# 技术面试中的几个注意点:

1 面试时,你熟悉的问题要和面试官多聊,不要为了回答问题而回答问题

2 一个问题的沟通时间最好能多聊一会儿,简单问题说3/5分钟,如果问题的规模比较大,10分钟左右也是可以的

3 回答问题时不要为了凑时间而凑时间,聊的内容一定要和问的问题相关,知识点可以连续的引入

4 了解的东西多聊,不了解的少说

5 对于知识可以有一些自己的见解,自己的想法,清晰表述出来,虽然自己的看法有时候不会特别的恰当.

6 面试时收集面试录音,面试题,可以发给我们 100元/人,多人评比,最好的400元奖金

7 答题思路 总 分 总 点 线 面

# 请聊一下java的集合类，以及在实际项目中你是如何用的？

参照java集合一章

注意说出集合体系,常用类 接口 实现类

加上你所知道的高并发集合类,JUC 参照集合增强内容

在实际项目中引用,照实说就好了

问集合的引子 … …

# Hashmap为什么要使用红黑树？

在jdk1.8版本后，java对HashMap做了改进，在链表长度大于8的时候，将后面的数据存在红黑树中，以加快检索速度

红黑树虽然本质上是一棵二叉查找树，但它在二叉查找树的基础上增加了着色和相关的性质使得红黑树相对平衡，从而保证了红黑树的查找、插入、删除的时间复杂度最坏为O(log n)。加快检索速率。

# 集合类是怎么解决高并发中的问题？

思路 先说一下那些是非安全

普通的安全的集合类

JUC中高并发的集合类

线程非安全的集合类 ArrayList LinkedList HashSet TreeSet HashMap TreeMap 实际开发中我们自己用这样的集合最多,因为一般我们自己写的业务代码中,不太涉及到多线程共享同一个集合的问题

线程安全的集合类 Vector HashTable 虽然效率没有JUC中的高性能集合高,但是也能够适应大部分环境

高性能线程安全的集合类

 1.ConcurrentHashMap

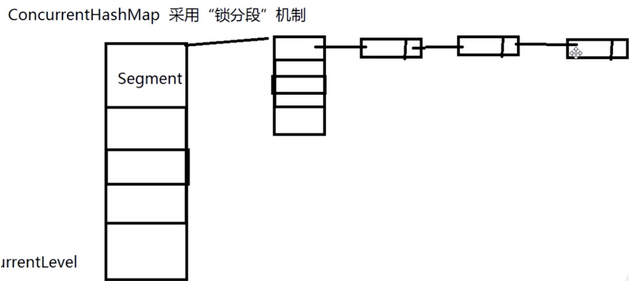
 2.ConcurrentHashMap和HashTable的区别

 3.ConcurrentHashMap线程安全的具体实现方式/底层具体实现

 4.说说CopyOnWriteArrayList

ConcurrentHashMap

java5.0在juc包中提供了大量支持并发的容器类，采用“锁分段”机制，Concurrentlevel分段级别，默认16，就是有16个段（segment)，每个段默认又有16个哈希表（table），每个又有链表连着。



在JDK1.7的时候，ConcurrentHashMap（分段锁） 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争。  
DK1.8ConcurrentHashMap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。Java 8在链表长度超过一定阈值（8）时将链表（寻址时间复杂度为O(N)）转换为红黑树（寻址时间复杂度为O(log(N))）  
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。

ConcurrentSkipListMap是**线程安全的有序的哈希表(相当于线程安全的TreeMap)**; 它**继承于AbstractMap类，并且实现ConcurrentNavigableMap接口**。ConcurrentSkipListMap是**通过“跳表”来实现的**，

ConcurrentSkipListSet是**线程安全的有序的集合(相当于线程安全的TreeSet)**；**它继承于AbstractSet，并实现了NavigableSet接口**。ConcurrentSkipListSet是通过ConcurrentSkipListMap实现的，它也支持并发。

CopyOnWriteArraySet addIfAbsent和 CopyOnWriteArrayList（写入并复制）也是juc里面的，它解决了并发修改异常，每当有写入的时候，就在底层重新复制一个新容器写入，最后把新容器的引用地址赋给旧的容器，在别人写入的时候，其他线程读数据，依然是旧容器的线程。这样是开销很大的，所以不适合频繁写入的操作。适合并发迭代操作多的场景。只能保证数据的最终一致性

# 简述一下自定义异常的应用场景？

借助异常机制,我们可以省略很多业务逻辑上的判断处理,直接借助java的异常机制可以简化业务逻辑判断代码的编写

1当你不想把你的错误直接暴露给前端或者你想让前端从业务角度判断后台的异常，这个时候自定义异常类是你的不二选择

2 虽然JAVA给我们提供了丰富的异常类型,但是在实际的业务上,还有很多情况JAVA提供的异常类型不能准确的表述出我们业务上的含义

3 控制项目的后期服务 … …

# 描述一下Object类中常用的方法？

参照面向对象章节toString hashCode equals clone finalized wait notify notifyAll … …

解释每个方法的作用

toString 定义一个对象的字符串表现形式 Object类中定义的规则是 类的全路径名+@+对象的哈希码 重写之后 我们可以自行决定返回的字符串中包含对象的那些属性信息 …

clone >>>返回一个对象的副本 深克隆 浅克隆 原型模式 重写时实现Cloneable

finalized GC 会调动该方法 自救

# 1.8的新特性有了解过吗？ (注意了解其他版本新特征) +JDK更新认识

* **Lambda表达式**
* **函数式接口 函数式编程**
* **方法引用和构造器调用**
* **Stream API**
* **接口中的默认方法和静态方法**
* **新时间日期API**

新的日期类

|  |  |
| --- | --- |
| 属性 | 含义 |
| Instant | 代表的是时间戳 |
| LocalDate | 代表日期，比如2020-01-14 |
| LocalTime | 代表时刻，比如12:59:59 |
| LocalDateTime | 代表具体时间 2020-01-12 12:22:26 |
| ZonedDateTime | 代表一个包含时区的完整的日期时间，偏移量是以UTC/ 格林威治时间为基准的 |
| Period | 代表时间段 |
| ZoneOffset | 代表时区偏移量，比如：+8:00 |
| Clock | 代表时钟，比如获取目前美国纽约的时间 |

## 一、接口的默认方法

Java 8允许我们给接口添加一个非抽象的方法实现，只需要使用 default关键字即可，这个特征又叫做扩展方法，示例如下：

代码如下:

interface Formula { double calculate(int a);

default double sqrt(int a) { return Math.sqrt(a); } }

Formula接口在拥有calculate方法之外同时还定义了sqrt方法，实现了Formula接口的子类只需要实现一个calculate方法，默认方法sqrt将在子类上可以直接使用。

代码如下:

Formula formula = new Formula() { @Override public double calculate(int a) { return sqrt(a \* 100); } };

formula.calculate(100); // 100.0 formula.sqrt(16); // 4.0

文中的formula被实现为一个匿名类的实例，该代码非常容易理解，6行代码实现了计算 sqrt(a \* 100)。在下一节中，我们将会看到实现单方法接口的更简单的做法。

译者注： 在Java中只有单继承，如果要让一个类赋予新的特性，通常是使用接口来实现，在C++中支持多继承，允许一个子类同时具有多个父类的接口与功能，在其他语言中，让一个类同时具有其他的可复用代码的方法叫做mixin。新的Java 8 的这个特新在编译器实现的角度上来说更加接近Scala的trait。 在C#中也有名为扩展方法的概念，允许给已存在的类型扩展方法，和Java 8的这个在语义上有差别。

## 二、Lambda 表达式

首先看看在老版本的Java中是如何排列字符串的：

代码如下:

List<String> names = Arrays.asList("peterF", "anna", "mike", "xenia");

Collections.sort(names, new Comparator<String>() { @Override public int compare(String a, String b) { return b.compareTo(a); } });

只需要给静态方法 Collections.sort 传入一个List对象以及一个比较器来按指定顺序排列。通常做法都是创建一个匿名的比较器对象然后将其传递给sort方法。

在Java 8 中你就没必要使用这种传统的匿名对象的方式了，Java 8提供了更简洁的语法，lambda表达式：

代码如下:

Collections.sort(names, (String a, String b) -> { return b.compareTo(a); });

看到了吧，代码变得更段且更具有可读性，但是实际上还可以写得更短：

代码如下:

Collections.sort(names, (String a, String b) -> b.compareTo(a));

对于函数体只有一行代码的，你可以去掉大括号{}以及return关键字，但是你还可以写得更短点：

代码如下:

Collections.sort(names, (a, b) -> b.compareTo(a));

Java编译器可以自动推导出参数类型，所以你可以不用再写一次类型。接下来我们看看lambda表达式还能作出什么更方便的东西来：

## 三、函数式接口

Lambda表达式是如何在java的类型系统中表示的呢？每一个lambda表达式都对应一个类型，通常是接口类型。而“函数式接口”是指仅仅只包含一个抽象方法的接口，每一个该类型的lambda表达式都会被匹配到这个抽象方法。因为 默认方法 不算抽象方法，所以你也可以给你的函数式接口添加默认方法。

我们可以将lambda表达式当作任意只包含一个抽象方法的接口类型，确保你的接口一定达到这个要求，你只需要给你的接口添加 @FunctionalInterface 注解，编译器如果发现你标注了这个注解的接口有多于一个抽象方法的时候会报错的。

示例如下：

代码如下:

@FunctionalInterface interface Converter<F, T> { T convert(F from); } Converter<String, Integer> converter = (from) -> Integer.valueOf(from); Integer converted = converter.convert("123"); System.out.println(converted); // 123

需要注意如果@FunctionalInterface如果没有指定，上面的代码也是对的。

译者注 将lambda表达式映射到一个单方法的接口上，这种做法在Java 8之前就有别的语言实现，比如Rhino JavaScript解释器，如果一个函数参数接收一个单方法的接口而你传递的是一个function，Rhino 解释器会自动做一个单接口的实例到function的适配器，典型的应用场景有 org.w3c.dom.events.EventTarget 的addEventListener 第二个参数 EventListener。

## 四、方法与构造函数引用

前一节中的代码还可以通过静态方法引用来表示：

代码如下:

Converter<String, Integer> converter = Integer::valueOf; Integer converted = converter.convert("123"); System.out.println(converted); // 123

Java 8 允许你使用 :: 关键字来传递方法或者构造函数引用，上面的代码展示了如何引用一个静态方法，我们也可以引用一个对象的方法：

代码如下:

converter = something::startsWith; String converted = converter.convert("Java"); System.out.println(converted); // "J"

接下来看看构造函数是如何使用::关键字来引用的，首先我们定义一个包含多个构造函数的简单类：

代码如下:

class Person { String firstName; String lastName;

Person() {}

Person(String firstName, String lastName) { this.firstName = firstName; this.lastName = lastName; } }

接下来我们指定一个用来创建Person对象的对象工厂接口：

代码如下:

interface PersonFactory<P extends Person> { P create(String firstName, String lastName); }

这里我们使用构造函数引用来将他们关联起来，而不是实现一个完整的工厂：

代码如下:

PersonFactory<Person> personFactory = Person::new; Person person = personFactory.create("Peter", "Parker");

我们只需要使用 Person::new 来获取Person类构造函数的引用，Java编译器会自动根据PersonFactory.create方法的签名来选择合适的构造函数。

## 五、Lambda 作用域

在lambda表达式中访问外层作用域和老版本的匿名对象中的方式很相似。你可以直接访问标记了final的外层局部变量，或者实例的字段以及静态变量。

## 六、访问局部变量

我们可以直接在lambda表达式中访问外层的局部变量：

代码如下:

final int num = 1; Converter<Integer, String> stringConverter = (from) -> String.valueOf(from + num);

stringConverter.convert(2); // 3

但是和匿名对象不同的是，这里的变量num可以不用声明为final，该代码同样正确：

代码如下:

int num = 1; Converter<Integer, String> stringConverter = (from) -> String.valueOf(from + num);

stringConverter.convert(2); // 3

不过这里的num必须不可被后面的代码修改（即隐性的具有final的语义），例如下面的就无法编译：

代码如下:

int num = 1; Converter<Integer, String> stringConverter = (from) -> String.valueOf(from + num); num = 3;

在lambda表达式中试图修改num同样是不允许的。

## 七、访问对象字段与静态变量

和本地变量不同的是，lambda内部对于实例的字段以及静态变量是即可读又可写。该行为和匿名对象是一致的：

代码如下:

class Lambda4 { static int outerStaticNum; int outerNum;

void testScopes() { Converter<Integer, String> stringConverter1 = (from) -> { outerNum = 23; return String.valueOf(from); };

Converter<Integer, String> stringConverter2 = (from) -> { ​ outerStaticNum = 72; ​ return String.valueOf(from); ​ }; } }

## 八、访问接口的默认方法

还记得第一节中的formula例子么，接口Formula定义了一个默认方法sqrt可以直接被formula的实例包括匿名对象访问到，但是在lambda表达式中这个是不行的。 Lambda表达式中是无法访问到默认方法的，以下代码将无法编译：

代码如下:

Formula formula = (a) -> sqrt( a \* 100); Built-in Functional Interfaces

JDK 1.8 API包含了很多内建的函数式接口，在老Java中常用到的比如Comparator或者Runnable接口，这些接口都增加了@FunctionalInterface注解以便能用在lambda上。 Java 8 API同样还提供了很多全新的函数式接口来让工作更加方便，有一些接口是来自Google Guava库里的，即便你对这些很熟悉了，还是有必要看看这些是如何扩展到lambda上使用的。

**Predicate接口**

Predicate 接口只有一个参数，返回boolean类型。该接口包含多种默认方法来将Predicate组合成其他复杂的逻辑（比如：与，或，非）：

代码如下:

Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test("foo"); // true predicate.negate().test("foo"); // false

Predicate<Boolean> nonNull = Objects::nonNull; Predicate<Boolean> isNull = Objects::isNull;

Predicate<String> isEmpty = String::isEmpty; Predicate<String> isNotEmpty = isEmpty.negate();

**Function 接口**

Function 接口有一个参数并且返回一个结果，并附带了一些可以和其他函数组合的默认方法（compose, andThen）：

代码如下:

Function<String, Integer> toInteger = Integer::valueOf; Function<String, String> backToString = toInteger.andThen(String::valueOf);

backToString.apply("123"); // "123"

**Supplier 接口** Supplier 接口返回一个任意范型的值，和Function接口不同的是该接口没有任何参数

代码如下:

Supplier<Person> personSupplier = Person::new; personSupplier.get(); // new Person

**Consumer 接口** Consumer 接口表示执行在单个参数上的操作。

代码如下:

Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName); greeter.accept(new Person("Luke", "Skywalker"));

**Comparator 接口** Comparator 是老Java中的经典接口， Java 8在此之上添加了多种默认方法：

代码如下:

Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);

Person p1 = new Person("John", "Doe"); Person p2 = new Person("Alice", "Wonderland");

comparator.compare(p1, p2); // > 0 comparator.reversed().compare(p1, p2); // < 0

**Optional 接口**

Optional 不是函数是接口，这是个用来防止NullPointerException异常的辅助类型，这是下一届中将要用到的重要概念，现在先简单的看看这个接口能干什么：

Optional 被定义为一个简单的容器，其值可能是null或者不是null。在Java 8之前一般某个函数应该返回非空对象但是偶尔却可能返回了null，而在Java 8中，不推荐你返回null而是返回Optional。

代码如下:

Optional<String> optional = Optional.of("bam");

optional.isPresent(); // true optional.get(); // "bam" optional.orElse("fallback"); // "bam"

optional.ifPresent((s) -> System.out.println(s.charAt(0))); // "b"

**Stream 接口**

java.util.Stream 表示能应用在一组元素上一次执行的操作序列。Stream 操作分为中间操作或者最终操作两种，最终操作返回一特定类型的计算结果，而中间操作返回Stream本身，这样你就可以将多个操作依次串起来。Stream 的创建需要指定一个数据源，比如 java.util.Collection的子类，List或者Set， Map不支持。Stream的操作可以串行执行或者并行执行。

首先看看Stream是怎么用，首先创建实例代码的用到的数据List：

代码如下:

List<String> stringCollection = new ArrayList<>(); stringCollection.add("ddd2"); stringCollection.add("aaa2"); stringCollection.add("bbb1"); stringCollection.add("aaa1"); stringCollection.add("bbb3"); stringCollection.add("ccc"); stringCollection.add("bbb2"); stringCollection.add("ddd1");

Java 8扩展了集合类，可以通过 Collection.stream() 或者 Collection.parallelStream() 来创建一个Stream。下面几节将详细解释常用的Stream操作：

**Filter 过滤**

过滤通过一个predicate接口来过滤并只保留符合条件的元素，该操作属于中间操作，所以我们可以在过滤后的结果来应用其他Stream操作（比如forEach）。forEach需要一个函数来对过滤后的元素依次执行。forEach是一个最终操作，所以我们不能在forEach之后来执行其他Stream操作。

代码如下:

stringCollection .stream() .filter((s) -> s.startsWith("a")) .forEach(System.out::println);

// "aaa2", "aaa1"

**Sort 排序**

排序是一个中间操作，返回的是排序好后的Stream。如果你不指定一个自定义的Comparator则会使用默认排序。

代码如下:

stringCollection .stream() .sorted() .filter((s) -> s.startsWith("a")) .forEach(System.out::println);

// "aaa1", "aaa2"

需要注意的是，排序只创建了一个排列好后的Stream，而不会影响原有的数据源，排序之后原数据stringCollection是不会被修改的：

代码如下:

System.out.println(stringCollection); // ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1

**Map 映射** 中间操作map会将元素根据指定的Function接口来依次将元素转成另外的对象，下面的示例展示了将字符串转换为大写字符串。你也可以通过map来讲对象转换成其他类型，map返回的Stream类型是根据你map传递进去的函数的返回值决定的。

代码如下:

stringCollection .stream() .map(String::toUpperCase) .sorted((a, b) -> b.compareTo(a)) .forEach(System.out::println);

// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"

**Match 匹配**

Stream提供了多种匹配操作，允许检测指定的Predicate是否匹配整个Stream。所有的匹配操作都是最终操作，并返回一个boolean类型的值。

代码如下:

boolean anyStartsWithA = stringCollection .stream() .anyMatch((s) -> s.startsWith("a"));

System.out.println(anyStartsWithA); // true

boolean allStartsWithA = stringCollection .stream() .allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA); // false

boolean noneStartsWithZ = stringCollection .stream() .noneMatch((s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ); // true

**Count 计数** 计数是一个最终操作，返回Stream中元素的个数，返回值类型是long。

代码如下:

long startsWithB = stringCollection .stream() .filter((s) -> s.startsWith("b")) .count();

System.out.println(startsWithB); // 3

**Reduce 规约**

这是一个最终操作，允许通过指定的函数来讲stream中的多个元素规约为一个元素，规越后的结果是通过Optional接口表示的：

代码如下:

Optional<String> reduced = stringCollection .stream() .sorted() .reduce((s1, s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println); // "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"

**并行Streams**

前面提到过Stream有串行和并行两种，串行Stream上的操作是在一个线程中依次完成，而并行Stream则是在多个线程上同时执行。

下面的例子展示了是如何通过并行Stream来提升性能：

首先我们创建一个没有重复元素的大表：

代码如下:

int max = 1000000; List<String> values = new ArrayList<>(max); for (int i = 0; i < max; i++) { UUID uuid = UUID.randomUUID(); values.add(uuid.toString()); }

然后我们计算一下排序这个Stream要耗时多久， 串行排序：

代码如下:

long t0 = System.nanoTime();

long count = values.stream().sorted().count(); System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0); System.out.println(String.format("sequential sort took: %d ms", millis));

// 串行耗时: 899 ms 并行排序：

代码如下:

long t0 = System.nanoTime();

long count = values.parallelStream().sorted().count(); System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0); System.out.println(String.format("parallel sort took: %d ms", millis));

// 并行排序耗时: 472 ms 上面两个代码几乎是一样的，但是并行版的快了50%之多，唯一需要做的改动就是将stream()改为parallelStream()。

**Map**

前面提到过，Map类型不支持stream，不过Map提供了一些新的有用的方法来处理一些日常任务。

代码如下:

Map<Integer, String> map = new HashMap<>();

for (int i = 0; i < 10; i++) { map.putIfAbsent(i, "val" + i); }

map.forEach((id, val) -> System.out.println(val)); 以上代码很容易理解， putIfAbsent 不需要我们做额外的存在性检查，而forEach则接收一个Consumer接口来对map里的每一个键值对进行操作。

下面的例子展示了map上的其他有用的函数：

代码如下:

map.computeIfPresent(3, (num, val) -> val + num); map.get(3); // val33

map.computeIfPresent(9, (num, val) -> null); map.containsKey(9); // false

map.computeIfAbsent(23, num -> "val" + num); map.containsKey(23); // true

map.computeIfAbsent(3, num -> "bam"); map.get(3); // val33

接下来展示如何在Map里删除一个键值全都匹配的项：

代码如下:

map.remove(3, "val3"); map.get(3); // val33

map.remove(3, "val33"); map.get(3); // null

另外一个有用的方法：

代码如下:

map.getOrDefault(42, "not found"); // not found

对Map的元素做合并也变得很容易了：

代码如下:

map.merge(9, "val9", (value, newValue) -> value.concat(newValue)); map.get(9); // val9

map.merge(9, "concat", (value, newValue) -> value.concat(newValue)); map.get(9); // val9concat

Merge做的事情是如果键名不存在则插入，否则则对原键对应的值做合并操作并重新插入到map中。

## 九、Date API

Java 8 在包java.time下包含了一组全新的时间日期API。新的日期API和开源的Joda-Time库差不多，但又不完全一样，下面的例子展示了这组新API里最重要的一些部分：

**Clock 时钟**

Clock类提供了访问当前日期和时间的方法，Clock是时区敏感的，可以用来取代 System.currentTimeMillis() 来获取当前的微秒数。某一个特定的时间点也可以使用Instant类来表示，Instant类也可以用来创建老的java.util.Date对象。

代码如下:

Clock clock = Clock.systemDefaultZone(); long millis = clock.millis();

Instant instant = clock.instant(); Date legacyDate = Date.from(instant); // legacy java.util.Date

**Timezones 时区**

在新API中时区使用ZoneId来表示。时区可以很方便的使用静态方法of来获取到。 时区定义了到UTS时间的时间差，在Instant时间点对象到本地日期对象之间转换的时候是极其重要的。

代码如下:

System.out.println(ZoneId.getAvailableZoneIds()); // prints all available timezone ids

ZoneId zone1 = ZoneId.of("Europe/Berlin"); ZoneId zone2 = ZoneId.of("Brazil/East"); System.out.println(zone1.getRules()); System.out.println(zone2.getRules());

// ZoneRules[currentStandardOffset=+01:00] // ZoneRules[currentStandardOffset=-03:00]

**LocalTime 本地时间**

LocalTime 定义了一个没有时区信息的时间，例如 晚上10点，或者 17:30:15。下面的例子使用前面代码创建的时区创建了两个本地时间。之后比较时间并以小时和分钟为单位计算两个时间的时间差：

代码如下:

LocalTime now1 = LocalTime.now(zone1); LocalTime now2 = LocalTime.now(zone2);

System.out.println(now1.isBefore(now2)); // false

long hoursBetween = ChronoUnit.HOURS.between(now1, now2); long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);

System.out.println(hoursBetween); // -3 System.out.println(minutesBetween); // -239

LocalTime 提供了多种工厂方法来简化对象的创建，包括解析时间字符串。

代码如下:

LocalTime late = LocalTime.of(23, 59, 59); System.out.println(late); // 23:59:59

DateTimeFormatter germanFormatter = DateTimeFormatter .ofLocalizedTime(FormatStyle.SHORT) .withLocale(Locale.GERMAN);

LocalTime leetTime = LocalTime.parse("13:37", germanFormatter); System.out.println(leetTime); // 13:37

**LocalDate 本地日期**

LocalDate 表示了一个确切的日期，比如 2014-03-11。该对象值是不可变的，用起来和LocalTime基本一致。下面的例子展示了如何给Date对象加减天/月/年。另外要注意的是这些对象是不可变的，操作返回的总是一个新实例。

代码如下:

LocalDate today = LocalDate.now(); LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS); LocalDate yesterday = tomorrow.minusDays(2);

LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4); DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();

System.out.println(dayOfWeek); // FRIDAY 从字符串解析一个LocalDate类型和解析LocalTime一样简单：

代码如下:

DateTimeFormatter germanFormatter = DateTimeFormatter .ofLocalizedDate(FormatStyle.MEDIUM) .withLocale(Locale.GERMAN);

LocalDate xmas = LocalDate.parse("24.12.2014", germanFormatter); System.out.println(xmas); // 2014-12-24

**LocalDateTime 本地日期时间**

LocalDateTime 同时表示了时间和日期，相当于前两节内容合并到一个对象上了。LocalDateTime和LocalTime还有LocalDate一样，都是不可变的。LocalDateTime提供了一些能访问具体字段的方法。

代码如下:

LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER, 31, 23, 59, 59);

DayOfWeek dayOfWeek = sylvester.getDayOfWeek(); System.out.println(dayOfWeek); // WEDNESDAY

Month month = sylvester.getMonth(); System.out.println(month); // DECEMBER

long minuteOfDay = sylvester.getLong(ChronoField.MINUTE\_OF\_DAY); System.out.println(minuteOfDay); // 1439

只要附加上时区信息，就可以将其转换为一个时间点Instant对象，Instant时间点对象可以很容易的转换为老式的java.util.Date。

代码如下:

Instant instant = sylvester .atZone(ZoneId.systemDefault()) .toInstant();

Date legacyDate = Date.from(instant); System.out.println(legacyDate); // Wed Dec 31 23:59:59 CET 2014

格式化LocalDateTime和格式化时间和日期一样的，除了使用预定义好的格式外，我们也可以自己定义格式：

代码如下:

DateTimeFormatter formatter = DateTimeFormatter .ofPattern("MMM dd, yyyy - HH:mm");

LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13", formatter); String string = formatter.format(parsed); System.out.println(string); // Nov 03, 2014 - 07:13

和java.text.NumberFormat不一样的是新版的DateTimeFormatter是不可变的，所以它是线程安全的。

## 十、Annotation 注解

在Java 8中支持多重注解了，先看个例子来理解一下是什么意思。 首先定义一个包装类Hints注解用来放置一组具体的Hint注解：

代码如下:

@interface Hints { Hint[] value(); }

@Repeatable(Hints.class) @interface Hint { String value(); }

Java 8允许我们把同一个类型的注解使用多次，只需要给该注解标注一下@Repeatable即可。

例 1: 使用包装类当容器来存多个注解（老方法）

代码如下:

@Hints({@Hint("hint1"), @Hint("hint2")}) class Person {}

例 2：使用多重注解（新方法）

代码如下:

@Hint("hint1") @Hint("hint2") class Person {}

第二个例子里java编译器会隐性的帮你定义好@Hints注解，了解这一点有助于你用反射来获取这些信息：

代码如下:

Hint hint = Person.class.getAnnotation(Hint.class); System.out.println(hint); // null

Hints hints1 = Person.class.getAnnotation(Hints.class); System.out.println(hints1.value().length); // 2

Hint[] hints2 = Person.class.getAnnotationsByType(Hint.class); System.out.println(hints2.length); // 2

即便我们没有在Person类上定义@Hints注解，我们还是可以通过 getAnnotation(Hints.class) 来获取 @Hints注解，更加方便的方法是使用 getAnnotationsByType 可以直接获取到所有的@Hint注解。 另外Java 8的注解还增加到两种新的target上了：

代码如下:

@Target({ElementType.TYPE\_PARAMETER, ElementType.TYPE\_USE}) @interface MyAnnotation {}

关于Java 8的新特性就写到这了，肯定还有更多的特性等待发掘。JDK 1.8里还有很多很有用的东西，比如Arrays.parallelSort, StampedLock和CompletableFuture等等。

# 简述一下Java面向对象的基本特征，继承、封装与多态，以及你自己的应用？

知识参照面向对象章节

注意单独解释 继承 封装 多态的概念

继承 基本概念解释 后面多态的条件

封装 基本概念解释 隐藏实现细节,公开使用方式

多态 基本概念解释 就是处理参数 提接口 打破单继承

设计模式 设计原则

# Java中重写和重载的区别？

联系: 名字相似 都是多个同名方法

重载 在同一个类之中发生的

重写 继承中,子类重写父类方法

1 目的差别

2 语法差别

# 怎样声明一个类不会被继承，什么场景下会用？

final修饰的类不能有子类 大部分都是出于安全考虑

String举例

# Java中的自增是线程安全的吗，如何实现线程安全的自增？

i++ ++i

增加synchronized进行线程同步

使用lock、unlock处理Reetrantent 锁进行锁定

AtomicInteger >>> Unsafe >>> cas >>> aba

首先说明，此处 AtomicInteger，一个提供原子操作的 Integer 的类，常见的还有AtomicBoolean、AtomicInteger、AtomicLong、AtomicReference 等，他们的实现原理相同，区别在与运算对象类型的不同。令人兴奋地，还可以通过 AtomicReference<V>将一个对象的所有操作转化成原子操作。

我们知道，在多线程程序中，诸如++i 或 i++等运算不具有原子性，是不安全的线程操作之一。通常我们会使用 synchronized 将该操作变成一个原子操作，但 JVM 为此类操作特意提供了一些同步类，使得使用更方便，且使程序运行效率变得更高。通过相关资料显示，通常AtomicInteger 的性能是 ReentantLock 的好几倍。

# Jdk1.8中的stream有用过吗，详述一下stream的并行操作原理？stream并行的线程池是从哪里来的？

Stream作为Java8的一大亮点，它与java.io包里的InputStream和OutputStream是完全不同的概念。它是对容器对象功能的增强，它专注于对容器对象进行各种非常便利、高效的聚合操作或者大批量数据操作。

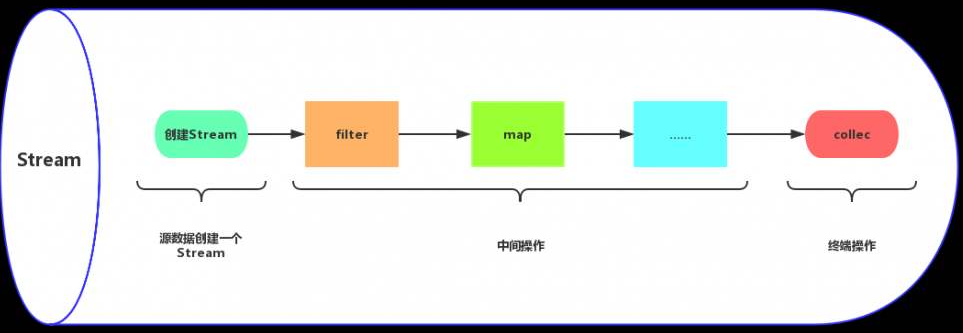
Stream API借助于同样新出现的Lambda表达式，极大的提高编程效率和程序可读性。同时，它提供串行和并行两种模式进行汇聚操作，并发模式能够充分利用多核处理器的优势，使用fork/join并行方式来拆分任务和加速处理过程。所以说，Java8中首次出现的 java.util.stream是一个函数式语言+多核时代综合影响的产物。

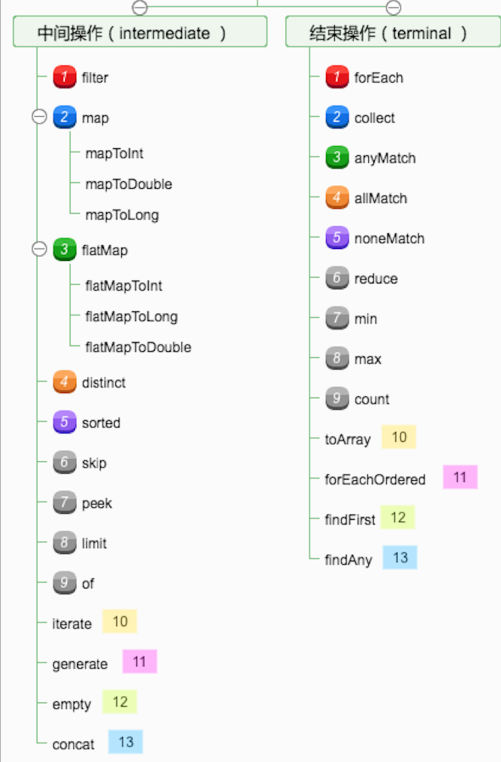
Stream有如下三个操作步骤：

一、创建Stream：从一个数据源，如集合、数组中获取流。

二、中间操作：一个操作的中间链，对数据源的数据进行操作。

三、终止操作：一个终止操作，执行中间操作链，并产生结果。





当数据源中的数据上了流水线后，这个过程对数据进行的所有操作都称为“中间操作”。中间操作仍然会返回一个流对象，因此多个中间操作可以串连起来形成一个流水线。比如map (mapToInt, flatMap 等)、filter、distinct、sorted、peek、limit、skip、parallel、sequential、unordered。

当所有的中间操作完成后，若要将数据从流水线上拿下来，则需要执行终止操作。终止操作将返回一个执行结果，这就是你想要的数据。比如：forEach、forEachOrdered、toArray、reduce、collect、min、max、count、anyMatch、allMatch、noneMatch、findFirst、findAny、iterator。

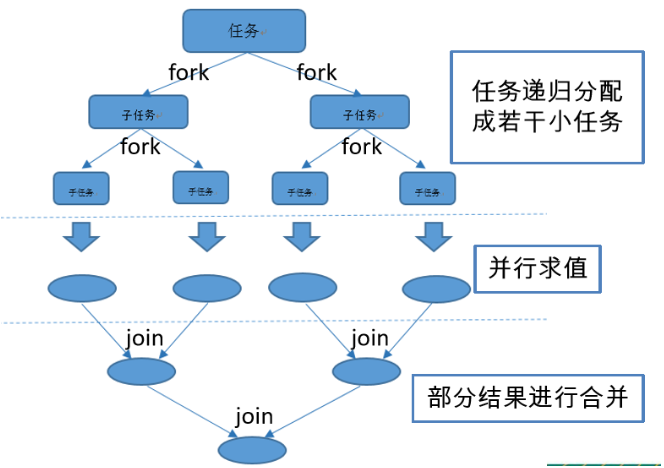
多个中间操作可以连接起来形成一个流水线，除非流水线上触发终止操作，否则中间操作不会执行任何处理！而在终止操作时一次性全部处理，称作“惰性求值”。

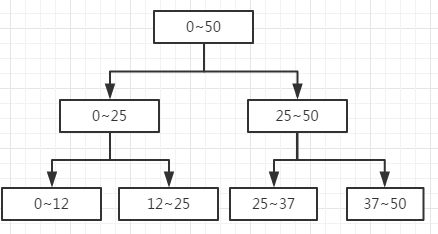
stream并行原理: 其实本质上就是在ForkJoin上进行了一层封装，将Stream 不断尝试分解成更小的split，然后使用fork/join 框架分而治之, parallize使用了默认的ForkJoinPool.common 默认的一个静态线程池.

# 什么是ForkJoin框架 适用场景

虽然目前处理器核心数已经发展到很大数目，但是按任务并发处理并不能完全充分的利用处理器资源，因为一般的应用程序没有那么多的并发处理任务。基于这种现状，考虑把一个任务拆分成多个单元，每个单元分别得到执行，最后合并每个单元的结果。

Fork/Join框架是JAVA7提供的一个用于并行执行任务的框架，是一个把大任务分割成若干小任务，最终汇总每个小任务结果得到大任务结果的框架。



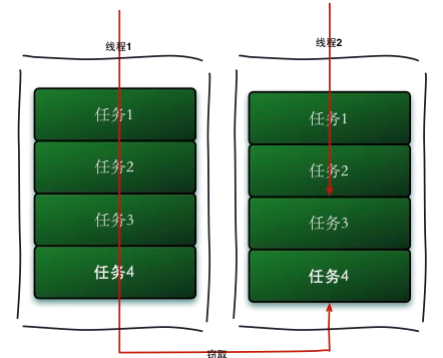


**2.工作窃取算法（work-stealing）**

一个大任务拆分成多个小任务，为了减少线程间的竞争，把这些子任务分别放到不同的队列中，并且每个队列都有单独的线程来执行队列里的任务，线程和队列一一对应。

但是会出现这样一种情况：A线程处理完了自己队列的任务，B线程的队列里还有很多任务要处理。

A是一个很热情的线程，想过去帮忙，但是如果两个线程访问同一个队列，会产生竞争，所以A想了一个办法，从双端队列的尾部拿任务执行。而B线程永远是从双端队列的头部拿任务执行。



注意：线程池中的每个线程都有自己的工作队列（PS，这一点和ThreadPoolExecutor不同，ThreadPoolExecutor是所有线程公用一个工作队列，所有线程都从这个工作队列中取任务），当自己队列中的任务都完成以后，会从其它线程的工作队列中偷一个任务执行，这样可以充分利用资源。

**工作窃取算法的优点：**

利用了线程进行并行计算，减少了线程间的竞争。

**工作窃取算法的缺点：**

任务争夺问题

# Java种的代理有几种实现方式？

动态代理

JDK >>> Proxy

1 面向接口的动态代理 代理一个对象去增强面向某个接口中定义的方法

2 没有接口不可用

3 只能读取到接口上的一些注解

MyBatis

DeptMapper dm=sqlSession.getMapper(DeptMapper.class)

第三方 CGlib

1 面向父类的动态代理

2 有没有接口都可以使用

3 可以读取类上的注解

AOP 日志 性能检测 事务

MyBatis 源码 spring源码

# equals()和==区别。为什么重写equal要重写hashcode？

== 是运算符 equals来自于Object类定义的一个方法

== 可以用于基本数据类型和引用类型

equals只能用于引用类型

== 两端如果是基本数据类型,就是判断值是否相同

equals在重写之后,判断两个对象的属性值是否相同

equals如果不重写,其实就是 ==

重写equals可以让我们自己定义判断两个对象是否相同的条件

Object中定义的hashcode方法生成的哈希码能保证同一个类的对象的哈希码一定是不同的

当equals 返回为true,我们在逻辑上可以认为是同一个对象,但是查看哈希码,发现哈希码不同,和equals方法的返回结果违背

Object中定义的hashcode方法生成的哈希码跟对象的本身属性值是无关的

重写hashcode之后,我们可以自定义哈希码的生成规则,可以通过对象的属性值计算出哈希码

HashMap中,借助equals和hashcode方法来完成数据的存储

将根据对象的内容查询转换为根据索引查询

# hashmap在1.8中做了哪些优化？

**数据结构**

在Java1.7中，HashMap的数据结构为**数组+单向链表**。Java1.8中变成了**数组+单向链表+红黑树**

链表插入节点的方式

在Java1.7中，插入链表节点使用**头插法**。Java1.8中变成了**尾插法。**

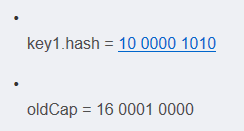
**hash函数**

Java1.8的hash()中，将hash值高位（前16位）参与到取模的运算中，使得计算结果的不确定性增强，降低发生哈希碰撞的概率

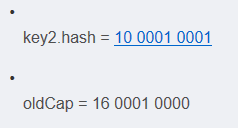
**扩容优化:**

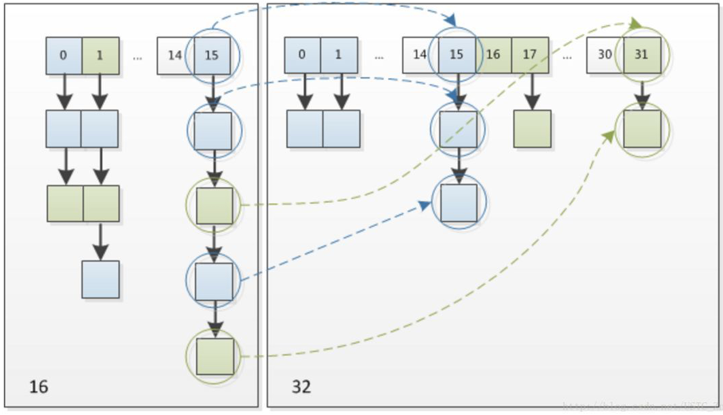
扩容以后,1.7对元素进行rehash算法,计算原来每个元素在扩容之后的哈希表中的位置,1.8借助2倍扩容机制,元素不需要进行重新计算位置

JDK 1.8 在扩容时并没有像 JDK 1.7 那样，重新计算每个元素的哈希值，而是通过高位运算**（e.hash & oldCap）**来确定元素是否需要移动，比如 key1 的信息如下：



使用 e.hash & oldCap 得到的结果，高一位为 0，当结果为 0 时表示元素在扩容时位置不会发生任何变化，而 key 2 信息如下





高一位为 1，当结果为 1 时，表示元素在扩容时位置发生了变化，新的下标位置等于原下标位置 + 原数组长度**hashmap,不必像1.7一样全部重新计算位置**

# hashmap线程安全的方式？

HashMap不是线程安全的,往往在写程序时需要通过一些方法来回避.其实JDK原生的提供了2种方法让HashMap支持线程安全.

方法一:通过Collections.synchronizedMap()返回一个新的Map,这个新的map就是线程安全的. 这个要求大家习惯基于接口编程,因为返回的并不是HashMap,而是一个Map的实现.

方法二:重新改写了HashMap,具体的可以查看java.util.concurrent.ConcurrentHashMap. 这个方法比方法一有了很大的改进.

下面对这2中实现方法从各个角度进行分析和比较.

方法一特点:

通过Collections.synchronizedMap()来封装所有不安全的HashMap的方法,就连toString, hashCode都进行了封装. 封装的关键点有2处,1)使用了经典的synchronized来进行互斥, 2)使用了代理模式new了一个新的类,这个类同样实现了Map接口.在Hashmap上面,synchronized锁住的是对象,所以第一个申请的得到锁,其他线程将进入阻塞,等待唤醒. 优点:代码实现十分简单,一看就懂.缺点:从锁的角度来看,方法一直接使用了锁住方法,基本上是锁住了尽可能大的代码块.性能会比较差.

方法二特点:

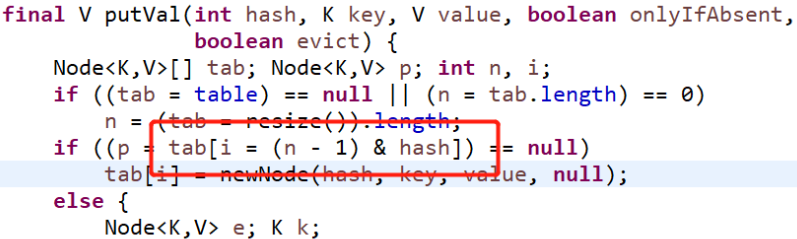
重新写了HashMap,比较大的改变有如下几点.使用了新的锁机制,把HashMap进行了拆分,拆分成了多个独立的块,这样在高并发的情况下减少了锁冲突的可能,使用的是NonfairSync. 这个特性调用CAS指令来确保原子性与互斥性.当如果多个线程恰好操作到同一个segment上面,那么只会有一个线程得到运行.

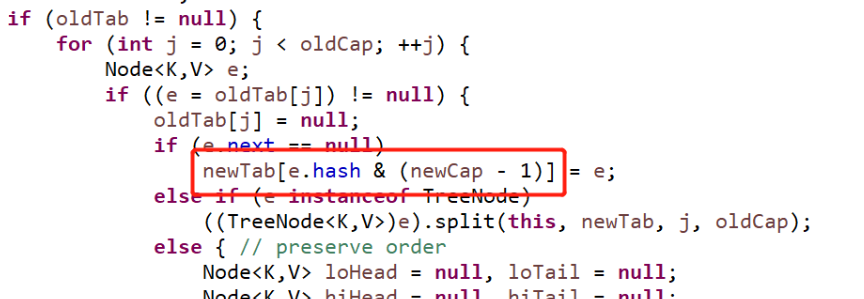
优点:需要互斥的代码段比较少,性能会比较好. ConcurrentHashMap把整个Map切分成了多个块,发生锁碰撞的几率大大降低,性能会比较好. 缺点:代码繁琐

# 为什么hashmap扩容的时候是两倍？

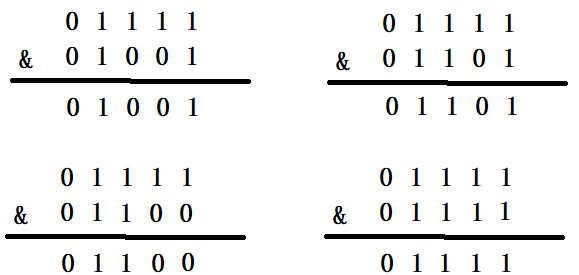
查看源代码

在存入元素时,放入元素位置有一个 (n-1)&hash 的一个算法,和hash&(newCap-1),这里用到了一个&位运算符

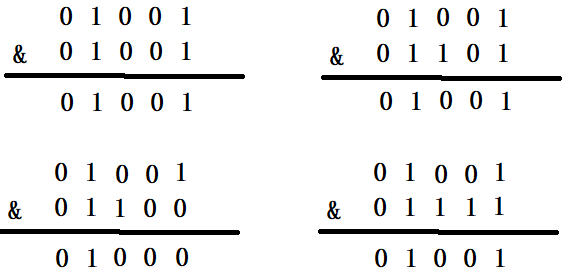




当HashMap的容量是16时，它的二进制是10000，(n-1)的二进制是01111，与hash值得计算结果如下



下面就来看一下HashMap的容量不是2的n次幂的情况，当容量为10时，二进制为01010，(n-1)的二进制是01001，向里面添加同样的元素，结果为



可以看出，有三个不同的元素进过&运算得出了同样的结果，严重的hash碰撞了

# 解决hash冲突的方式有哪些？

1开放定址法

所谓的开放定址法就是一旦发生了冲突，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将记录存入

2再哈希法：

再哈希法又叫双哈希法，有多个不同的Hash函数，当发生冲突时，使用第二个，第三个，….，等哈希函数计算地址，直到无冲突。虽然不易发生聚集，但是增加了计算时间。

3链地址法

链地址法的基本思想是：每个哈希表节点都有一个next指针，多个哈希表节点可以用next指针构成一个单向链表，被分配到同一个索引上的多个节点可以用这个单向 链表连接起来

4建立公共溢出区

这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表

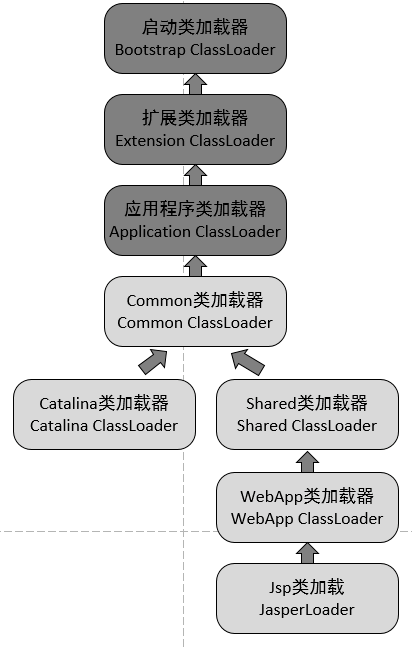
# Tomcat为什么要重写类加载器？

**这里简单解释类加载器双亲委派：**

**无法实现隔离性**：如果使用默认的类加载器机制，那么是无法加载两个相同类库的不同版本的，默认的类加器是不管你是什么版本的，只在乎你的全限定类名，并且只有一份。一个web容器可能要部署两个或者多个应用程序，不同的应用程序，可能会依赖同一个第三方类库的不同版本，因此要保证每一个应用程序的类库都是独立、相互隔离的。部署在同一个web容器中的相同类库的相同版本可以共享，否则，会有重复的类库被加载进JVM, web容器也有自己的类库，不能和应用程序的类库混淆，需要相互隔离

**无法实现热替换：**jsp 文件其实也就是class文件，那么如果修改了，但类名还是一样，类加载器会直接取方法区中已经存在的，修改后的jsp是不会重新加载的。

打破双亲委派机制(参照JVM中的内容)OSGI是基于Java语言的动态模块化规范，类加载器之间是网状结构，更加灵活，但是也更复杂,JNDI服务，使用线程上线文类加载器，父类加载器去使用子类加载器



2. tomcat自己定义的类加载器：

CommonClassLoader：tomcat最基本的类加载器，加载路径中的class可以被tomcat和各个webapp访问

CatalinaClassLoader：tomcat私有的类加载器，webapp不能访问其加载路径下的class，即对webapp不可见

SharedClassLoader：各个webapp共享的类加载器，对tomcat不可见

WebappClassLoader：webapp私有的类加载器，只对当前webapp可见

3. 每一个web应用程序对应一个WebappClassLoader，每一个jsp文件对应一个JspClassLoader，所以这两个类加载器有多个实例

4. 工作原理：

a. CommonClassLoader能加载的类都可以被Catalina ClassLoader和SharedClassLoader使用，从而实现了公有类库的共用

b. CatalinaClassLoader和SharedClassLoader自己能加载的类则与对方相互隔离

c. WebAppClassLoader可以使用SharedClassLoader加载到的类，但各个WebAppClassLoader实例之间相互隔离，多个WebAppClassLoader是同级关系

d. 而JasperLoader的加载范围仅仅是这个JSP文件所编译出来的那一个.Class文件，它出现的目的就是为了被丢弃：当Web容器检测到JSP文件被修改时，会替换掉目前的JasperLoader的实例，并通过再建立一个新的Jsp类加载器来实现JSP文件的HotSwap功能

5. tomcat目录结构，与上面的类加载器对应

/common/\*

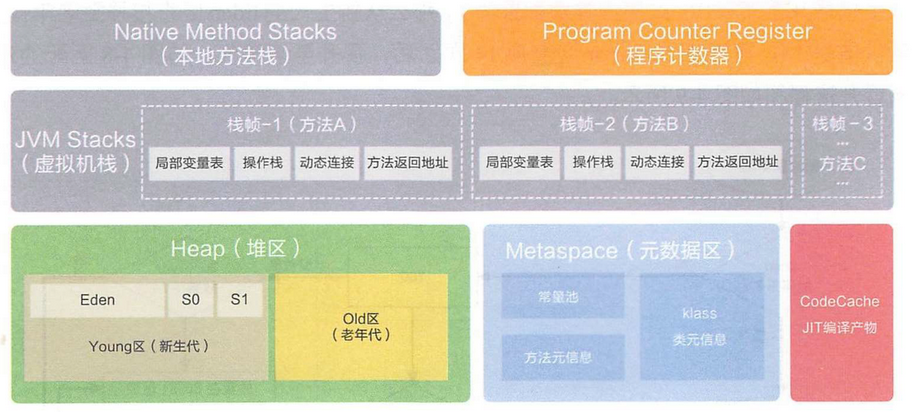
/server/\*

/shared/\*

/WEB-INF/\*

6. 默认情况下，conf目录下的catalina.properties文件，没有指定server.loader以及shared.loader，所以tomcat没有建立CatalinaClassLoader和SharedClassLoader的实例，这两个都会使用CommonClassLoader来代替。Tomcat6之后，把common、shared、server目录合成了一个lib目录。所以在我们的服务器里看不到common、shared、server目录。

# 简述一下Java运行时数据区？



**Java虚拟机栈**

与程序计数器一样，Java 虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。

虚拟机栈描述的是 Java 方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧（Stack Frame，是方法运行时的基础数据结构

**程序计数器**

程序计数器（Program Counter Register）是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。

由于 Java 虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器内核都只会执行一条线程中的指令。

**本地方法栈**

本地方法栈（Native Method Stack）与虚拟机栈所发挥的作用是非常相似的，它们之间的区别不过是虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。Sun HotSpot 虚拟机直接就把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区域也会抛出 StackOverflowError 和 OutOfMemoryError 异常

**Java堆**

对于大多数应用来说，Java 堆（Java Heap）是 Java 虚拟机所管理的内存中最大的一块。Java 堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。

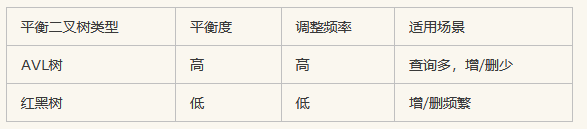
**方法区**

方法区（Method Area）与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

# 说一下反射，反射会影响性能吗？

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。反射这种运行时动态的功能可以说是非常重要的,可以说无反射不框架!!!,反射方式实例化对象和,属性赋值和调用方法肯定比直接的慢,但是程序运行的快慢原因有很多,不能主要归于反射,如果你只是偶尔调用一下反射，反射的影响可以忽略不计,如果你需要大量调用反射,会产生一些影响,适当考虑减少使用或者使用缓存,你的编程的思想才是限制你程序性能的最主要的因素

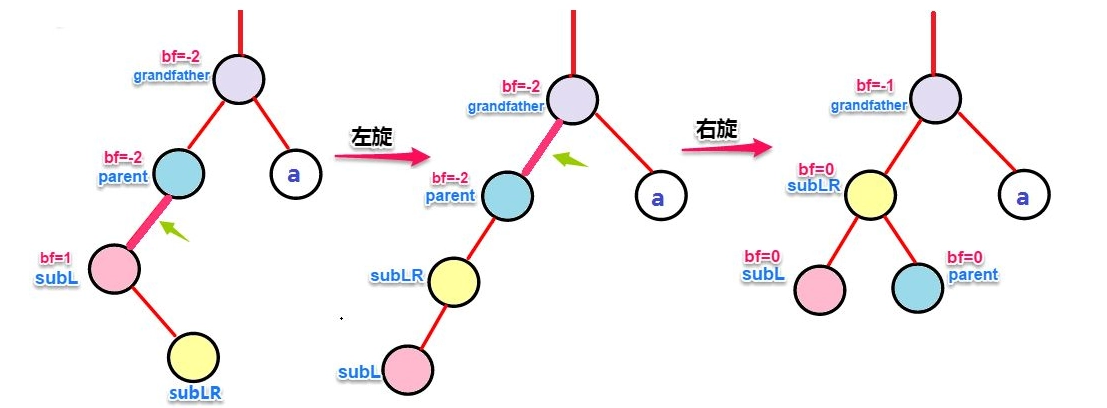
# hashmap为什么用红黑树不用普通的AVL树？



AVL树

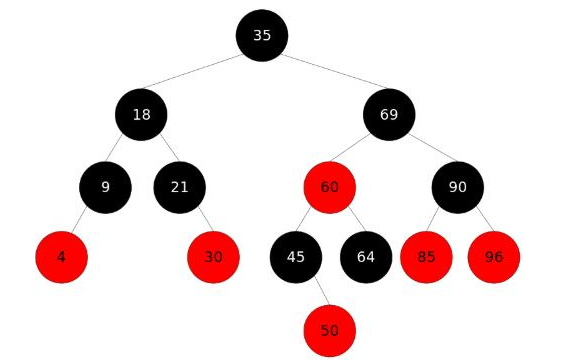
一般用平衡因子判断是否平衡并通过旋转来实现平衡，左右子树树高不超过1，和红黑树相比，AVL树是高度平衡的二叉树，平衡条件必须满足（所有节点的左右子树高度差不超过1）。不管我们是执行插入还是删除操作，只要不满足上面的条件，就要通过旋转来保持平衡，而的由于旋转比较耗时，由此我们可以知道AVL树适合用于插入与删除次数比较少，但查找多的情况

在[计算机科学](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6/9132)中，**AVL树**是最先发明的自平衡二叉查找树。**在AVL树中任何节点的两个子树的高度最大差别为1**，所以它也被称为**高度平衡树**。增加和删除可能需要通过一次或多次[树旋转](https://baike.baidu.com/item/%E6%A0%91%E6%97%8B%E8%BD%AC)来重新平衡这个树



红黑树:

也是一种平衡二叉树，但每个节点有一个存储位表示节点的颜色，可以是红或黑。通过对任何一条从根到叶子的路径上各个节点着色的方式的限制，红黑树确保没有一条路径会比其它路径长出两倍，因此，红黑树是一种弱平衡二叉树**红黑树从根到叶子的最长路径不会超过最短路径的2倍**（由于是弱平衡，可以看到，在相同的节点情况下，AVL树的高度<=红黑树），相对于要求严格的AVL树来说，它的旋转次数少，所以对于搜索，插入，删除操作较多的情况下，用红黑树



# sleep 与 wait 区别

1. 对于 sleep()方法，我们首先要知道该方法是属于 Thread 类中的。而 wait()方法，则是属于 Object 类中的。
2. sleep()方法导致了程序暂停执行指定的时间，让出 cpu,但是他的监控状态依然保持者，当指定的时间到了又会自动恢复运行状态。在调用 sleep()方法的过程中，线程不会释放对象锁。
3. 而当调用 wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用 notify()方法后本线程才进入对象锁定池准备获取对象锁进入运行状态。
4. sleep用Thread调用,在非同步状态下就可以调用, wait用同步监视器调用,必须在同名代码中调用

# synchronized 和 ReentrantLock 的区别

**两者的共同点：**

1. 都是用来协调多线程对共享对象、变量的访问
2. 都是可重入锁，同一线程可以多次获得同一个锁
3. 都保证了可见性和互斥性

**两者的不同点：**

1. ReentrantLock 显示的获得、释放锁，synchronized 隐式获得释放锁
2. ReentrantLock 可响应中断、可轮回，synchronized 是不可以响应中断的，为处理锁的不可用性提供了更高的灵活性
3. ReentrantLock 是 API 级别的，synchronized 是 JVM 级别的
4. ReentrantLock 可以实现公平锁
5. ReentrantLock 通过 Condition 可以绑定多个条件
6. 底层实现不一样， synchronized 是同步阻塞，使用的是悲观并发策略，lock 是同步非阻塞，采用的是乐观并发策略
7. Lock 是一个接口，而 synchronized 是 Java 中的关键字，synchronized 是内置的语言实现。
8. synchronized 在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而 Lock 在发生异常时，如果没有主动通过 unLock()去释放锁，则很可能造成死锁现象，因此使用 Lock 时需要在 finally 块中释放锁。
9. Lock 可以让等待锁的线程响应中断，而 synchronized 却不行，使用 synchronized 时，等待的线程会一直等待下去，不能够响应中断。
10. 通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。
11. Lock 可以提高多个线程进行读操作的效率，既就是实现读写锁等。多个读取线程使用共享锁,写线程使用排它锁/独占

# Condition 类和Object 类锁方法区别

1. Condition 类的 awiat 方法和 Object 类的 wait 方法等效
2. Condition 类的 signal 方法和 Object 类的 notify 方法等效
3. Condition 类的 signalAll 方法和 Object 类的 notifyAll 方法等效
4. ReentrantLock 类可以唤醒指定条件的线程，而 object 的唤醒是随机的

# tryLock和Lock和lockInterruptibly 的区别

1. tryLock 能获得锁就返回 true，不能就立即返回 false，tryLock(long timeout,TimeUnit unit)，可以增加时间限制，如果超过该时间段还没获得锁，返回 false
2. lock 能获得锁就返回 true，不能的话一直等待获得锁
3. lock 和 lockInterruptibly，如果两个线程分别执行这两个方法，但此时中断这两个线程， lock 不会抛出异常，而 lockInterruptibly 会抛出异常。

# 单例模式有哪些实现方式,有什么优缺点

一个教室里面有很多同学,每个同学都要有自己的一个水杯.教室里还有一个饮水机,一个饮水机可以为教室内所有的同学提供用水,没有必要每个同学都准备一个饮水机.程序中往往一个类只需要一个对象就可以为整个系统服务,如果产生多个对象,消耗更多的资源.单例模式就是为了实现如何控制一个类只能产生一个对象. 单例模式控制控制对象不要反复创建,提高我们工作的效率.减少资源的占用

单例模式下类的组成部分

1私有的构造方法

2私有的当前类对象作为静态属性

3公有的向外界提供当前类对象的静态方法

但凡是控制一个类只能产生一个对象的模式都叫做单例模式,常见的有饿汉式,懒汉式,内部类式(接口/抽象类),静态内部类式 ... ...

## 饿汉式代码实现

|  |
| --- |
| */\* 多例 只要调用了构造方法 就会在内存上产生一个独立的空间 1将构造方法私有化 构造方法私有化了,外界不能new对象了?对象怎么产生? 2组合当前类本身作为私有静态属性并调用构造方法实例化 如何让外界获取属性值呢? 3在当前类中准备一个共有的静态方法向外界提供当前类对象  \*/* **public class** SingleTon {  **private static** SingleTon *singleTon* =**new** SingleTon();  **private** SingleTon(){  }  **public static** SingleTon getSingleTon(){  **return** *singleTon*;  } } **class** Test{  **public static void** main(String[] args) {  SingleTon st =SingleTon.*getSingleTon*();  SingleTon st2=SingleTon.*getSingleTon*();  System.***out***.println(st==st2);  System.***out***.println(st);  System.***out***.println(st2);  } } |

好处: 饿汉式单例模式在类加载进入内存初始化static变量是会初始化当前类对象,此时也不会涉及多个线程对象访问该对象的问题。虚拟机保证只会装载一次该类，肯定不会发生并发访问的问题。因此，可以省略synchronized关键字。

问题：如果只是加载本类，而不是要调用getInstance()，甚至永远没有调用，则会造成资源浪费,不能延迟加载！

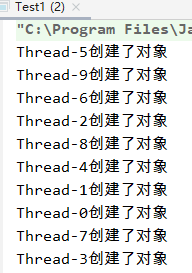
## 懒汉式单例模式

|  |
| --- |
| */\* 多例 只要调用了构造方法 就会在内存上产生一个独立的空间 1将构造方法私有化 构造方法私有化了,外界不能new对象了?对象怎么产生? 2组合当前类本身作为私有静态属性并调用构造方法实例化 如何让外界获取属性值呢? 3在当前类中准备一个共有的静态方法向外界提供当前类对象  \*/* **public class** SingleTon {  **private static** SingleTon *singleTon*;  **private** SingleTon(){  }  **public static** SingleTon getSingleTon(){  **if**(**null** == *singleTon*){  *singleTon*=**new** SingleTon();  }  **return** *singleTon*;  } } **class** Test{  **public static void** main(String[] args) {  SingleTon st =SingleTon.*getSingleTon*();  SingleTon st2=SingleTon.*getSingleTon*();  System.***out***.println(st==st2);  System.***out***.println(st);  System.***out***.println(st2);  } } |

延迟加载,也叫作懒加载,等到真正用的时候才加载.

懒汉式代理模式在多线程并发情况下仍然是有可能创建多次,是线程非安全的

|  |
| --- |
| **public class** Test1 {  **public static void** main(String[] args) {  **for** (**int** i = 0; i < 10; i++) {  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  SingleTon.*getSingleTon*();  }  }).start();  }  } } **class** SingleTon {  **private static** SingleTon *singleTon*;  **private** SingleTon(){  System.***out***.println(Thread.*currentThread*().getName()+**"创建了对象"**);  }  **public static** SingleTon getSingleTon(){  **if**(**null** == *singleTon*){  *singleTon*=**new** SingleTon();  }  **return** *singleTon*;  } } |



## 双重检测式单例模式

为了解决线程并发问题我们需要对其进行优化,作为一个双重检测式的单例模式,就是我们说的DCL单例模式

|  |
| --- |
| **package** com.msb.singleTon;  **public class** Test1 {  **public static void** main(String[] args) {  **for** (**int** i = 0; i < 10; i++) {  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  SingleTon.*getSingleTon*();  }  }).start();  }  } } **class** SingleTon {  **private volatile static** SingleTon *singleTon*;  **private** SingleTon(){  System.***out***.println(Thread.*currentThread*().getName()+**"创建了对象"**);  }  **public static** SingleTon getSingleTon(){  **if**(**null** ==*singleTon*){  **synchronized** (SingleTon.**class**){  **if**(**null** == *singleTon*){  *singleTon*=**new** SingleTon();  */\*\*  \* 1分配空间  \* 2执行构造方法  \* 3将创建对象的引用地址赋值值singleTon变量  \* 为了避免多线程下的指令重拍问题和多线程缓存造成的数据更新不及时问题  \* 我们应该在加上volatile处理  \*/* }  }  }  **return** *singleTon*;  } } |

## 静态内部类单例模式

除此之外,我们还可以使用内部类实现单例模式的控制

|  |
| --- |
| **class** Single{  */\*  \* 私有构造方法  \* \*/* **private** Single(){   }  */\*  \* 范围内部类的属性  \* \*/* **public static** Single getSingle(){  **return** InnerClass.*single*;  }  */\*  \* 静态内部类  \* \*/* **public static class** InnerClass{  */\*  \* 组合外部类对象作为属性  \* \*/* **private static final** Single *single*=**new** Single();  } } |

外部类没有static属性，则不会像饿汉式那样立即加载对象,只有真正调用getInstance(),才会加载静态内部类。加载类时是线程安全的。 instance是static final 类型，保证了内存中只有这样一个实例存在，而且只能被赋值一次，从而保证了线程安全性.兼备了并发高效调用和延迟加载的优势

## 枚举式单例模式

|  |
| --- |
| **public class** Test3 {  **public static void** main(String[] args) {  SingleTon1 s1=SingleTon1.***INSTANCE***;  SingleTon1 s2=SingleTon1.***INSTANCE***;  s1.singleTonOperation();  System.***out***.println(s1==s2);  } }  **enum** SingleTon1{  ***INSTANCE***;  **public void** singleTonOperation(){  System.***out***.println(**"operation"**);  } } |

优点：实现简单,枚举本身就是单例模式。由JVM从根本上提供保障！避免通过反射和反序列化的漏洞！

缺点:无延迟加载

单例模式总结:

单例模式主要的两种实现方式

饿汉式 线程安全,调用效率高,不能延时加载

懒汉式 线程安全,调用效率不高,可以延时加载

其他方式

双重检测锁式 极端情况下偶尔会出现问题,不建议使用

静态内部类式 线程安全,调用效率高,可以延时加载

枚举式 线程安全,调用效率高,不能延时加载

