，这些方法有相同的名称，但是方法签名不同。
* 重写指在子类中定义一个方法，该方法与父类中的方法的签名相同，返回值类型相同或者子类的方法的返回值类型是父类的方法的返回值类型的子类。

### 数据类型

Java 语言的数据类型分为两种：**基本数据类型**和**引用数据类型**。

* 基本数据类型包括 boolean（布尔型）、float（单精度浮点型）、char（字符型）、byte（字节型）、short（短整型）、int（整型）、long（长整型）和 double （双精度浮点型）共 8 种
* 引用数据类型建立在基本数据类型的基础上，包括数组、类和接口。

注意点：

* 引用数据类型是由用户自定义，用来限制其他数据的类型。
* Java 是一种强制类型的语言，所有的变量都必须先明确定义其数据类型，然后才能使用。
* String不属于基本数据类型，而是一种字符数组。
* 拓宽类型：将小范围类型的变量转换为大范围类型，不需要显性声明类型转换
* 缩窄类型：将大范围类型的变量转换为小范围类型，，必须显性声明类型转换，否则会导致编译错误

#### 容器

容器是在数组的基础上衍生出来的，解决数组如下弊端：

1. 长度难以扩充
2. 数据的类型必须相同
3. 数组无法获得有多少个真实的数据，只能获得数组的长度。

在Java当中，有一个类专门用来存放其它类的对象，这个类就叫做容器，它就是将若干性质相同或相近的类对象组合在一起而形成的一个整体。

* Iterator 接口从 JDK 1.2 开始出现，其含义是迭代器，可以用于迭代容器中的元素。

**接口方法：**

* 方法 hasNext 的作用是检测容器中是否还有需要迭代的元素
* 方法next 的作用是返回迭代器指向的元素，并且更新迭代器的状态，将迭代器指向的元素后移一位
* 方法remove 的作用是将迭代器指向的元素删除。该方法有默认实现，默认实现为抛出 UnsupportedOperationException 异常，如果迭代器迭代的容器不支持 remove 操作，则对迭代器调用方法 remove 时会抛出 UnsupportedOperationException 异常
* 方法 forEachRemaining 从 JDK 1.8 开始出现。该方法有默认实现，其作用是对容器中的剩余元素进行处理，直到剩余元素处理完毕或者抛出异常。
* Iterable 接口从 JDK 1.5 开始出现，是 Java 容器的最顶级的接口之一，该接口的作用是使容器具备迭代元素的功能。

**接口方法**：

* 方法 iterator 是 Iterable 接口的核心方法，返回 Iterator 类的迭代器实例。
* 方法 forEach 从 JDK 1.8 开始出现。该方法有默认实现，其作用是对容器中的每个元素进行处理。
* 方法 spliterator 从 JDK 1.8 开始出现。该方法有默认实现，其作用是并行遍历元素

Collection 接口统一定义了单列容器，该接口继承了 Iterable 接口。

**Collection 接口方法：**

* add 一次添加一个元素，addAll 一次将另一个容器中的元素全部添加到当前容器中
* remove 一次删除一个元素，removeAll 一次将另一个容器中的元素全部从当前容器中删除，clear 删除当前容器中的全部元素
* 保留元素的方法有 retainAll，该方法保留既在当前容器中又在另一个容器中的元素，和集合的交集运算相似
* contains 判断当前容器是否包含一个指定元素，containsAll 判断当前容器是否包含另一个容器中的全部元素。
* isEmpty 判断当前容器是否为空（即不包含元素）
* size 返回容器中的元素数目
* toArray 将容器转化成 Object 数组并返回

List 和 Set 是 Collection 接口的子接口。

* List 是线性表，存储一组顺序排列的元素。
* Set 是集合，存储一组互不相同的元素。

 Map，该接口与 Collection 并列存在。不同于 Collection 接口存放单值元素，Map 接口存放的是键值对。

有常用的三种类型的容器，分别是List 、Map、Set。

##### List

List（列表）:是一个有序的列表，元素如果有重复，也会一一列出来

* ArrayList是List一个派生类，非线安全，是基于Object数组实现的可动态扩展的容器，在调用Add的时候会判断当前的长度是否已经超过了Size.
* LinkList与ArrayList区别是LinkList 是基于链表的结构设计 ，插入和删除的性能要高于ArrayList,查询的效率低于LinkList,使用方法基本一致，也是非线安全。
* 比arraylist多了个同步化机制（线程安全），用法相同，但效率比较低，不建议使用

##### Map

键值对，有key和Value两个元素。

* HashMap是**线程不安全，允许多线程去同时访问，允许插入空值**。（JDK1.8之前HashMap由数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突）。JDK1.8以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间）
* Hashtable：**线程安全的，不允许插入空值**， （数组+链表组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的）
* TreeMap： 红黑树，是一个有顺序的HaspMap（自平衡的排序二叉树）

##### Set

Set（集）:与list都是有Collection类的派生出来， 分辨各个元素的标识是HashCode，所以元素不能有重复。

* HashSet（无序，唯一）: 基于 HashMap 实现的，底层采用 HashMap 来保存元素。
* LinkedHashSet： LinkedHashSet 继承与 HashSet，并且其内部是通过 LinkedHashMap 来实现的。有点类似于我们之前说的LinkedHashMap 其内部是基于 Hashmap 实现一样，不过还是有一点点区别的。

## 面向对象

### getClass

Object类中有个getClass方法，该方法返回对象的元对象。

注意点：

* 元对象是一个包含类信息的对象，包括类名、构造方法和方法等。
* 一个类只有一个元对象，同一个类的多个对象对应的元对象相同。
* 每个对象都有一个元对象

### 反射机制

Java 反射机制的核心是**在程序运行时动态加载类以及获取类的信息，从而使用类和对象的数据域和方法**。

Class 类的作用是在程序运行时保存每个对象所属的类的信息，在程序运行时分析类。

一个 Class 类型的对象表示一个特定类的属性。

**获得Class类型的实例：**

1. 对一个对象调用 getClass 方法，获得该对象所属的类的 Class 对象；
2. 调用静态方法 Class.forName，将类名作为参数，获得类名对应的 Class 对象；
3. 对任意的 Java 类型 T（包括基本数据类型、引用类型、数组、关键字 void），调用 T.class 获得类型 T 对应的 Class 对象，此时获得的 Class 对象表示一个类型，但是这个类型不一定是一种类；

**Class类的常用方法：**

1. getName，该方法返回类的名字
2. getFields、getMethods 和 getConstructors 方法分别返回类中所有的公有（即使用可见修饰符 public 修饰）的数据域、方法和构造方法。
3. getDeclaredFields、getDeclaredMethods 和 getDeclaredConstructors 方法分别返回类中所有的数据域、方法和构造方法（包括所有可见修饰符）
4. getField、getMethod 和 getConstructor 方法分别返回类中单个的公有（即使用可见修饰符 public 修饰）的数据域、方法和构造方法
5. getDeclaredField、getDeclaredMethod 和 getDeclaredConstructor 方法分别返回类中单个的数据域、方法和构造方法（包括所有可见修饰符）

### 包装类

基本数据类型包装成对象，对应的类称为包装类。

将基本数据类型的值转换为包装对象，称为装箱。将包装对象转换为基本数据类型的值，称为拆箱。（从 JDK 1.5 开始，基本数据类型和包装类之间可以进行自动转换）



注意点：

* 包装类的构造方法都是有参数的，没有无参数构造方法。
* 调用构造方法时，构造方法的参数值可以是基本数据类型的值，也可以是表示值的字符串。
* 包装类的实例都是不可变的，一旦创建了包装对象，其内部的值就不能再改变。

### 静态和实例

使用关键字 static 修饰的类成员是静态的类成员，从外部调用静态的类成员时，可以通过类名调用，也可以通过对象名调用。

不使用关键字 static 修饰的类成员则是实例的类成员，从外部调用实例的类成员，则只能通过对象名调用。

### 类的继承

从已有的类（父类）派生出新类（子类），称为继承。

如果一个类在定义时没有指定继承，它的父类默认是 Object。

关键字 super 指向当前类的的父类。调用父类的构造方法，使用 super() 或 super(参数)，该语句必须是子类构造方法的第一个语句，且这是调用父类构造方法的唯一方式。调用父类的方法，使用 super.方法名(参数)。

## 异常处理

异常处理主要处理运行错误（引起程序非正常中断或者中止）

Java 提供了 try-catch 块的结构，用于捕获并处理异常，保证程序的稳定性。

在 try 块中包含可能抛出异常的语句，catch 块包含特定异常类型的参数。当 try 块中的一个语句抛出异常且和 catch 块的参数的异常类型一致时，try 块中剩下的语句会被跳过，catch 块中的语句会被执行。

1.声明异常：如果一个方法可能抛出异常，则需要在方法声明中使用关键字 throws 声明异常。如果一个方法可能抛出多种类型的异常，则需要在关键字 throws 之后依次列举可能抛出的异常类型。

2.抛出异常：如果程序检查到错误，则可以创建一个异常的实例并抛出该异常实例。使用关键字 throw 抛出异常。

3.捕获异常：捕获异常通过 try-catch 块实现。如果发现一个 catch 块的参数的异常类型和抛出的异常实例匹配，则将异常实例赋给该 catch 块的参数，执行该 catch 块的语句。如果在当前方法中没有发现异常处理器，则异常没有被捕获和处理，退出当前的方法，并将异常传递给当前方法的调用者，继续寻找异常处理器。

4.finally：有时，无论异常是否出现或者被捕获，都需要执行一些语句，可以通过 finally 子句实现。使用关键字 finally 声明 finally 子句。

如果在 try-catch 块中包含 return 语句，finally 子句将在方法返回之前被执行

## JVM

Jvm是面向操作系统的。它负责把class字节码解释成系统所能识别的指令并执行。

同时也负责程序运行时内存的管理。

### 1.Runtime 数据区域

Java 虚拟机在执行 Java 程序的过程中会把它管理的内存划分成若干个不同的数据区域，不同区域有不同的用途。

注意点：

**【线程私有的区域】 程序计数器，虚拟机栈，本地方法栈**

**【线程共有的进程区域】堆，方法区**

1. 程序计数器

* 当前线程所执行的字节码的行号指示器。
* 为了线程切换后能恢复到正确的执行位置，每个线程都需要有独立的程序计数器。
* 这类内存区域被称为线程私有的内存区域

1. Java 虚拟机栈

* 虚拟机栈为虚拟机执行 Java 方法服务
* 线程私有的，它的生命周期与线程相同。
* **是Java 方法执行的内存模型，每个方法被执行的时候会创建一个栈帧，用于存储局部变量表、操作栈、动态链接、方法出口等信息**
* 一个方法被调用直至执行完成的过程对应一个栈帧在虚拟机中从入栈到出栈的过程。
* 局部变量表存放编译器可知的各种基本数据类型、对象引用类型和返回地址类型。

注意点：

* **如果虚拟机栈不可以动态扩展，当线程请求的栈深度大于虚拟机所允许的深度时，将抛出 StackOverflowError 异常**；
* **如果虚拟机栈可以动态扩展，当无法申请到足够的内存时，将抛出 OutOfMemoryError 异常。**

1. 本地方法栈

* 本地方法栈为虚拟机使用到的本地方法服务
* 本地方法栈也会出现 StackOverflowError 和 OutOfMemoryError 两种异常。

1. **Java堆**

Java 堆是被所有线程共享的内存区域，其目的是**存放对象实例，几乎所有的对象实例都在堆中分配内存**

* Java 堆是垃圾回收器管理的主要内存，因此也称为 GC 堆（Garbage Collected Heap）
* 从垃圾回收的角度，由于现代编译器基本都采用分代垃圾回收算法，所以 Java 堆还可以分成新生代和老年代

1. **方法区**

方法区**用于存储已经被虚拟机加载的类信息、常量、静态变量**、即时编译器编译后的代码等数据。

* 方法区也是被所有线程共享的内存区域
* 当方法区无法满足内存分配需求时，将抛出 OutOfMemoryError 异常。
* JDK 1.8 将方法区彻底移除，取而代之的是元空间，元空间使用的是直接内存。

Runtime常量池

* 是方法区的一部分。
* 常量池信息，用于存放编译器生成的字面量和符号引用，这些信息将在类加载后存放到方法区的运行时常量池中
* 运行时常量池也受到方法区内存的限制，当常量池无法再申请到内存时将抛出 OutOfMemoryError 异常

1. 直接内存（**不是虚拟机运行时数据区域的一部分**）

直接内存不是虚拟机规范中定义的内存区域，但是这部分也被频繁地使用，而且也可能导致 OutOfMemoryError 异常出现。

本机直接内存的分配不受到 Java 堆大小的限制，但是直接内存仍然受到本机总内存地大小及处理器寻址空间的限制。如果各个内存区域的总和大于物理内存限制，就会导致动态扩展时出现 OutOfMemoryError 异常

### 2.垃圾回收

**主要是针对生命周期未结束但是已经无人再使用的空间的回收**。

Java 内存运行时数据区域包括程序计数器、虚拟机栈、本地方法栈、堆等区域。

其中，程序计数器、虚拟机栈和本地方法栈都是线程私有的，当线程结束时，这些区域的生命周期也结束了，因此不需要过多考虑回收的问题。

而**堆是虚拟机管理的内存中最大的一块，堆中的内存的分配和回收是动态的，垃圾回收主要关注的是堆空间**

1. **调用垃圾回收器的方法**

调用垃圾回收器的方法是**gc**，该方法在 System 类和 Runtime 类中都存在。

方法 System.gc 的作用是提示 Java 虚拟机进行垃圾回收，该方法被调用之后，由 Java 虚拟机决定是立即回收还是延迟回收。

在 System 类中，方法 gc 是静态方法，

在 Runtime 类中，方法 gc 是实例方法，

方法 System.gc 会调用 Runtime 类中的 gc 方法，System.gc 等价于 Runtime.getRuntime.gc。

与垃圾回收有关的另一个方法是**finalize 方法**。该方法在 Object 类中被定义，在释放对象占用的内存之前会调用该方法。该方法的默认实现不做任何事，如果必要，子类应该重写该方法，一般建议在该方法中释放对象持有的资源。

1. **判断对象是否可回收**

垃圾回收器在对堆进行回收之前，首先需要确定哪些对象是可回收的。常用的算法有两种，引用计数算法和根搜索算法。

* 根搜索算法的思路是，从若干被称为 **GC Roots 的对象**开始进行搜索，不能到达的对象即为可回收的对象。（主流的商用程序语言都是使用根搜索算法判断对象是否可回收）
* 引用计数算法给每个对象添加引用计数器，用于记录对象被引用的计数，引用计数为 0 的对象即为可回收的对象。（ Java 语言没有使用引用计数算法）
  + 优点：实现简单，判定效率也很高
  + 缺点：无法解决对象之间循环引用的情况

注意：在 Java 中，**GC Roots 一般包含下面几种对象：**

* 虚拟机栈中引用的对象；
* 本地方法栈中的本地方法引用的对象；
* 方法区中的类静态属性引用的对象；
* 方法区中的常量引用的对象。

1. **引用的分类**

引用计数算法和根搜索算法都需要通过判断引用的方式判断对象是否可回收。

在 JDK 1.2 之后，Java 将引用分成四种，按照引用强度从高到低的顺序依次是：强引用、软引用、弱引用、虚引用。

* 强引用是指在程序代码中普遍存在的引用。垃圾回收器永远不会回收被强引用关联的对象。
* 软引用描述还有用但并非必需的对象。只有在系统将要发生内存溢出异常时，被软引用关联的对象才会被回收。在 JDK 1.2 之后，提供了 SoftReference 类实现软引用。
* 弱引用描述非必需的对象，其强度低于软引用。被弱引用关联的对象只能存活到下一次垃圾回收发生之前，当垃圾回收器工作时，被弱引用关联的对象一定会被回收。在 JDK 1.2 之后，提供了 WeakReference 类实现弱引用。
* 虚引用是最弱的引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是能在这个对象被回收时收到一个系统通知。在 JDK 1.2 之后，提供了 PhantomReference 类实现虚引用。

1. **垃圾回收算法**

一般把 Java 堆分为新生代和老年代。在新生代中，大多数对象的生命周期都很短，因此选用复制算法。在老年代中，对象存活率高，因此选用标记—清除算法或标记—整理算法。

* 标记—清除算法是最基础的垃圾回收算法，分为“标记”和“清除”两个阶段，首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。
  + 缺点：一是效率问题，标记和清除的效率都不高，二是空间问题，标记清除之后会产生大量不连续的内存碎片，导致程序在之后的运行过程中无法为较大对象找到足够的连续内存。
* **复制算法**是将可用内存分成大小相等的两块，每次只使用其中的一块。当用完一块内存时，将还存活着的对象复制到另外一块内存，然后把已使用过的内存空间一次清理掉。
  + 优点：实现简单，运行高效，在内存分配时不需要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。
  + 缺点是将内存缩小为了原来的一半，以及在对象存活率较高时复制操作的次数较多，导致效率降低
* **标记—整理算法**是根据老年代的特点提出的。标记过程与标记—清除算法一样，但后续步骤不是直接回收被标记的对象，而是让所有存活的对象都向一端移动，然后清除边界以外的内存。
* 分代收集算法根据对象的存活周期不同将内存划分为多个区域，对每个区域选用不同的垃圾回收算法。

1. **分配内存和回收策略**

Java 堆可以分成新生代和老年代。

新生代又可以细分成 Eden 区、From Survivor 区、To Survivor 区等。

对于不同的对象，有相应的内存分配规则。

* 垃圾回收操作
  + 新生代的垃圾回收操作：Minor GC （因为大多数对象的生命周期都很短，因此 Minor GC 会频繁执行，一般回收速度也比较快。）
  + 老年代的垃圾回收操作：Full GC（也称 Major GC），出现了 Full GC，经常会伴随至少一次的 Minor GC。老年代对象的存活时间长，因此 Full GC 很少执行，而且执行速度会比 Minor GC 慢很多。

注意点：在对新生代进行垃圾回收操作（Minor GC）之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象的空间总和，如果大于，那么Minor GC 可以确保是安全的，否则会进行Full GC（先老年代垃圾回收）

* 内存分配策略

虚拟机采用分代收集的思想管理内存，因此需要识别每个对象应该放在新生代还是老年代。

* + **对象优先在Eden区分配：** 大多数情况下，对象在新生代 Eden 区分配，当 Eden 区空间不够时，发起 Minor GC
  + **大对象直接进入老年代：**大对象是指需要连续内存空间的对象，最典型的大对象是那种很长的字符串以及数组。大对象对于虚拟机的内存分配而言是坏消息，经常出现大对象会导致内存还有不少空间时就提前触发垃圾回收以获取足够的连续空间分配给大对象。（**将大对象直接在老年代中分配**的目的是避免在 Eden 区和 Survivor 区之间出现大量内存复制。）
  + **长期存活的对象进入老年代：**虚拟机给每个对象定义了年龄计数器，对象在 Eden 区出生之后，如果经过第一次 Minor GC 之后仍然存活，将进入 Survivor 区，同时对象年龄变为 1，对象在 Survivor 区每经过一次 Minor GC 且存活，年龄就增加 1，增加到一定阈值时则进入老年代（阈值默认为 15）。

**注意点：动态对象年龄判定**

为了能更好地适应不同程序的内存状况，虚拟机并不总是要求对象的年龄必须达到阈值才能进入老年代。如果在 Survivor 区中相同年龄的所有对象的空间总和大于 Survivor 区空间的一半，则年龄大于或等于该年龄的对象直接进入老年代。

## 多线程

进程是程序的一次执行过程。

线程是进程划分成的更小的运行单位。

### 1.进程

进程是程序的一次执行过程，是系统运行程序的基本单位。

系统运行一个程序即为一个进程的创建、运行以及消亡的过程。

多个线程共享进程的堆和方法区内存资源。

### 2.线程

多线程是开发高并发系统的基础（即允许在同一时间段执行多个任务），利用好多线程机制可以显著提高系统的并发能力和性能。（线程是程序执行的最小单位，多个线程同时运行可以减少线程上下文切换的开销）

一个进程在其执行的过程中可以产生多个线程，**多个线程共享进程的堆和方法区内存资源。**

每个线程都有自己的程序计数器、虚拟机栈和本地方法栈。

注意：

* 进程和线程**最大的区别是，各进程是独立的，而各线程则不一定独立**，因为同一进程中的多个线程极有可能会相互影响
* 线程也称为轻量级进程，是程序执行的最小单位，线程间的切换和调度的成本远远小于进程。多线程是开发高并发系统的基础（多个线程同时运行可以减少线程上下文切换的开销）
* 多线程并发编程并不总是能提高程序的执行效率和运行速度，而且可能存在一些问题，包括内存泄漏、上下文切换、死锁以及受限于硬件和软件的资源限制问题等。

线程状态：

线程在运行的生命周期中的任何时刻只能是 6 种不同状态的其中一种。

* 初始状态（NEW）：线程已经构建，尚未启动。
* 运行状态（RUNNABLE）：包括就绪（READY）和运行中（RUNNING）两种状态，统称为运行状态。
* 阻塞状态（BLOCKED）：线程被**锁**阻塞。
* 等待状态（WAITING）：线程需要等待其他线程做出特定动作（通知或中断）。
* 超时等待状态（TIME\_WAITING）：不同于等待状态，**超时等待状态可以在指定的时间自行返回。**
* 终止状态（TERMINATED）：当前线程已经执行完毕。

### 3. 多线程的资源同步问题

多线程之间由于并不独立，所以涉及资源同步与互斥的问题。

* 关键字**synchronized** 解决的是多个线程之间访问资源的同步性，**该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**。

关键字 synchronized 最主要的三种使用方式是：修饰实例方法、修饰静态方法、修饰代码块。

* 修饰实例方法：给当前对象实例加锁，进入同步代码之前需要获得当前对象实例的锁。
* 修饰静态方法：给当前类加锁，进入同步代码之前需要获得当前类的锁。
* 修饰代码块：指定加锁对象，给指定对象加锁，进入同步代码块之前需要获得指定对象的锁。
* 关键字 **volatile** 解决的是变量在多个线程之间的可见性，以及防止指令进行重排序优化。

**该关键字修饰的变量会直接在主内存中进行读写操作，保证了变量的可见性**。

为了提高执行程序时的性能，编译器和处理器会对指令进行重排序优化，因此代码的执行顺序和编写代码的顺序可能不一致。添加关键字 volatile 可以禁止指令进行重排序优化。

只有当一个变量满足以下两个条件时，才能使用关键字 volatile。

* 对变量的写入操作不依赖变量的当前值，或者能确保只有单个线程更新变量的值。
* 该变量没有包含在具有其他变量的不变式中。

### 多线程方法

#### 4.1 Thread 类

创建一个 Thread 类的实例，即为创建了一个处于初始状态的线程。

* start方法：对一个处于初始状态的线程调用方法 start，该线程被启动，进入运行状态。

注意：

* 方法 run 在 Runnable 接口中被定义，调用方法 start 之后，方法 run 会自动执行
* 通过调用方法 start，执行方法 run，才是多线程工作。如果直接执行方法 run，则方法 run 会被当成一个主线程下的普通方法执行，而不会在某个线程中执行，因此不是多线程工作。
* sleep方法: 使当前线程**暂停**执行一段时间，让其他线程有机会继续执行，但是该方法不会释放锁。方法 sleep 需要捕获 InterruptedException 异常。
* join方法：阻塞调用该方法的线程，直到当前线程执行完毕之后，调用该方法的线程再继续执行。方法 join 需要捕获 InterruptedException 异常。
* yield方法：暂停当前正在执行的线程对象，并执行其他线程。实际调用方法 yield 时无法保证一定能让其他线程执行，因为线程调度时可能再次选中原来的线程对象。
* interrupt方法：中断线程，该方法不会中断一个正在运行的线程，只是改变中断标记。
  + 当线程处于运行状态时，可以对线程调用方法 interrupt 将线程的中断标记设为 true从而达到终止线程的目的，也可以添加一个 volatile 修饰的额外标记，当需要终止线程时，更改该标记的值即可。
  + 当线程处于等待状态、超时等待状态或阻塞状态时，如果对线程调用方法 interrupt 将线程的中断标记设为 true，则中断标记会被清除，同时会抛出 InterruptedException 异常。**可以通过 try-catch 块捕获该异常，即可终止线程**。

#### Object 类

* wait：该方法必须在 synchronized 语句块内使用，作用是释放锁，让其他线程可以运行，当前线程进入等待池中。
* notify ：从等待池中移走任意一个等待当前对象的线程并放到锁池中，只有锁池中的线程可以获取锁
* notifyAll ：从等待池中移走全部等待当前对象的线程并放到锁池中，锁池中的这些线程将争夺锁。

### 线程池

线程池是一种线程的使用模式。创建若干个可执行的线程放入一个池（容器）中，有任务需要处理时，会提交到线程池中的任务队列，处理完之后线程并不会被销毁，而是仍然在线程池中等待下一个任务。

好处：

* 降低资源消耗。重复利用线程池中已经创建的线程，可以避免频繁地创建和销毁线程，从而减少资源消耗。
* 提高响应速度。由于线程池中有已经创建的线程，因此当任务到达时，可以直接执行，不需要等待线程创建。
* 提高线程的可管理性。线程是稀缺资源，如果无限制地创建，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分配、调优和监控。
* 创建线程池

public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler)

|  |  |
| --- | --- |
| corePoolSize | 核心线程数，定义了最少可以同时运行的线程数量，当有新的任务时就会创建一个线程执行任务，当线程池中的线程数量达到 corePoolSize 之后，到达的任务进入阻塞队列 |
| maximumPoolSize | 最大线程数，定义了线程池中最多能创建的线程数量 |
| keepAliveTime | 等待时间，当线程池中的线程数量大于 corePoolSize 时，如果一个线程的空闲时间达到 keepAliveTime 时则会终止，直到线程池中的线程数不超过 corePoolSize |
| unit | 参数 keepAliveTime 的单位 |
| workQueue | 阻塞队列，用来存储等待执行的任务 |
| threadFactory | 创建线程的工厂 |
| handler | 当拒绝处理任务时的策略 |

注意：使用顺序：核心线程—》阻塞队列—》非核心线程—》抛出RejectExecutionException 异常

* shutdown 或 shutdownNow 关闭线程池。这两个方法的原理是遍历线程池中的工作线程，对每个工作线程调用 interrupt 方法中断线程，无法响应中断的任务可能永远无法终止。

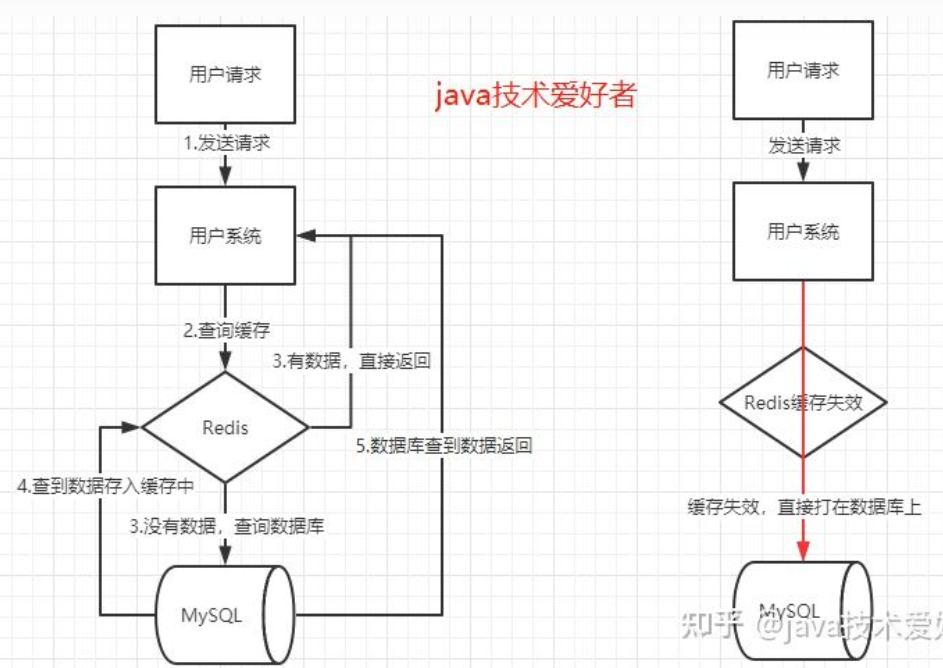
## Java高并发

### Redis缓存

缓存雪崩，缓存击穿，缓存穿透，是使用redis过程中三个比较致命的问题。在日常开发中一定要注意，每次使用Redis时，都要对其保持严谨的态度。还有一个需要注意的是要做好熔断，一旦出现缓存雪崩，击穿，穿透这种情况，至少还有熔断机制保护数据库不会被打死。

#### 1.缓存雪崩

当某一个时刻出现大规模的缓存失效的情况，那么就会导致大量的请求直接打在数据库上面，导致数据库压力巨大，如果在高并发的情况下，可能瞬间就会导致数据库宕机。这时候如果运维马上又重启数据库，马上又会有新的流量把数据库打死。这就是缓存雪崩。



【出现缓存雪崩的原因】

**同一时间大规模的key失效**。

第一种可能是Redis宕机，

第二种可能是采用了相同的过期时间。

【解决方案】

1、在原有的失效时间上加上一个随机值，比如1-5分钟随机。这样就避免了因为采用相同的过期时间导致的缓存雪崩。

2、使用熔断机制。当流量到达一定的阈值时，就直接返回“系统拥挤”之类的提示，防止过多的请求打在数据库上。至少能保证一部分用户是可以正常使用，其他用户多刷新几次也能得到结果。

3、提高数据库的容灾能力，可以使用分库分表，读写分离的策略。

4、为了防止Redis宕机导致缓存雪崩的问题，可以搭建Redis集群，提高Redis的容灾性。

#### 2.缓存击穿

缓存击穿是一个热点的Key，有大并发集中对其进行访问，突然间这个Key失效了，导致大并发全部打在数据库上，导致数据库压力剧增。这种现象就叫做缓存击穿

【分析】关键在于某个热点的key失效了，导致大并发集中打在数据库上。第一是否可以考虑热点key不设置过期时间，第二是否可以考虑降低打在数据库上的请求数量。

【解决方案】

1、上面说过了，如果业务允许的话，对于热点的key可以设置永不过期的key。

2、使用互斥锁。如果缓存失效的情况，只有拿到锁才可以查询数据库，降低了在同一时刻打在数据库上的请求，防止数据库打死。当然这样会导致系统的性能变差。

#### 3.缓存穿透

我们使用Redis大部分情况都是通过Key查询对应的值，假如发送的请求传进来的key是不存在Redis中的，那么就查不到缓存，查不到缓存就会去数据库查询。假如有大量这样的请求，这些请求像“穿透”了缓存一样直接打在数据库上，这种现象就叫做缓存穿透。

【分析】关键在于在Redis查不到key值，这和缓存击穿有根本的区别。

区别在于**缓存穿透的情况是传进来的key在Redis中是不存在的**。假如有黑客传进大量的不存在的key，那么大量的请求打在数据库上是很致命的问题，所以在日常开发中要对参数做好校验，一些非法的参数，不可能存在的key就直接返回错误提示，要对调用方保持这种“不信任”的心态。

【解决方案】

1、**把无效的Key存进Redis中**。如果Redis查不到数据，数据库也查不到，我们把这个Key值保存进Redis，设置value="null"，当下次再通过这个Key查询时就不需要再查询数据库。这种处理方式肯定是有问题的，假如传进来的这个不存在的Key值每次都是随机的，那存进Redis也没有意义。

2、**使用布隆过滤器**。布隆过滤器的作用是某个 key 不存在，那么就一定不存在，它说某个 key 存在，那么很大可能是存在(存在一定的误判率)。于是我们可以在缓存之前再加一层布隆过滤器，在查询的时候先去布隆过滤器查询 key 是否存在，如果不存在就直接返回。

