# 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

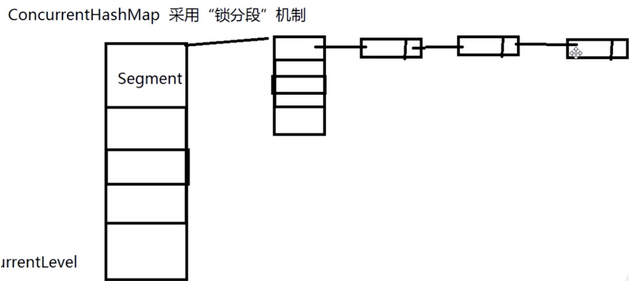
 2.ConcurrentHashMap和HashTable的区别

 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。

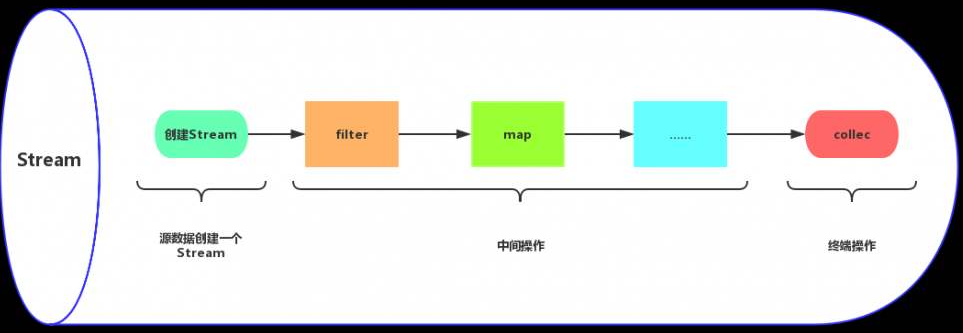
Stream API借助于同样新出现的Lambda表达式，极大的提高编程效率和程序可读性。同时，它提供串行和并行两种模式进行汇聚操作，并发模式能够充分利用多核处理器的优势，使用fork/join并行方式来拆分任务和加速处理过程。所以说，Java8中首次出现的 java.util.stream是一个函数式语言+多核时代综合影响的产物。

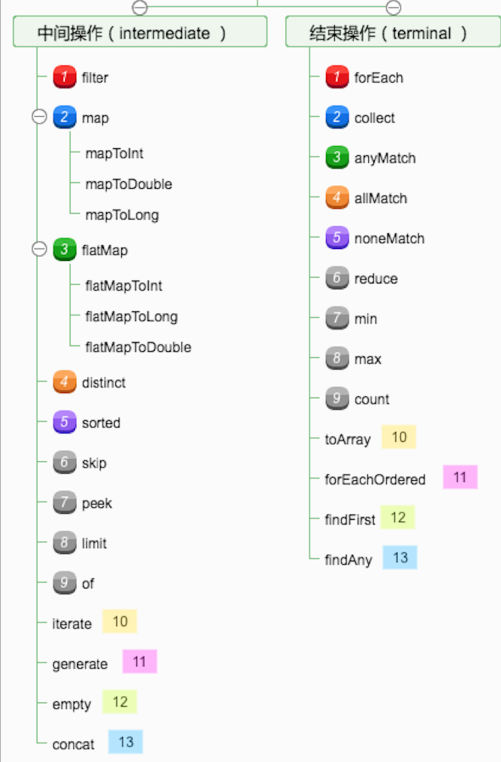
Stream有如下三个操作步骤：

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。

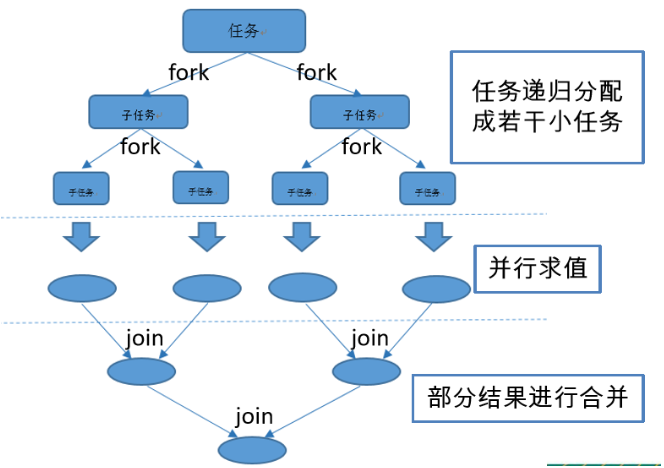
多个中间操作可以连接起来形成一个流水线，除非流水线上触发终止操作，否则中间操作不会执行任何处理！而在终止操作时一次性全部处理，称作“惰性求值”。

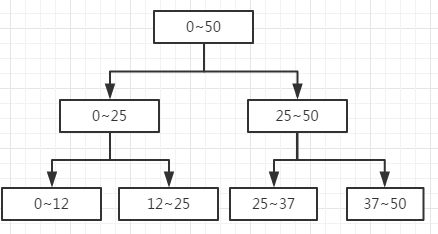
stream并行原理: 其实本质上就是在ForkJoin上进行了一层封装，将Stream 不断尝试分解成更小的split，然后使用fork/join 框架分而治之, parallize使用了默认的ForkJoinPool.common 默认的一个静态线程池.

# 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。



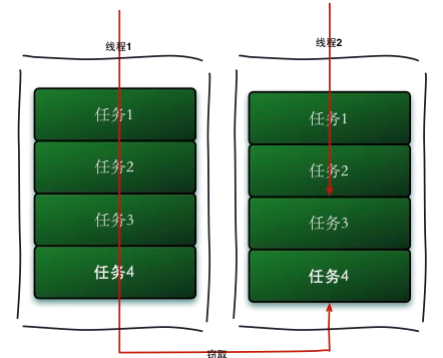


**2.工作窃取算法（work-stealing）**

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。**

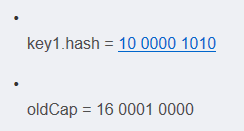
**hash函数**

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

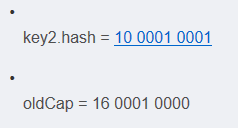
**扩容优化:**

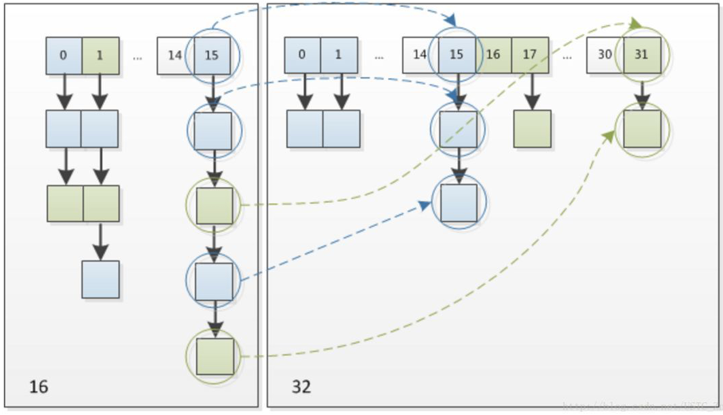
扩容以后,1.7对元素进行rehash算法,计算原来每个元素在扩容之后的哈希表中的位置,1.8借助2倍扩容机制,元素不需要进行重新计算位置

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:

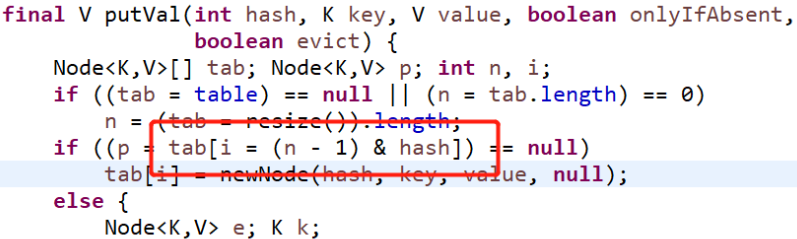
重新写了HashMap,比较大的改变有如下几点.使用了新的锁机制,把HashMap进行了拆分,拆分成了多个独立的块,这样在高并发的情况下减少了锁冲突的可能,使用的是NonfairSync. 这个特性调用CAS指令来确保原子性与互斥性.当如果多个线程恰好操作到同一个segment上面,那么只会有一个线程得到运行.

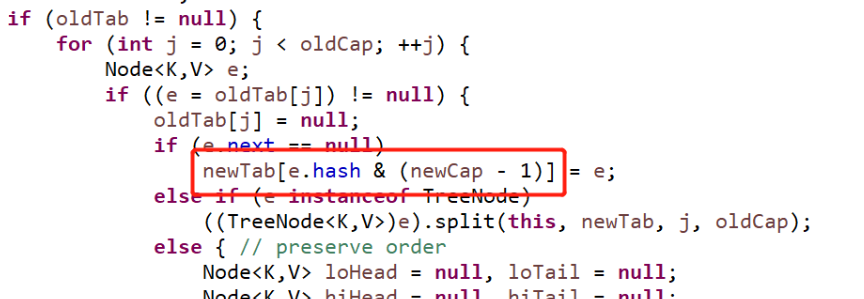
优点:需要互斥的代码段比较少,性能会比较好. ConcurrentHashMap把整个Map切分成了多个块,发生锁碰撞的几率大大降低,性能会比较好. 缺点:代码繁琐

# 为什么hashmap扩容的时候是两倍？

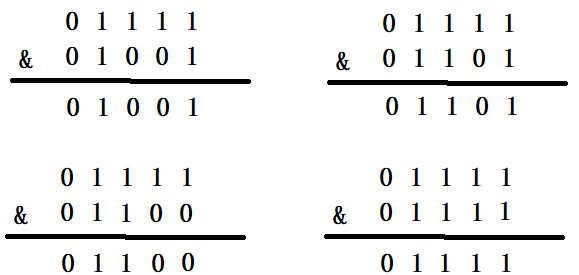
查看源代码

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

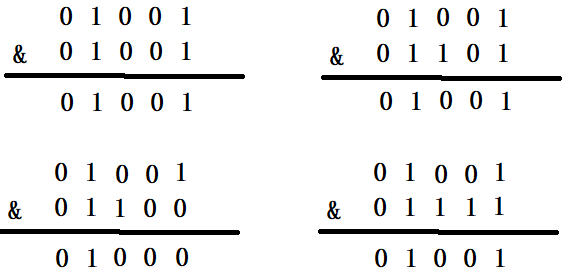




当HashMap的容量是16时，它的二进制是10000，(n-1)的二进制是01111，与hash值得计算结果如下



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

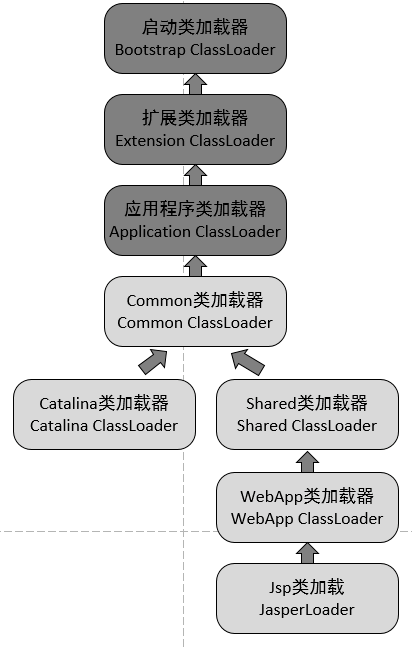
# Tomcat为什么要重写类加载器？

**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/\*

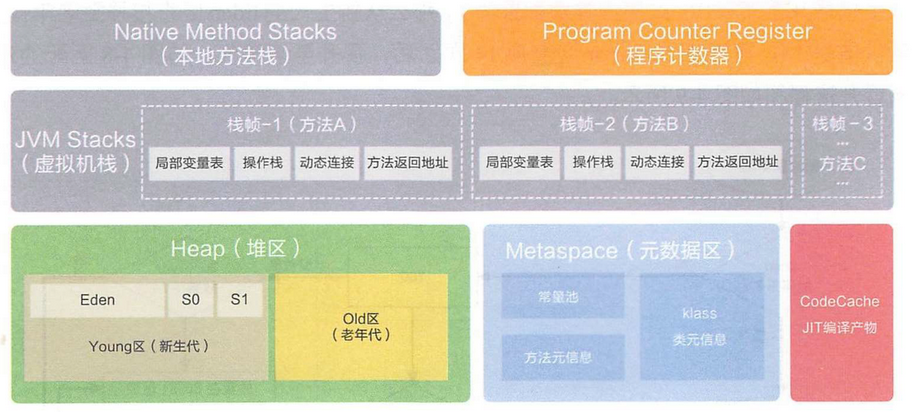
/server/\*

/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

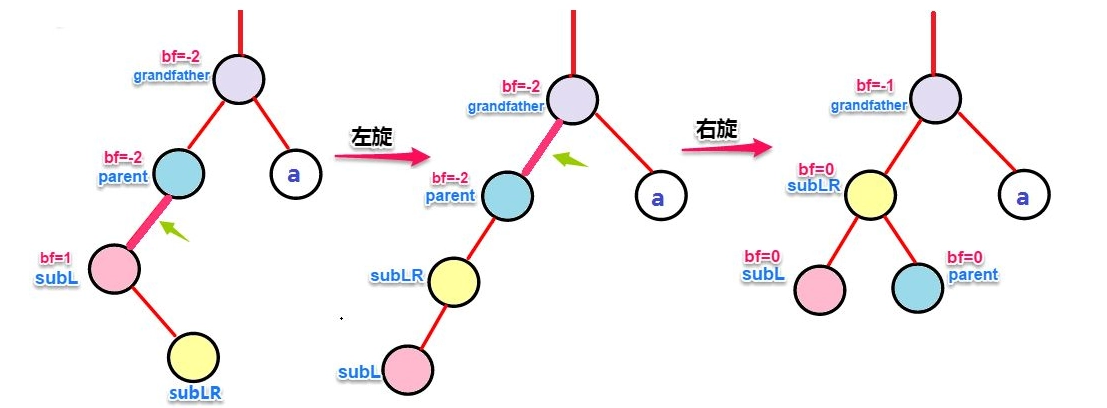
# hashmap为什么用红黑树不用普通的AVL树？



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！

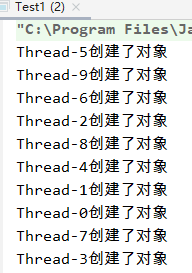
## 懒汉式单例模式

|  |
| --- |
| */\* 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



# 关于intern

String a=**new** String(**"123"**)+**new** String(**"456"**);  
//String b=**new** String(**"123456"**);  
String intern = a.intern();  
System.***out***.println(intern==a);

注释输出true,取消注释 输出false

<https://blog.csdn.net/qq_41884976/article/details/83353389>

# BIO,NIO,AIO 有什么区别?

* BIO：Block IO 同步阻塞式 IO，就是我们平常使用的传统 IO，它的特点是模式简单使用方便，并发处理能力低。
* NIO：Non IO 同步非阻塞 IO，是传统 IO 的升级，客户端和服务器端通过 Channel（通道）通讯，实现了多路复用。
* AIO：Asynchronous IO 是 NIO 的升级，也叫 NIO2，实现了异步非堵塞 IO ，异步 IO 的操作基于事件和回调机制。

详细回答

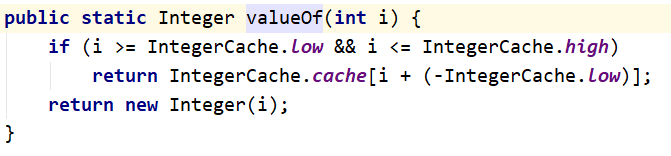
* BIO (Blocking I/O): 同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。
* NIO (New I/O): NIO是一种同步非阻塞的I/O模型，在Java 1.4 中引入了NIO框架，对应 java.nio 包，提供了 Channel , Selector，Buffer等抽象。NIO中的 N可以理解为Non-blocking，不单纯是New。它支持面向缓冲的，基于通道的I/O操作方法。 NIO提供了与传统BIO模型中的 Socket 和 ServerSocket 相对应的 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现,两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞I/O来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发
* AIO (Asynchronous I/O): AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的IO模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步IO的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO操作本身是同步的。就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

# 包装类缓存问题

Integer i =128;  
Integer i2=128;  
System.***out***.println(i==i2);

问题 i i2 = 127 结果是true 128 结果为false

valueOf方法源码



数值在缓存之内,返回缓存中的对象,如果不在返回新的对象

|  |
| --- |
| **private static class** IntegerCache {  **static final int *low*** = -128;  **static final int *high***;  **static final** Integer ***cache***[];   **static** {  *// high value may be configured by property* **int** h = 127;  String integerCacheHighPropValue =  sun.misc.VM.*getSavedProperty*(**"java.lang.Integer.IntegerCache.high"**);  **if** (integerCacheHighPropValue != **null**) {  **try** {  **int** i = *parseInt*(integerCacheHighPropValue);  i = Math.*max*(i, 127);  *// Maximum array size is Integer.MAX\_VALUE* h = Math.*min*(i, Integer.***MAX\_VALUE*** - (-***low***) -1);  } **catch**( NumberFormatException nfe) {  *// If the property cannot be parsed into an int, ignore it.* }  }  ***high*** = h;   ***cache*** = **new** Integer[(***high*** - ***low***) + 1];  **int** j = ***low***;  **for**(**int** k = 0; k < ***cache***.**length**; k++)  ***cache***[k] = **new** Integer(j++);   *// range [-128, 127] must be interned (JLS7 5.1.7)* **assert** IntegerCache.***high*** >= 127;  }   **private** IntegerCache() {} } |

还有哪些包装类有缓存

Byte short Long Integer Character 都有缓存

具体细节,翻阅源代码

# 简述线程生命周期(状态)

当线程被创建并启动以后，它既不是一启动就进入了执行状态，也不是一直处于执行状态。在线程的生命周期中，它要经过新建(New)、就绪（Runnable）、运行（Running）、阻塞(Blocked)和死亡(Dead)5 种状态。尤其是当线程启动以后，它不可能一直"霸占"着 CPU 独自运行，所以 CPU 需要在多条线程之间切换，于是线程状态也会多次在运行、阻塞之间切换

**新建状态（NEW）**

当程序使用 new 关键字创建了一个线程之后，该线程就处于新建状态，此时仅由 JVM 为其分配内存，并初始化其成员变量的值

**就绪状态（RUNNABLE）：**

当线程对象调用了 start()方法之后，该线程处于就绪状态。Java 虚拟机会为其创建方法调用栈和程序计数器，等待调度运行。

**运行状态（Running）**

线程运行,正在占用CUP运行的一个状态

**阻塞状态（BLOCKED）：**

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

态。阻塞的情况分三种：

**等待阻塞（o.wait等待对列）：**

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

**同步阻塞(lock)**

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

**其他阻塞(sleep/join)**

运行(running)的线程执行 Thread.sleep(long ms)或 t.join()方法，或者发出了 I/O 请求时，JVM 会把该线程置为阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O处理完毕时，线程重新转入可运行(runnable)状态。

**线程死亡（DEAD）**

线程会以下面三种方式结束，结束后就是死亡状态。

正常结束

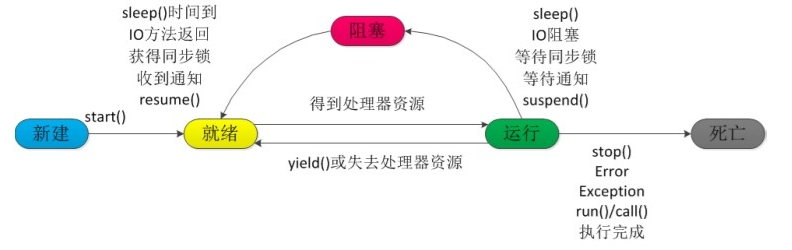
1. run()或 call()方法执行完成，线程正常结束。

异常结束

1. 线程抛出一个未捕获的 Exception 或 Error。

调用stop

1. 直接调用该线程的 stop()方法来结束该线程—该方法通常容易导致死锁，不推荐使用。



# 终止线程 4 种方式

## 正常运行结束

程序运行结束，线程自动结束。

## 使用退出标志退出线程

一般 run()方法执行完，线程就会正常结束，然而，常常有些线程是伺服线程。它们需要长时间的

运行，只有在外部某些条件满足的情况下，才能关闭这些线程。使用一个变量来控制循环，例如：

最直接的方法就是设一个 boolean 类型的标志，并通过设置这个标志为 true 或 false 来控制 while循环是否退出，代码示例：

public class ThreadSafe extends Thread {  
public volatile boolean exit = false;  
public void run() { while (!exit){  
//do something  
}  
}  
}

定义了一个退出标志 exit，当 exit 为 true 时，while 循环退出，exit 的默认值为 false.在定义 exit时，使用了一个 Java 关键字 volatile，这个关键字的目的是使 exit 同步，也就是说在同一时刻只能由一个线程来修改 exit 的值。

## Interrupt 方法结束线程

使用 interrupt()方法来中断线程有两种情况：

1线程处于阻塞状态：如使用了 sleep,同步锁的 wait,socket 中的 receive,accept 等方法时，会使线程处于阻塞状态。当调用线程的 interrupt()方法时，会抛出 InterruptException 异常。阻塞中的那个方法抛出这个异常，通过代码捕获该异常，然后 break 跳出循环状态，从而让我们有机会结束这个线程的执行。通常很多人认为只要调用 interrupt 方法线程就会结束，实际上是错的， 一定要先捕获 InterruptedException 异常之后通过 break 来跳出循环，才能正常结束 run 方法。

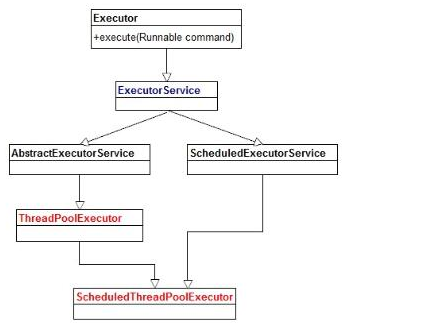
2线程未处于阻塞状态：使用 isInterrupted()判断线程的中断标志来退出循环。当使用interrupt()方法时，中断标志就会置 true，和使用自定义的标志来控制循环是一样的道理。

public class ThreadSafe extends Thread {  
public void run() {  
while (!isInterrupted()){ //非阻塞过程中通过判断中断标志来退出  
try{  
Thread.sleep(51000);//阻塞过程捕获中断异常来退出  
}catch(InterruptedException e){  
e.printStackTrace();  
break;//捕获到异常之后，执行 break 跳出循环  
}  
}  
}  
}

## stop 方法终止线程（线程不安全）

程序中可以直接使用 thread.stop()来强行终止线程，但是 stop 方法是很危险的，就象突然关闭计算机电源，而不是按正常程序关机一样，可能会产生不可预料的结果，不安全主要是：thread.stop()调用之后，创建子线程的线程就会抛出 ThreadDeatherror 的错误，并且会释放子线程所持有的所有锁。一般任何进行加锁的代码块，都是为了保护数据的一致性，如果在调用thread.stop()后导致了该线程所持有的所有锁的突然释放(不可控制)，那么被保护数据就有可能呈现不一致性，其他线程在使用这些被破坏的数据时，有可能导致一些很奇怪的应用程序错误。因此，并不推荐使用 stop 方法来终止线程。

# 线程池用过没?如何获得一个线程池?各个参数的含义?



new Thread 直接就是创建一个线程对象 线程的任务和线程对象合并到一起

new Runnable new Callable 创建一个任务对象 将任务对象放到一个线程对象中去执行 将任务和线程对象分开

## 线程池的好处

* 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。
* 提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。
* 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。

1）核心参数

public ThreadPoolExecutor(int corePoolSize, // 核心线程数量大小     
int maximumPoolSize, // 线程池最大容纳线程数  
long keepAliveTime, // 线程空闲后的存活时长  
TimeUnit unit,  
//缓存异步任务的队列 //用来构造线程池里的worker线程  
BlockingQueue<Runnable> workQueue,  
ThreadFactory threadFactory,  
//线程池任务满载后采取的任务拒绝策略   
RejectedExecutionHandler handler)

* corePoolSize

线程池的核心线程数。在没有设置 allowCoreThreadTimeOut 为 true 的情况下，核心线程会在线程池中一直存活，即使处于闲置状态。

maximumPoolSize

线程池所能容纳的最大线程数。当活动线程(核心线程+非核心线程)达到这个数值后，后续任务将会根据 RejectedExecutionHandler 来进行拒绝策略处理。

* keepAliveTime

非核心线程闲置时的超时时长。超过该时长，**非核心线程**就会被回收。若线程池通设置核心线程也允许 timeOut，即 allowCoreThreadTimeOut 为 true，则该时长同样会作用于核心线程，在超过 aliveTime 时，核心线程也会被回收，AsyncTask 配置的线程池就是这样设置的。

* unit

keepAliveTime 时长对应的单位。Timeunit

* workQueue

线程池中的任务队列，通过线程池的 execute() 方法提交的 Runnable 对象会存储在该队列中。

* ThreadFactory

线程工厂，功能很简单，就是为线程池提供创建新线程的功能。这是一个接口，可以通过自定义，做一些自定义线程名的操作。

* RejectedExecutionHandler

当任务无法被执行时(超过线程最大容量 maximum 并且 workQueue 已经被排满了)的处理策略，这里有四种任务拒绝类型。

## 线程池工作原则

* 1、当线程池中线程数量小于 corePoolSize 则创建线程，并处理请求。
* 2、当线程池中线程数量大于等于 corePoolSize 时，则把请求放入 workQueue 中,随着线程池中的核心线程们不断执行任务，只要线程池中有空闲的核心线程，线程池就从 workQueue 中取任务并处理。
* 3 、当 workQueue 已存满，放不下新任务时则新建非核心线程入池，并处理请求直到线程数目达到 maximumPoolSize（最大线程数量设置值）。
* 4、如果线程池中线程数大于 maximumPoolSize 则使用 RejectedExecutionHandler 来进行任务拒绝处理。

## 任务队列 BlockingQueue

任务队列 workQueue 是用于存放不能被及时处理掉的任务的一个队列，它是 一个 BlockingQueue 类型。

关于 BlockingQueue，虽然它是 Queue 的子接口，但是它的主要作用并不是容器，而是作为线程同步的工具，他有一个特征，当生产者试图向 BlockingQueue 放入(put)元素，如果队列已满，则该线程被阻塞；当消费者试图从 BlockingQueue 取出(take)元素，如果队列已空，则该线程被阻塞。(From 疯狂Java讲义)

## 任务拒绝类型

* ThreadPoolExecutor.AbortPolicy:

当线程池中的数量等于最大线程数时抛 java.util.concurrent.RejectedExecutionException 异常，涉及到该异常的任务也不会被执行，线程池默认的拒绝策略就是该策略。

* ThreadPoolExecutor.DiscardPolicy():

当线程池中的数量等于最大线程数时,默默丢弃不能执行的新加任务，不报任何异常。

* ThreadPoolExecutor.CallerRunsPolicy():

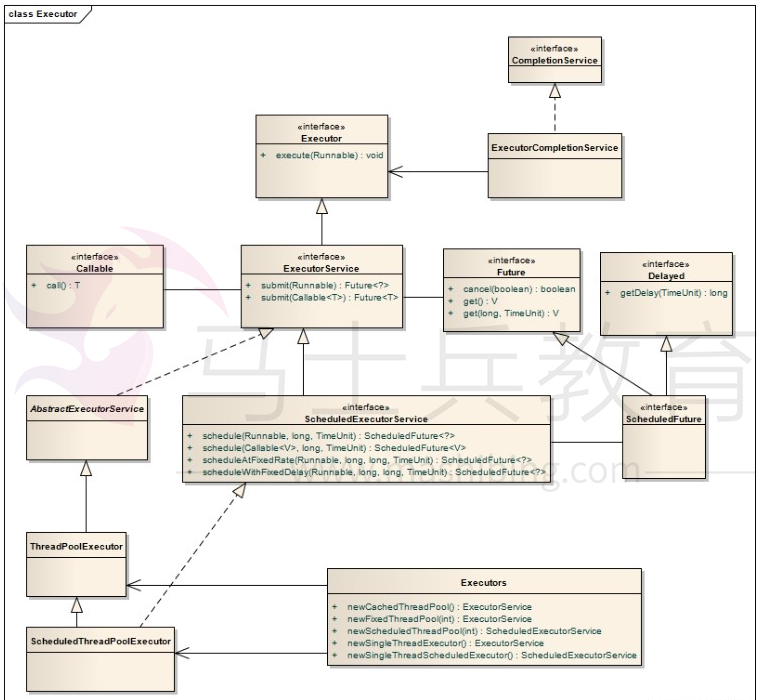
当线程池中的数量等于最大线程数时，重试添加当前的任务；它会自动重复调用execute()方法。

* ThreadPoolExecutor.DiscardOldestPolicy():

当线程池中的数量等于最大线程数时,抛弃线程池中工作队列头部的任务(即等待时间最久的任务)，并执行新传入的任务。

## 4种线程池

Java 里面线程池的顶级接口是 **Executor**，但是严格意义上讲 Executor 并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是 **ExecutorService**。



**Executors 线程池工具类给我们提供了一些API,用于返回一些不同的线程池**

**newCachedThreadPool**

创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。对于执行很多短期异步任务的程序而言，这些线程池通常可提高程序性能。调用 execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。因此，长时间保持空闲的线程池不会使用任何资源。

**newFixedThreadPool**

创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程。在任意点，在大多数 nThreads 线程会处于处理任务的活动状态。如果在所有线程处于活动状态时提交附加任务，则在有可用线程之前，附加任务将在队列中等待。如果在关闭前的执行期间由于失败而导致任何线程终止，那么一个新线程将代替它执行后续的任务（如果需要）。在某个线程被显式地关闭之前，池中的线程将一直存在。

**newScheduledThreadPool**

创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

\ScheduledExecutorService scheduledThreadPool= Executors.newScheduledThreadPool(3);  
scheduledThreadPool.schedule(newRunnable(){  
@Override  
public void run() { System.out.println("延迟三秒"); }  
}, 3, TimeUnit.SECONDS);  
scheduledThreadPool.scheduleAtFixedRate(newRunnable(){  
@Override  
public void run() {  
System.out.println("延迟 1 秒后每三秒执行一次");  
}  
},1,3,TimeUnit.SECONDS);

**newSingleThreadExecutor**

Executors.newSingleThreadExecutor()返回一个线程池（这个线程池只有一个线程）,这个线程池可以在线程死后（或发生异常时）重新启动一个线程来替代原来的线程继续执行下去！

# 为什么要把堆和栈区分出来呢？栈中不是也可以存储数据吗？

## 栈是运行时的单位，而堆是存储的单位。

栈解决程序的运行问题，即程序如何执行，或者说如何处理数据；堆解决的是数据存储的问题，即数据怎么放、放在哪儿。

在Java中一个线程就会相应有一个线程栈与之对应，这点很容易理解，因为不同的线程执行逻辑有所不同，因此需要一个独立的线程栈。而堆则是所有线程共享的。栈因为是运行单位，因此里面存储的信息都是跟当前线程（或程序）相关信息的。包括局部变量、程序运行状态、方法返回值等等；而堆只负责存储对象信息。

1. 从软件设计的角度看，栈代表了处理逻辑，而堆代表了数据。这样分开，使得处理逻辑更为清晰。分而治之的思想。这种隔离、模块化的思想在软件设计的方方面面都有体现。
2. 堆与栈的分离，使得堆中的内容可以被多个栈共享（也可以理解为多个线程访问同一个对象）。这种共享的收益是很多的。一方面这种共享提供了一种有效的数据交互方式(如：共享内存)，另一方面，堆中的共享常量和缓存可以被所有栈访问，节省了空间。
3. 栈因为运行时的需要，比如保存系统运行的上下文，需要进行地址段的划分。由于栈只能向上增长，因此就会限制住栈存储内容的能力。而堆不同，堆中的对象是可以根据需要动态增长的，因此栈和堆的拆分，使得动态增长成为可能，相应栈中只需记录堆中的一个地址即可。
4. 面向对象就是堆和栈的完美结合。其实，面向对象方式的程序与以前结构化的程序在执行上没有任何区别。但是，面向对象的引入，使得对待问题的思考方式发生了改变，而更接近于自然方式的思考。当我们把对象拆开，你会发现，对象的属性其实就是数据，存放在堆中；而对象的行为（方法），就是运行逻辑，放在栈中。我们在编写对象的时候，其实即编写了数据结构，也编写的处理数据的逻辑。不得不承认，面向对象的设计，确实很美。

# 为什么不把基本类型放堆中呢？

因为其占用的空间一般是1~8个字节——需要空间比较少，而且因为是基本类型，所以不会出现动态增长的情况——长度固定，因此栈中存储就够了，如果把他存在堆中是没有什么意义的。可以这么说，基本类型和对象的引用都是存放在栈中，而且都是几个字节的一个数，因此在程序运行时，他们的处理方式是统一的。但是基本类型、对象引用和对象本身就有所区别了，因为一个是栈中的数据一个是堆中的数据。最常见的一个问题就是，Java中参数传递时的问题。

# 堆中存什么？栈中存什么？

堆中存的是对象。栈中存的是基本数据类型和堆中对象的引用。一个对象的大小是不可估计的，或者说是可以动态变化的，但是在栈中，一个对象只对应了一个4btye的引用（堆栈分离的好处：）。

# Java中的参数传递时传值呢？还是传引用？

|  |
| --- |
| */\*\*  \** ***@Author:*** *Ma HaiYang  \** ***@Description:*** *MircoMessage:Mark\_7001  \*/* **public class** Test1 {  **public static void** main(String[] args) {  */\*int[] arr ={0};  change2(arr);  System.out.println(arr[0]);\*/   /\* String s ="bbb";  changeStr(s);  System.out.println(s);\*/* Integer i =10;  *changeInteger*(i);  System.***out***.println(i);  }  **public static void** changeInteger(Integer i){  i=100;  }    **public static void** changeStr(String s){  s=**"aaaa"**;  }    **public static void** change(**int**[] arr){  arr[0]=100;  }  **public static void** change2(**int**[] arr){  arr=**new int**[]{100};  }    } |

参数是基本数据类型 按值传递

参数是引用类型 按引用传递

按引用传递在方法体重修改形参时,可能会对实参产生影响, 注意坑 修改的方式,String 包装类诶

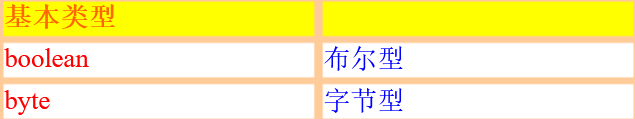
# JAVA有哪几种关键字,作用区间是什么

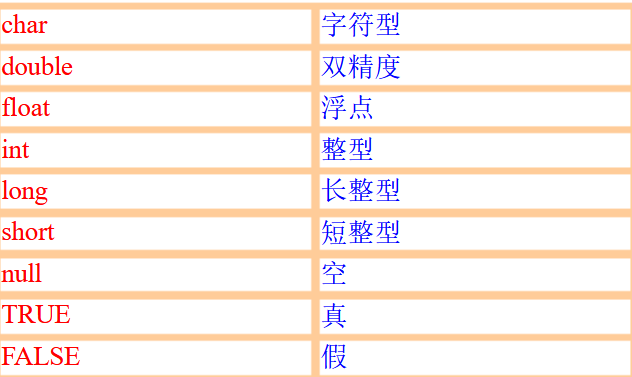


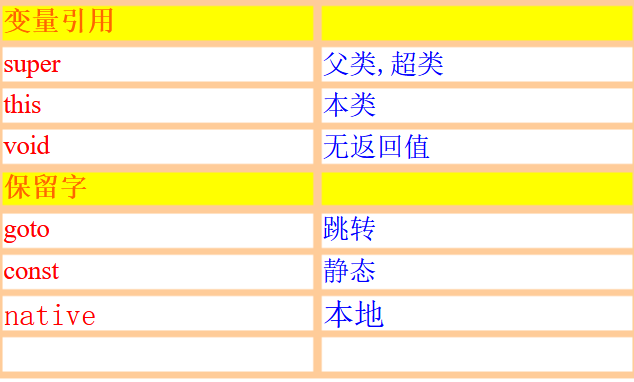














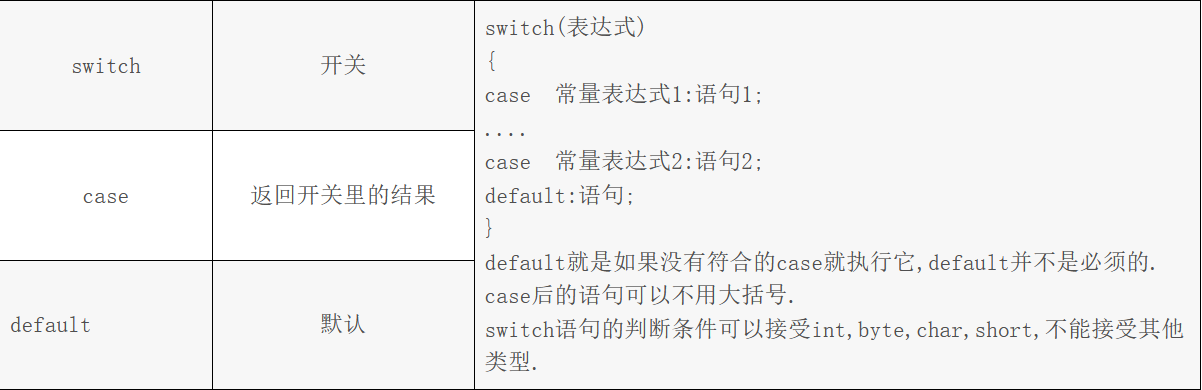




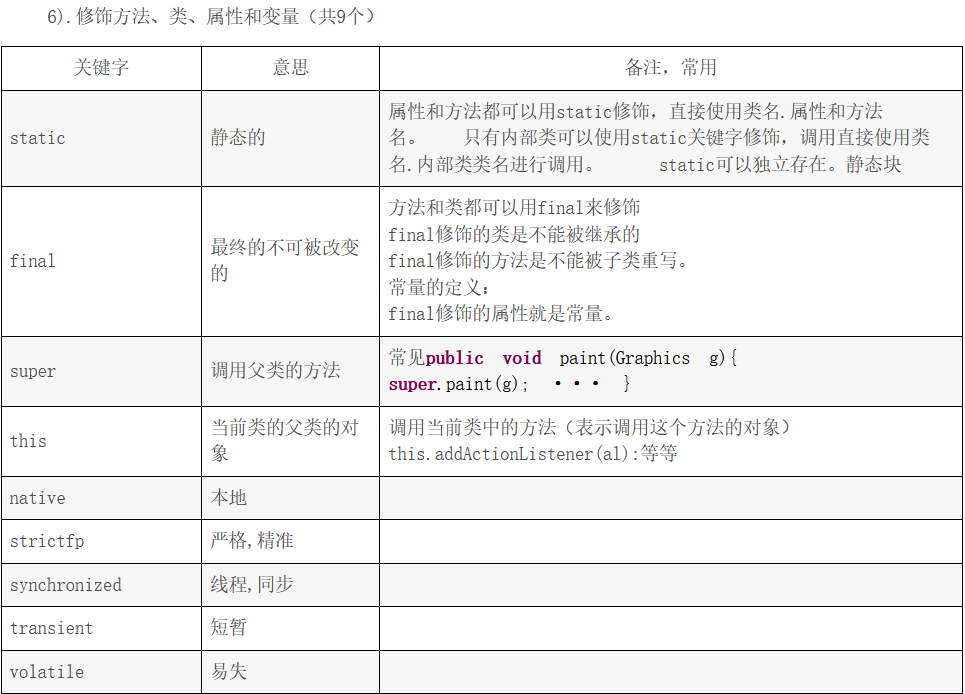


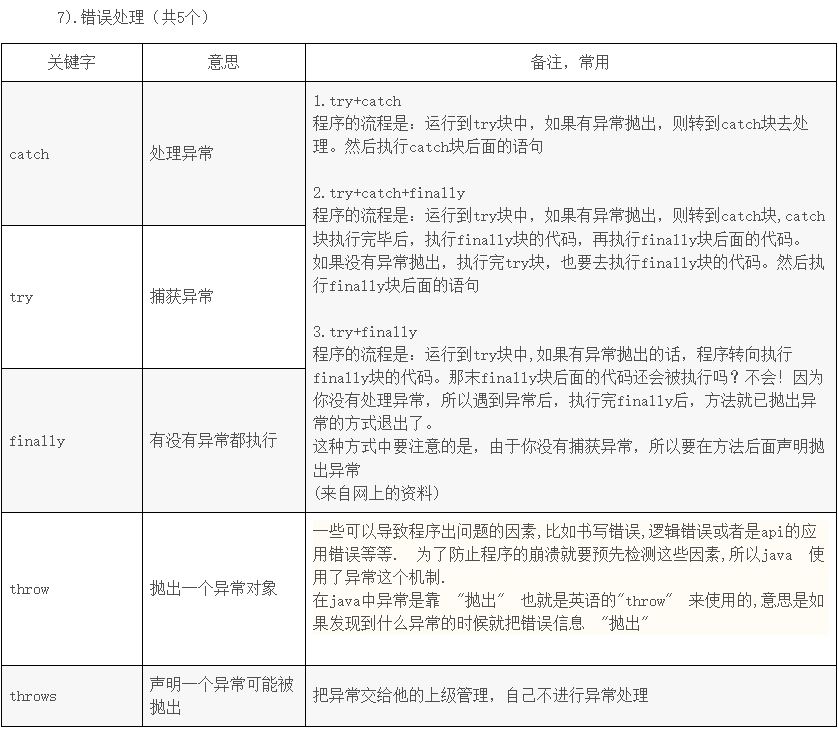












# final和finally的理解

final修饰局部变量 修饰参数列表

final 修饰成员变量,往往修饰静态成员变量

局部内部类中,使用的变量得是final修饰的,JDK1.8中可以省略final,但是不能产生事实上的修改

final 修饰方法 不能被重写

final 修饰类 不能被继承

finally是异常处理中的一个关键字,作为一个可选项,作用是定义一些无论异常是否出现都会执行的代码

finally语句块不能单独使用,必须依赖try语句,但是未必非要有catch

在具有返回值的方法中,finally语句块中应该尽量避免使用return返回结果

# 说一说接口和抽象类

接口和抽象类都是继承树的上层，他们的共同点如下：  
1) 都是上层的抽象层。  
2) 都不能被实例化  
3) 都能包含抽象的方法，这些抽象的方法用于描述类具备的功能，但是不比提供具体的实现

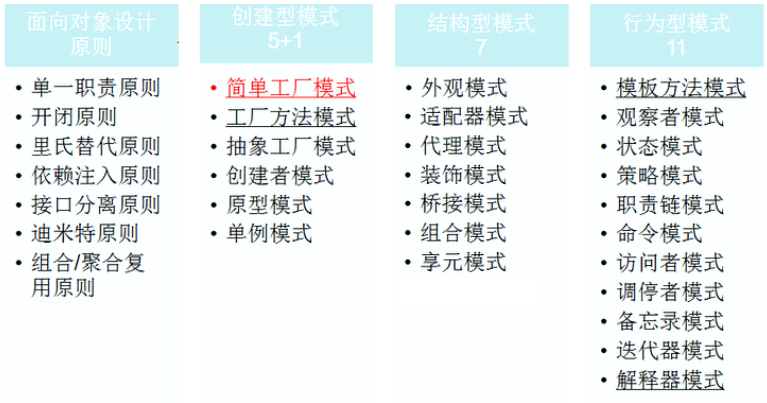
不同点如下

1)

# 说一说程序的设计原则和设计模式

## 设计原则

程序设计的要遵循的一些理论,也可以理解为程序设计的一种要求和目标,是面向对象程序设计的基石,也是面向对象程序设计的质量保障和依据





### 1.1单一职责原则



**解释**:

单一职责 SRP--Single Responsibility Principle

应该有且仅有一个原因引起类的变更

1系统中的每个类都应该只有一个职责,而所有类所关注的就是自身之职责的完成

2职责是指为”变化的原因”

3如果能想到多个原因去改变一个类,这个类就是多个职责

4并不是单一功能原则,并不是每个类只能有一个方法,而是单一”变化的愿意”原则

5如果一个类有多个职责,这些职责就耦合在了一起,当一个职责发生变化时,可能会影响其他职责

6多个职责耦合在一起,会影响服用性(可能只需要服用该类的某一个功能,但是该职责和其他职责耦合在一起,很难分离出来)

其实就是我们常说的高内聚低耦合原则,.单一职责原则是最简单也非常难实现的原则

**好处**

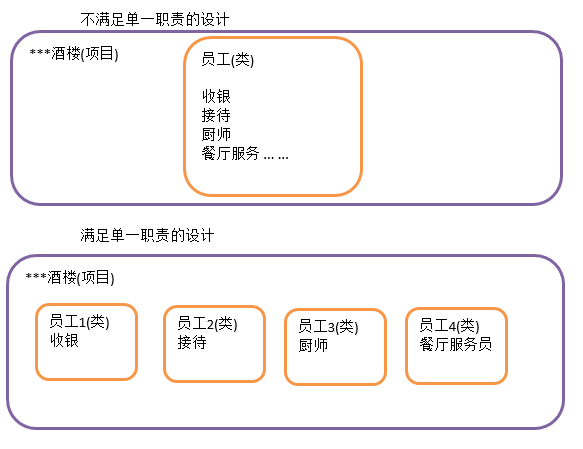
将功能分类,模块划分明确,修改一个模块不会造成其他模块的修改,降低模块之间的耦合度

**举例**:

MVC模式下,无论是Controller层还是service亦或是DAO层,针对每个表格都有独立的实现类,不要掺杂在一起,以免因为任何一个操作的变化都引起类的修改

**图解**:

饭店老板 既当厨师,又当服务员,这不合适



### 1.2开闭原则

**解释:**

开闭原则 OCP--- Open Closed Principle

软件实体应该对功能的拓展开放,对修改关闭的原则, 因为需求有变化,要求我们设计程序时必须为程序功能的新增留好接口,在新增功能时,不要修改原有代码,而是新增代码.让程序实现对拓展开放,对修改关闭测设计要求



薛定谔的猫理论,用户需求不确定理论

1实现开闭原则的关键是抽象

2定义一个抽象层,之规定功能而不提供实现,实现通过定义具体的类来完成

3当需求变化时,不通过修改抽象层来完成,而是通过定义抽象层的新实现完成

4通过抽象类及接口,规定了具体的类的特征作为抽象层,相对稳定,不需要修改,从而满足对修改关闭,从抽象类到处的具体类可以作为改变系统的行为,从而满足对扩展开放

**好处:**

1通过扩展已有软件系统,可以提供新的行为,以满足对软件新的需求,提高了软件系统的适应性和灵活性

2已有的软件模块,特别是重要的抽象层模块不能再修改,提高了软件系统的一定的稳定性和延续性

3这样的设计同时也满足了可复用性和可维护性

**举例:**

MVC模式下,每一层都有接口,是和其他层对接的规范,新增功能时,我们可以选择新增Controller层和service层和mapper层代码 不用去修改原有代码

**图解:**

每一个层次的要求都要留好接口,作为规范,增加功能就相当于多了一个实现类去实现接口,而不是在先有代码上修改,工厂方法模式

### 1.3里氏代换原则

**解释**

里氏代换原则 LSP------ Liskov Substitution Principle

所有引用的基类的地方必须能够透明的使用其子类的对象

凡是父类出现的地方,其子类就可以出现,而且调用子类还不能产生多于父类的错误和异常,调用者根本就不要需要知道是子类还是父类对象.但是反过来就不行,子类出现的地方父类未必就能使用

1子类必须完全实现了父类的方法,具备父类完全的功能

2子类可以有自己的特征

3覆盖和实现父类方法时,输入的参数可以被放大,但是不能被缩小

4覆盖和实现父类方法时,输出的结果可以被缩小,但是不能被放大

5覆盖和实现父类方法时,产生的异常可以被缩小,但是不能被放大

**好处**:

为我们如何实现良好的继承和使用多态提供了依据,也是实现开闭原则的重要保证

**举例**:

白马 马也

乘白马 乘马也

黑马 马也

乘黑马 乘马也

乘的是马 什么颜色的马都是马

马 是父类 各种颜色的马是子类 要求的是父类,所有的子类对象都可以使用

娣 美人也 爱娣 非爱美人也

美人 父类 娣 子类 要求是子类的地方,父类不一定能够出现

父类作为方法参数和返回值,可以让参数更灵活

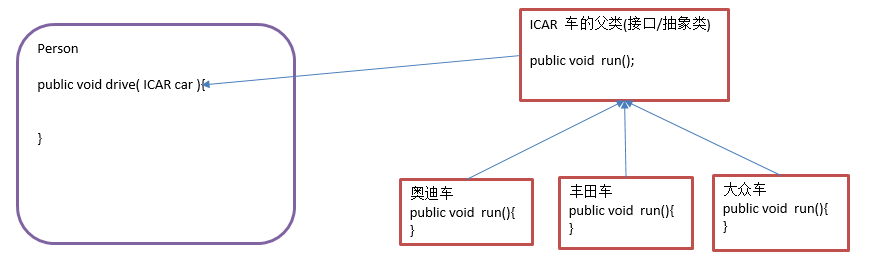
张三能开车

张三就能开奥迪

张三就能开奔驰

张三就能开宝马

**图解**:



### 1.4依赖倒置原则

**解释**

依赖倒置 DIP--Dependence Inversion Principle

高层模块不应该依赖底层模块,两者都应该依赖抽象,抽象不应该依赖细节,细节应该依赖抽象

抽象:即是抽象类或者接口,两者是不能够实例化的

细节:即是具体的嫌累,实现接口或者继承抽象类的类

依赖正置就是类之间的依赖是实实在在的实现类之间的依赖,也就是面向实现编程

依赖倒置就是通过抽象(抽象类或者接口),使各个模块之间实现彼此独立,不相互应影响

依赖倒置有三种方式实现

1构造函数传递依赖对象

2setter方法实现

3接口声明实现依赖对象 ()匿名内部类

依赖倒置原则要求

1每个类尽量都有接口和抽象类,或者抽象类和接口都有

2变量的表面类型应是接口和抽象类

3任何类都不应该从具体派生(尽量避免继承,做二次开发时,无法获得高层代码时除外)

4尽量不要覆盖基类已经实现好的方法(尽量不要重写父类方法)

**好处**:

采用依赖倒置原则可以降低模块之间的耦合性,提高系统的稳定性,减少并行开发的风险,提高代码的可读性和可维护性

**举例**:

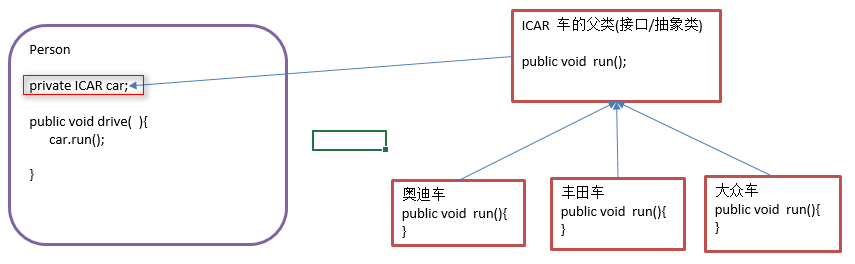
一个类组合另一个类作为属性时,应尽量选择抽象类或者是接口,尽量避免直接组合实现类

电脑里的零件坏了,如硬盘,内存,CPU等,那么直接更换对应的配件,只要插槽一样就能互换,维护比较方便.插槽就是接口,具体的硬件就是实现类.留好接口可以随时换零件,如果零件直接焊死在电脑上,坏了就没有办法更换了

**图解**:

抽象的Driver接口和抽象的Car接口 实现司机开车功能

|  |
| --- |
| interface Car{  public void run()  }  Audi implements Car{  public void run(){    }  }  WV implements Car{  public void run(){    }  }  TOYOTA implements Car{  public void run(){    }  }  Person{  private Car car;  public void drive(){  car.run();  }  } |



### 1.5接口隔离原则

**解释**

接口隔离原则 ISP Interface Segregation Principle

客户端不应该强行依赖他不需要的接口,类之间的依赖关系应该建立在最小的接口上,建立单一接口,不要建立庞大臃肿的接口,应该尽量细化接口,接种的方法尽量少,也就是说要为各个类建立专门的接口,而不要试图去建立一个庞大的接口供所有的依赖它的类去调用

接口隔离原则使用的一些规范

1接口尽量小,尽量细致

2接口尽量高内聚,尽量和其他接口撇清关系

3定制服务,为调用者提供且之提供他需要的方法

4接口设计有限度,根据业务及经验,仔细四高筹划,适度隔离接口

单一职责和接口隔离辨析

1单一职责原则侧重职责,接口隔离侧重对接口的依赖的隔离

2单一职责原则侧重约束类,其次是接口,针对程序中实现的细节

3接口隔离原则侧重约束接口,主要针对抽象需求,针对程序的整体框架的构建

**好处**:

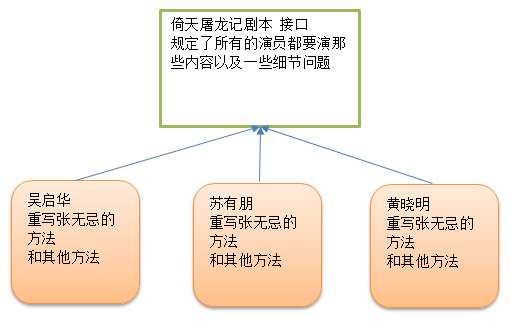
防止庞大,臃肿的接口,避免接口污染,提高程序设计要求的细致划分性,降低大面积维护成本.一旦出现接口污染,会造成实现类中存在大量的不相关不需要去重写的方法

**举例**:

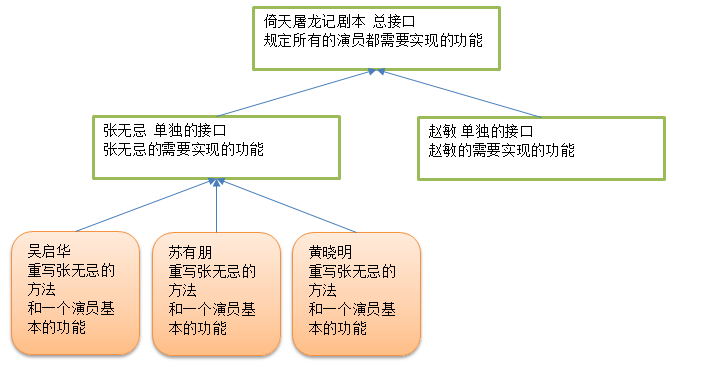
演员和角色以及整部戏的关系

**图解**:

不满足接口隔离原则的情况



满足接口隔离原则的情况



### 1.6迪米特法则

**解释**

迪米特法则 LOD--Law of Demeter

只与你只直接的朋友通信 ,不要和不相关的人产生大量通信,如果两个类不必彼此通信,那么两个类就不应到发生直接的相互作用,如果其中的一个类需要调用另一个类的方法的话,可以通过第三者转发这个调用

迪米特法则的初衷是在于降低类之间的耦合.但是迪米特法则有可能造成一个后果就是程序中存在大量的中介类,这些中介类完全就是为了传递类间的相互调用关系,一定程度上增加了系统的复杂度

**好处**:

降低了类和类之间的耦合

**举例**:

找中介卖房子,不用我们自己直接接触大量的买房人员

**图解**:





### 1.7合成/聚合复用原则

**解释**

合成/聚合复用原则 CARP Composite Aggregate Reuse Principle

当一个类想使用另一个类的功能时,优先使用对象的组合,而不是继承,尽量多使用组合

合成聚合复用原则是值在一个新对象中通过组合关系使用原来已有的一些对象,使之成为新对象的一部分,通过使用已有对象的API完成已有功能的调用

为什么要是用合成聚合,尽量不要使用继承?

1继承破坏包装,把超类的实现细节直接暴露给子类,不利于信息的隐匿

2如果父类发生改变,会引发一系列子类的改变,类之间耦合度高

3继承是一种静态功能的使用,在运行的过程中不能发生改变,聚合复用可以动态传入子类对象实现功能动态改变

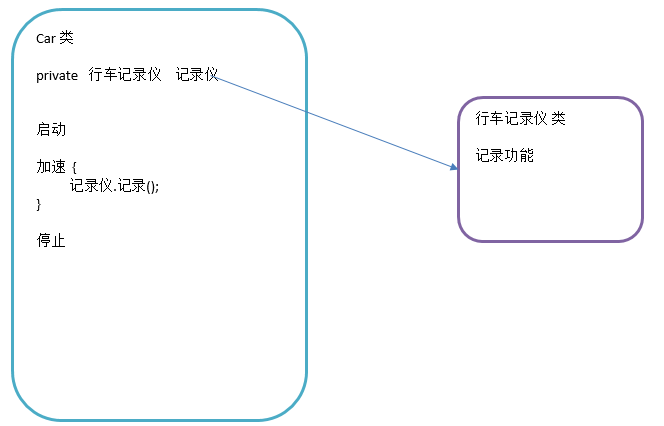
**好处**:

非常有利于构建可维护,可复用,可扩展和灵活性好的软件系统

**举例**:

汽车上如果想获得行车记录的功能,单独组合一个行车记录仪对象作为属性即可,没有必要继承行车记录仪类

**图解**:



## 设计模式概念

### 1.1 设计模式引入

设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、提高扩展性、提高灵活性和提高开发效率，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。简单说：

模式：在某些场景下，针对某类问题的某种通用的解决方案。

场景：项目所在的环境

问题：约束条件，项目目标等

解决方案：通用、可复用的设计，解决约束达到目标。

设计模式可以分为三种类型

创建型模式：对象实例化的模式，创建型模式用于解耦对象的实例化过程。

结构型模式：把类或对象结合在一起形成一个更大的结构。

行为型模式：类和对象如何交互，及划分责任和算法。



### 1.2 创建型模式

对象实例化的模式，创建型模式用于解耦对象的实例化过程。这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。创建型设计模式描述了如何创建类，对类的实例化过程进行了抽象，将软件模块中对象的创建和使用分离。在使用这些对象时只需要知道接口，而不用知道细节

主要有:

单例模式：某个类只能有一个实例，提供一个全局的访问点。

简单工厂：一个工厂类根据传入的参量决定创建出那一种产品类的实例。

工厂方法：定义一个创建对象的接口，让子类决定实例化那个类。

抽象工厂：创建相关或依赖对象的家族，而无需明确指定具体类。

建造者模式：封装一个复杂对象的构建过程，并可以按步骤构造。

原型模式：通过复制现有的实例来创建新的实例。

### 1.3 结构型模式

把类或对象结合在一起形成一个更大的结构。这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。结构型设计模式可以分为类结构型模式（关心类的组合，一般只存在继承关系和实现关系）和对象结构型模式（关心类与对象的组合，通过关联关系使得在一个类中定义另一个类的实例对象，然后通过该对象调用其方法）。根据合成复用原则，尽量用关联关系，少用继承关系，因此大部分结构型设计模式都是对象结构型模式。

主要有:

适配器模式：将一个类的方法接口转换成客户希望的另外一个接口。

组合模式：将对象组合成树形结构以表示“”部分-整体“”的层次结构。

装饰模式：动态的给对象添加新的功能。

代理模式：为其他对象提供一个代理以便控制这个对象的访问。

亨元（蝇量）模式：通过共享技术来有效的支持大量细粒度的对象。

外观模式：对外提供一个统一的方法，来访问子系统中的一群接口。

桥接模式：将抽象部分和它的实现部分分离，使它们都可以独立的变化。

### 1.3 行为型模式

用于描述类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，以及怎样分配职责类和对象如何交互，及划分责任和算法,这些设计模式特别关注对象之间的通信。

主要有:

模板模式：定义一个算法结构，而将一些步骤延迟到子类实现。

解释器模式：给定一个语言，定义它的文法的一种表示，并定义一个解释器。

策略模式：定义一系列算法，把他们封装起来，并且使它们可以相互替换。

状态模式：允许一个对象在其对象内部状态改变时改变它的行为。

观察者模式：对象间的一对多的依赖关系。

备忘录模式：在不破坏封装的前提下，保持对象的内部状态。

中介者模式：用一个中介对象来封装一系列的对象交互。

命令模式：将命令请求封装为一个对象，使得可以用不同的请求来进行参数化。

访问者模式：在不改变数据结构的前提下，增加作用于一组对象元素的新功能。

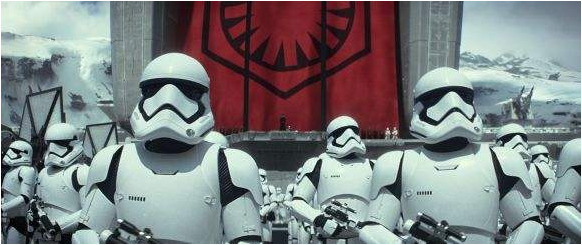
责任链模式：将请求的发送者和接收者解耦，使的多个对象都有处理这个请求的机会。

迭代器模式：一种遍历访问聚合对象中各个元素的方法，不暴露该对象的内部结构。

## 创建型设计模式

### 2.1 原型模式

通过new关键字产生一个对象需要非常繁琐的数据准备或者访问权限,为了方便快捷的创建一个一模一样的对象,我们可以选择使用原型模式,即把一个对象当做原型,通过原型拷贝(复制)对象.克隆出来的新的对象和原有对象相同.JAVA中已有原型模式的处理方案,我们直接使用就可以了.



浅克隆

|  |
| --- |
| **package** com.msb.prototype;  **public class** Test1 {  **public static void** main(String[] args) **throws** CloneNotSupportedException {  Person p=**new** Person(**"张三"**,**"男"**);  System.***out***.println(p.toString());  Person p2 =(Person)p.clone();  System.***out***.println(p2.toString());  System.***out***.println(p==p2);   } } **class** Person **implements** Cloneable{  **private** String **name**;  **private** String **gender**;   @Override  **public** Object clone() **throws** CloneNotSupportedException {  **return super**.clone();  }   @Override  **public** String toString() {  **return "Person{"** +  **"name='"** + **name** + **'\''** +  **", gender='"** + **gender** + **'\''** +  **'}'**;  }   **public** Person() {  }   **public** Person(String name, String gender) {  **this**.**name** = name;  **this**.**gender** = gender;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public** String getGender() {  **return gender**;  }   **public void** setGender(String gender) {  **this**.**gender** = gender;  } } |

深克隆(组合关系下的克隆)

|  |
| --- |
| **package** com.msb.prototype;  **public class** Test1 {  **public static void** main(String[] args) **throws** CloneNotSupportedException {   Person p=**new** Person(**"张三"**,**"男"**,**new** Dog(**"大黄"**,**"金毛"**));  System.***out***.println(p.toString());  Person p2 =(Person)p.clone();  System.***out***.println(p2.toString());  System.***out***.println(p==p2);  System.***out***.println(p.getDog()==p2.getDog());   } } **class** Person **implements** Cloneable{  **private** String **name**;  **private** String **gender**;  **private** Dog **dog**;   @Override  **public** Object clone() **throws** CloneNotSupportedException {  Person person = (Person)**super**.clone();  Dog dog =(Dog) **this**.getDog().clone();  person.setDog(dog);  **return** person;  }   **public** Person() {  }   @Override  **public** String toString() {  **return "Person{"** +  **"name='"** + **name** + **'\''** +  **", gender='"** + **gender** + **'\''** +  **", dog="** + **dog** +  **'}'**;  }   **public** Person(String name, String gender, Dog dog) {  **this**.**name** = name;  **this**.**gender** = gender;  **this**.**dog** = dog;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public** String getGender() {  **return gender**;  }   **public void** setGender(String gender) {  **this**.**gender** = gender;  }   **public** Dog getDog() {  **return dog**;  }   **public void** setDog(Dog dog) {  **this**.**dog** = dog;  } }  **class** Dog **implements** Cloneable{  **private** String **name**;  **private** String **type**;   @Override  **public** Object clone() **throws** CloneNotSupportedException {  **return super**.clone();  }   @Override  **public** String toString() {  **return "Dog{"** +  **"name='"** + **name** + **'\''** +  **", type='"** + **type** + **'\''** +  **'}'**;  }   **public** Dog() {  }   **public** Dog(String name, String type) {  **this**.**name** = name;  **this**.**type** = type;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public** String getType() {  **return type**;  }   **public void** setType(String type) {  **this**.**type** = type;  } } |

### 2.2 简单工厂模式

让一个方法可以返回不同的值,让返回值更灵活的一种设计模式.父类/接口/抽象类作为方法的返回值 类型,子类/实现类对象就可以作为实际结果返回,要求我们设计方法时,尽量选择父类作为方法的参数和返回值. 里氏代换原则:父类出现的地方,子类就可以出现,子类出现的地方,父类一般不可以出现

一般简单工厂模式有如下几个组成部分

1一个抽象的父类/接口作为方法的返回值 抽象的产品

2多个实际可以使用子类作为可以返回的结果 实际可以返回产品

3一个用父类作为返回值类型的一个方法 获得产品的工厂

举例:

水果工厂,可以根据要求,返回不同的水果

1一个抽象的父类/接口作为方法的返回值

|  |
| --- |
| **interface** Fruit{  **void** plant();  **void** grow();  **void** harvest(); } |

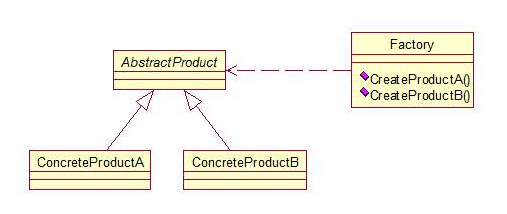
2多个实际可以使用子类作为可以返回的结果

|  |
| --- |
| **class** Banana **implements** Fruit{  @Override  **public void** plant() {  System.***out***.println(**"香蕉种植了"**);  }   @Override  **public void** grow() {  System.***out***.println(**"香蕉正在猥琐发育"**);  }   @Override  **public void** harvest() {  System.***out***.println(**"香蕉黄了,可以吃了"**);  } } **class** Apple **implements** Fruit{  @Override  **public void** plant() {  System.***out***.println(**"种下一颗种子"**);  }   @Override  **public void** grow() {  System.***out***.println(**"苹果在慢慢的变大"**);  }   @Override  **public void** harvest() {  System.***out***.println(**"苹果红了,可摘下来啦"**);  } }  **class** Grape **implements** Fruit{  @Override  **public void** plant() {  System.***out***.println(**"栽下一棵葡萄树"**);  }   @Override  **public void** grow() {  System.***out***.println(**"葡萄在慢慢长大"**);  }   @Override  **public void** harvest() {  System.***out***.println(**"葡萄紫了,可以吃了"**);  } } |

3一个用父类作为返回值类型的一个方法

|  |
| --- |
| **class** FruitFactory{  **public** Fruit getFruit(String name){  **if**(name.equals(**"banana"**)){  **return new** Banana();  }**else if**(name.equals(**"apple"**)){  **return new** Apple();  }**else**{  **return new** Grape();  }  } } |

UML图解:



符合里氏代换原则,但是不符合开闭原则.如果我们增加一种水果,那么就要修改工厂中的代码.

开闭原则:程序设计的原则之一,对拓展开放,对修改关闭.当程序的需求功能发生变化时,尽量以增加代码的形式完成,而不是以修改代码的形式实现

### 2.3 工厂方法模式

简单工厂模式符合里氏代换原则,但是不符合开闭原则.如果我们增加一种水果,那么就要修改工厂中的代码.如何让工厂模式符合开闭原则呢?我们可以把工厂也抽象成一个接口,为每个实体产品都准备专有的工厂,如果新增产品,那么只要新增一个工厂类和产品类即可,不用修改原有代码.

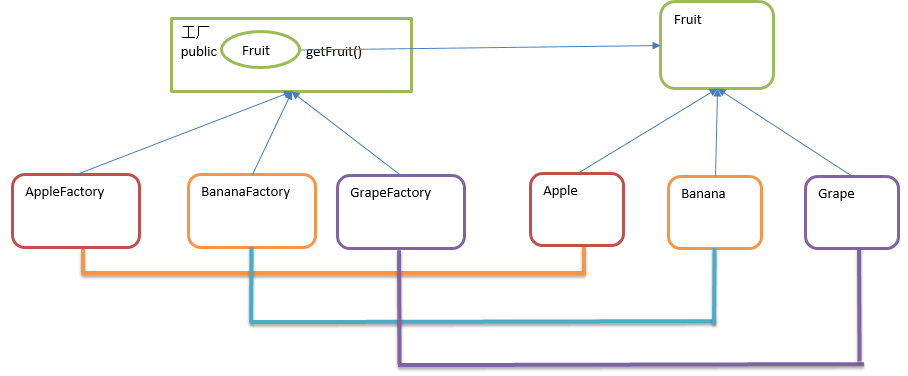
共有以下几个组成部分

1抽象的产品父类/抽象类 抽象的产品

2多个产品的实现类 实际的产品

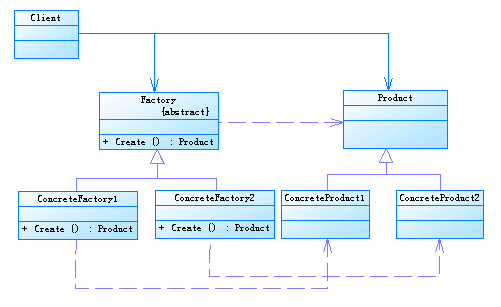
3抽象的工厂 工厂的接口

4针对于每个实体产品的专用工厂 工厂的实现类



|  |
| --- |
| **package** com.msb.factoryMethod;  **public class** Test2 {  **public static void** main(String[] args) {  FruitFactory ff =**new** OrangeFactory();  Fruit f = ff.getFruit();  f.plant();  f.grow();  f.harvest();  } } 3抽象的工厂 **interface** FruitFactory{  **public** Fruit getFruit(); }  4针对于每个实体产品的专用工厂 **class** BananaFactory **implements** FruitFactory{  @Override  **public** Fruit getFruit() {  **return new** Banana();  } } **class** AppleFactory **implements** FruitFactory{  @Override  **public** Fruit getFruit() {  **return new** Apple();  } } **class** GrapeFactory **implements** FruitFactory{  @Override  **public** Fruit getFruit() {  **return new** Grape();  } } **class** OrangeFactory **implements** FruitFactory{  @Override  **public** Fruit getFruit() {  **return new** Orange();  } } 1抽象的产品父类/抽象类 **interface** Fruit{  **void** plant();  **void** grow();  **void** harvest(); } 2多个产品的实现类 **class** Banana **implements** Fruit{  @Override  **public void** plant() {  System.***out***.println(**"香蕉种植了"**);  }   @Override  **public void** grow() {  System.***out***.println(**"香蕉正在猥琐发育"**);  }   @Override  **public void** harvest() {  System.***out***.println(**"香蕉黄了,可以吃了"**);  } }  **class** Apple **implements** Fruit{  @Override  **public void** plant() {  System.***out***.println(**"种下一颗种子"**);  }   @Override  **public void** grow() {  System.***out***.println(**"苹果在慢慢的变大"**);  }   @Override  **public void** harvest() {  System.***out***.println(**"苹果红了,可摘下来啦"**);  } }  **class** Grape **implements** Fruit{  @Override  **public void** plant() {  System.***out***.println(**"栽下一棵葡萄树"**);  }   @Override  **public void** grow() {  System.***out***.println(**"葡萄在慢慢长大"**);  }   @Override  **public void** harvest() {  System.***out***.println(**"葡萄紫了,可以吃了"**);  } } **class** Orange **implements** Fruit{  @Override  **public void** plant() {  System.***out***.println(**"栽下一棵橘子树"**);  }   @Override  **public void** grow() {  System.***out***.println(**"橘子在慢慢长大"**);  }   @Override  **public void** harvest() {  System.***out***.println(**"具体甜了,可以吃了"**);  } } |

UML图解:



### 2.4 单例模式

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 ... ...

#### 2.4.1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！

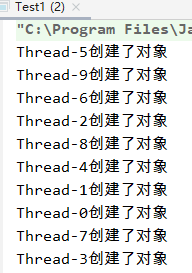
#### 2.4.2懒汉式单例模式

|  |
| --- |
| **package** com.msb.singleTon; */\* 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*;  } } |



#### 2.4.3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*;  } } |

#### 2.4.4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

#### 2.4.5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



## 结构型设计模式

### 3.1 外观模式(门面模式)

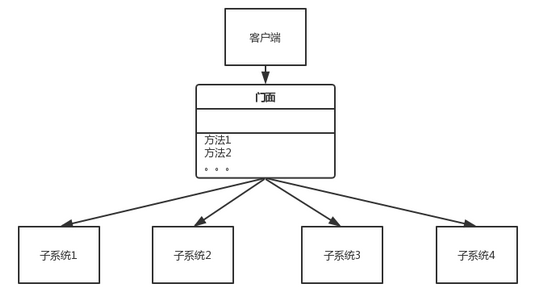
外观模式提供了一个统一的接口，用来访问子系统中的一群接口。它让一个应用程序中子系统间的相互依赖关系减少到了最少，它给子系统提供了一个简单、单一的屏障，客户通过这个屏障来与子系统进行通信。通过使用外观模式，使得客户对子系统的引用变得简单了，实现了客户与子系统之间的松耦合。

使用外观模式之前



使用外观模式之后

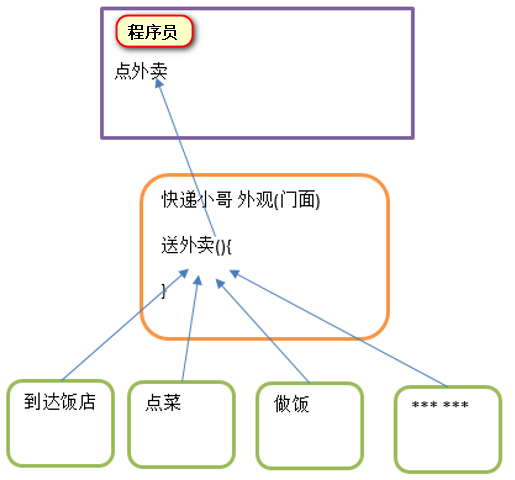




举例:

使用美团定外卖吃饭和自己去去店里打包带回家吃对比

逻辑图如下



|  |
| --- |
| **package** com.msb.facadeDemo;  **public class** Test1 {  **public static void** main(String[] args) {  WaimaiBrother wb =**new** WaimaiBrother();  wb.songwaimai();  } } *// 外观 门面* **class** WaimaiBrother{  **public void** songwaimai(){  A a = **new** A();  B b = **new** B();  C c = **new** C();  D d = **new** D();  E e = **new** E();  F f = **new** F();  a.methodA();  b.methodb();  c.methodc();  d.methodd();  e.methode();  f.methodf();  } }  **class** A{  **public void** methodA(){  System.***out***.println(**"到达饭店"**);  } } **class** B{  **public void** methodb(){  System.***out***.println(**"跟服务员点菜"**);  } } **class** C{  **public void** methodc(){  System.***out***.println(**"厨师开始做菜"**);  } } **class** D{  **public void** methodd(){  System.***out***.println(**"初始把菜做好了"**);  } } **class** E{  **public void** methode(){  System.***out***.println(**"服务员将饭菜打包"**);  } } **class** F{  **public void** methodf(){  System.***out***.println(**"将饭菜带走"**);  } } |

### 3.2 装饰者模式

一种动态的给一个对象添加功能,动态撤销的一种模式,多个功能之间可以实现灵活的叠加的模式,是继承的一种替换解决方案,可以在面向大量不同的要求时减少子类的创建数量.在不必改变原类文件和使用继承的情况下,动态扩展一个类的功能,提供了比继承更具有弹性的代替方案.让程序更加符合开闭原则和聚合复用原则

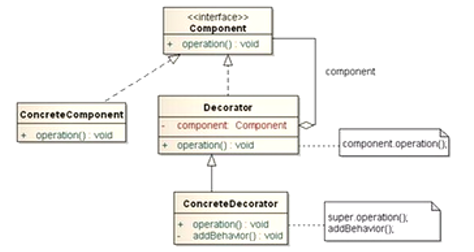
装饰者模式涉及到的组成部分

1一个抽象构建角色,一个抽象的被装饰物 蛋糕

2多个具体的构建角色,多个实际的被装饰物 奶油蛋糕 巧克力蛋糕 水果蛋糕 ……

3一个抽象的装饰角色,一个抽象的装饰者 饰品

4多个具体的装饰角色,多个实际的装饰者 卡片 花瓣 蜡烛 果仁… …



举例:

蛋糕店制作蛋糕

蛋糕有:奶油蛋糕 巧克力蛋糕 水果蛋糕 冰淇淋蛋糕 等等

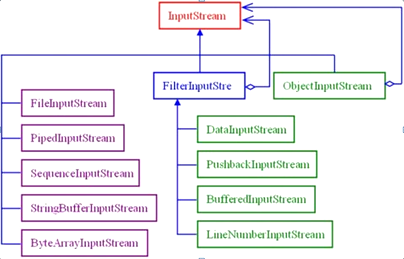
装饰物有:卡片,花瓣,果仁 等等

我们可以根据需求,灵活的在蛋糕上添加各种装饰物而不用大量创建对象

|  |
| --- |
| **package** com.msb.decoration;  **import** com.sun.org.apache.bcel.internal.classfile.Deprecated;  **public class** Test1 {  **public static void** main(String[] args) {  ChoCake cake=**new** ChoCake();  FlowerDecorator fd =**new** FlowerDecorator(cake);  *//CardDecorator cd =new CardDecorator(fd);* NutDecorator nd=**new** NutDecorator(fd);  nd.make();  } } *//1一个抽象构建角色,一个抽象的被装饰物* **abstract class** Cake{  **public abstract void** make(); } *//2多个具体的构建角色,多个实际的被装饰物* **class** FruitCake **extends** Cake{  @Override  **public void** make() {  System.***out***.println(**"制作了水果蛋糕一个"**);  } } **class** ChoCake **extends** Cake{  @Override  **public void** make() {  System.***out***.println(**"制作了巧克力蛋糕一个"**);  } } **class** CreamCake **extends** Cake{  @Override  **public void** make() {  System.***out***.println(**"制作了奶油蛋糕一个"**);  } } *//3一个抽象的装饰角色,一个抽象的装饰者* **abstract class** Decorator **extends** Cake{  **private** Cake **cake**;  @Override  **public void** make() {  **cake**.make();  }  **public** Decorator(Cake cake) {  **this**.**cake** = cake;  } } *//4多个具体的装饰角色,多个实际的装饰者* **class** FlowerDecorator **extends** Decorator{  **public** FlowerDecorator(Cake cake) {  **super**(cake);  }  @Override  **public void** make() {  **super**.make();  System.***out***.println(**"撒上一些花瓣"**);  } } **class** CardDecorator **extends** Decorator{  **public** CardDecorator(Cake cake) {  **super**(cake);  }  @Override  **public void** make() {  **super**.make();  System.***out***.println(**"放上一张贺卡"**);  } } **class** NutDecorator **extends** Decorator{  **public** NutDecorator(Cake cake) {  **super**(cake);  }  @Override  **public void** make() {  **super**.make();  System.***out***.println(**"撒上一点果仁"**);  } } |

实际开发中的应用:

IO流体系中,处理流之间和实现功能的动态叠加



|  |
| --- |
| *// 输入字节节点流 被装饰物* InputStream is =**new** FileInputStream(**""**);  *// 直接读取对象的 对象处理就 装饰者1* ObjectInputStream ois =**new** ObjectInputStream(is); *// 带缓冲区的字节处理流 添加缓冲功能 装饰者2* BufferedInputStream bis =**new** BufferedInputStream(ois); |

### 3.3 代理模式



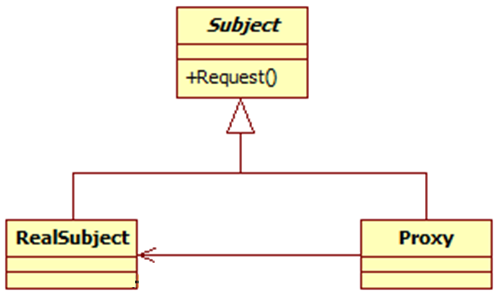
代理模式就是给一个对象提供一个代理，并由代理对象控制对原对象的引用。它使得客户不能直接与真正的目标对象通信。代理对象是目标对象的代表，其他需要与这个目标对象打交道的操作都是和这个代理对象在交涉。

代理对象是客户端和目标对象之间起的中介,它有如下三个主要作用:

1控制目标对象不能直接访问,对目标对象可以形成一定的保护作用

2一定程度上面减少了目标对象和客户端之间的耦合度

3代理模式可以在访问过程中增加逻辑,实现功能的扩展



代理模式涉及到的组成部分

1一个被代理的对象

2一个代理对象

3一个使用代理对象/被代理对象的客户端程序



代理模式按照创建机制来划分,分为静态代理和动态代理两种模式

**1静态代理**:由程序员创建代理类,在程序运行前代理类的.class文件就已经存在了, 代理对象只服务于一种类型的对象

**2动态代理**:在程序运行时运用反射机制动态创建代理对象,能够代理各种类型的对象

#### 3.3.1 静态代理

一种我们自己定义代理类的模式

需求:张三请律师打官司

法庭接口

|  |
| --- |
| **public interface** Court {  **void** doCourt(); } |

实现类

|  |
| --- |
| **public class** Person **implements** Court {  @Override  **public void** doCourt() {  System.***out***.println(**"我没有超速"**);  } } |

代理类

|  |
| --- |
| **public class** Lawyer **implements** Court{  **private** Court **court**;  @Override  **public void** doCourt() {  System.***out***.println(**"经取证,当事人是没有开车,在骑自行车"**);  System.***out***.println(**"当事人不可能骑自行车超速"**);  **court**.doCourt();  }   **public** Lawyer() {  }   **public** Lawyer(Court court) {  **this**.**court** = court;  }   **public** Court getCourt() {  **return court**;  }   **public void** setCourt(Court court) {  **this**.**court** = court;  } } |

测试代码

|  |
| --- |
| **public class** Test1 {  **public static void** main(String[] args) {  Court c =**new** Person();  Lawyer l =**new** Lawyer(c);  l.doCourt();  } } |

总结:

1代理类和委托类实现了相同的接口，代理类通过委托类实现了相同的方法。这样就出现了大量的代码重复。如果接口增加一个方法，除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法。增加了代码维护的复杂度。

2代理对象只服务于一种类型的对象，如果要服务多类型的对象。势必要为每一种对象都进行代理。

#### 3.3.2 Proxy动态代理

Prxoy动态代理是一种面向接口的动态代理,可以实现代理实现类对象去完成接口的要求.也就是说,这种动态代理必须有接口

需求:动态代理模拟张三请律师打官司

**1一个法庭接口**

|  |
| --- |
| **public interface** Court {    **int** doCourt(**int** a); } |

**2一个实现类**

|  |
| --- |
| **public class** Person **implements** Court{  **private** String **name**;   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public** Person(String name) {  **this**.**name** = name;  }   **public** Person() {  }   @Override  **public int** doCourt( **int** a) {  System.***out***.println(**name**+**"说:我没超速"**);  **return** a\*10;  }  **public void** eat(){   } } |

**3通过Proxy动态生成一个代理对象**

|  |
| --- |
| **public class** Test1 {  **public static void** main(String[] args) {  *// 张三请律师打官司* Person person=**new** Person(**"张三"**);  *//person.doCourt();  // 给person 生成一个代理对象 可以完成被代理的person对象原有的功能 同时还可以进行拓展和增强  // JDK Proxy  /\* Proxy.newProxyInstance 方法的三个参数  1 类加载器  加载一个类的工具 用于将类的字节码 加载进内存后 进行解析 解析成 Class Filed Method  通过类的字节码对象获得 person.getClass().getClassLoader()  2 被代理的对象所在的类 person >>> Person 所有实现的接口的 字节码 数组  通过类的字节码对象获得 person.getClass().getInterfaces()  3 方法执行的处理器对象 定义增强被代理对象方法规则 person doCourt()  添加一些其他功能的代码 实现类 invocationHandler 接口的一个实现类对象  \*/  // 参数1* ClassLoader classLoader = person.getClass().getClassLoader();  *// 参数2* Class[] interfaces =person.getClass().getInterfaces();*// {Court.class};  // 参数3* InvocationHandler handler =**new** InvocationHandler(){  *// 定义 增强规则的方法 当 执行代理对象中任何一个方法的时候 都会触发 invoke方法  // 当代理对象 执行任何一个接口中定义方法时 都会将被代理对象原有的方法传过来 执行 invoke 方法一次* @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  */\*  proxy >>>> 代理对象  method >>>> 被代理对象 person 对象中原有的真正被执行的 已经实现的方法 Person doCourt  手动控制 该方法的执行  args >>>> 被代理对象 方法被调用时 传入的 实参 {10}  return null; 返回被增强方法的结果  \*/   /\*System.out.println(method.getName());  System.out.println(Arrays.toString(args));\*/* Object res =**null**;  **if**(method.getName().equals(**"doCourt"**)){*// 当执行 被代理的方法 是 doCourt 的时候 增强 如果是其他方法 让其正常执行即可* System.***out***.println(**"律师取证:当事人当时在骑车"**);  *// 让被代理对象原有的方法执行* res =method.invoke(person,args);  *// 律师添加其他的增强代码* System.***out***.println(**"经过论证:当事人不具超速条件"**);  }**else**{*// 其他方法 正常执行即可 speak* res =method.invoke(person,args);  }  *// 返回原有方法的运行结果* **return** res;  }  };   Court lawyer = (Court)Proxy.*newProxyInstance*(classLoader,interfaces,handler);  *// 使用 代理对象 代替person对象去完成打官司的功能* **int** x =lawyer.doCourt(10);  System.***out***.println(x);  } } |

总结:

1.代理模式可以在在不修改原有代码,或者没有办法修改原有代码的情况下增强对象功能 ,使用代理对象代替原来的对象去完成功能,进而达到拓展功能的目的

2.Proxy动态代理面向接口的动态代理, 一定要有接口和实现类的存在,代理对象增强的是实现类在实现接口时重写的方法

3.Proxy生成的代理对象只能转换成接口类型,不能转换成实现类类型

4.Proxy代理对象只能增强接口中定义的方法,实现类中定义的方法是无法增强的

5.Proxy只能读取到接口中方法上的注解,不能读取到实现类方法上的注解

#### 3.3.3 cglib动态代理

cglib动态代理是一种面向父类的动态代理,其实现原理是通过子类继承父类,子类重写父类方法时可以实现功能增强.所以在cglib代理模式下,有没有接口皆可以实现,只要明确父类即可

需求:动态代理模拟增强父类定义的 eat()方法

1定义一个普通的被代理类

|  |
| --- |
| **public class** Person {  **public void** eat(){  System.***out***.println(**"吃东西"**);  } } |

2使用cglib技术实现动态代理增强eat方法

|  |
| --- |
| **public class** TestCglib {  **public static void** main(String[] args) {  Person p =**new** Person();  *// p.eat();  // 使用cglib 生成 p对象的一个子类对象* Enhancer en =**new** Enhancer(); *// 获取 Enhancer 对象 用于生成代理对象的工具* en.setSuperclass(p.getClass());*// 设置父类的字节码  /\*  需要传入一个参数  MethodIntercepter 一个对象 用于定义 增强父类方法具体功能的 接口  \*/* **class** MyMethodIntercepter **implements** MethodInterceptor{  @Override *// invoked* **public** Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) **throws** Throwable {  */\*  Object o 生成之后的代理对象 子类对象 personProxy  method 父类中原本要执行的方法 Person >>>> eat  objects 父类方法在执行时 传入的参数数组  methodProxy 子类对象中 重写的父类的方法 personProxy >>>> eat()  \*/  // 执行其父类中原本的方法* System.***out***.println(**"饭前洗手"**);  Object res =methodProxy.invokeSuper(o,objects);*// 执行子类对象中的方法 子类对象方法在执行时 一定会调用父类的方法* System.***out***.println(**"饭后刷碗"**);  **return** res;  }  }  en.setCallback(**new** MyMethodIntercepter()); *// 用于定义增强父类方法中细节* Person personProxy =(Person) en.create();*// 生成了一个代理对象 本质上就是 Person类的一个子类对象* personProxy.eat();  } } |

总结:

1.cglib动态代理面向父类的动态代理, 有无接口都可以使用

2.cglib生成的代理对象可以直接声明成父类对象,或者是接口

3.cglib只要是父类中能够被子类继承的或者重写的方法,都可以实现增强

4.cglib可以读取到实现类方法上的注解

## 行为型设计模式

### 4.1 责任链模式



将能够处理同一类请求的对象连城一条链,所提交的请求沿着链条传递,链条上的对象逐个判断是否有能力处理该问题,如果能处理则处理,不能处理则向后传递给下一个对象

常用场景有

打牌时轮换出牌

接力赛跑

各种审批……

责任链模式组成部分

1一个要处理的请求

2一个抽象的处理者

3多个具体的处理者形成责任处理顺序



举例:

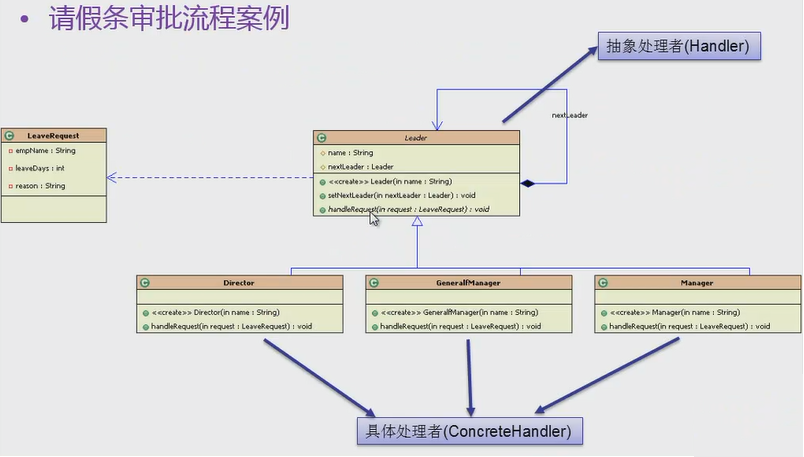
员工请假审批:

如果请假天数小于3天,主任审批即可

如果请假天数大于等于3天,小于10天,经理审批即可

如果请假天数大于等于10天,小于30天,总经理审批

如果请假天数大于等于30天,审批不通过



请假申请对象

|  |
| --- |
| */\*\*  \* 请假请求封装成一个对象  \*/* **class** LeaveRequest{  **private** String **ename**;  **private int leaveDays**;  **private** String **reason**;   **public** LeaveRequest() {  }   **public** LeaveRequest(String ename, **int** leaveDays, String reason) {  **this**.**ename** = ename;  **this**.**leaveDays** = leaveDays;  **this**.**reason** = reason;  }   **public** String getEname() {  **return ename**;  }   **public void** setEname(String ename) {  **this**.**ename** = ename;  }   **public int** getLeaveDays() {  **return leaveDays**;  }   **public void** setLeaveDays(**int** leaveDays) {  **this**.**leaveDays** = leaveDays;  }   **public** String getReason() {  **return reason**;  }   **public void** setReason(String reason) {  **this**.**reason** = reason;  } } |

抽象的审批者类

|  |
| --- |
| */\*\*  \* 抽象的审批者类  \*/* **abstract class** Leader{  *//审批者姓名* **private** String **name**;  *// 下一级审批者* **private** Leader **nextLeader**;  *// 处理审批业务的核心方法* **public abstract void** handleLeaveRequest(LeaveRequest request);   **public** Leader(String name) {  **this**.**name** = name;  }   **public** Leader(Leader nextLeader) {  **this**.**nextLeader** = nextLeader;  }   **public** Leader(String name, Leader nextLeader) {  **this**.**name** = name;  **this**.**nextLeader** = nextLeader;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public** Leader getNextLeader() {  **return nextLeader**;  }   **public void** setNextLeader(Leader nextLeader) {  **this**.**nextLeader** = nextLeader;  }    } |

多层具体的审批者

|  |
| --- |
| **class** Director **extends** Leader{  **public** Director(String name){  **super**(name);  }  @Override  **public void** handleLeaveRequest(LeaveRequest request) {  System.***out***.println(**"主任"**+**this**.getName()+**"正在审批"**+request.getEname()+**"的"**+request.getReason()+**"的请假申请"**);  **if**(request.getLeaveDays()<3){  System.***out***.println(**"审核意见:请假"**+request.getLeaveDays()+**"天,同意"**);  }**else**{  **if**(**null** !=**this**.getNextLeader()){  System.***out***.println(**"审核意见:请假"**+request.getLeaveDays()+**"天,请上级继续审批"**);  **this**.getNextLeader().handleLeaveRequest(request);  }  }  } } **class** ViceManager **extends** Leader{  **public** ViceManager(String name){  **super**(name);  }  @Override  **public void** handleLeaveRequest(LeaveRequest request) {  System.***out***.println(**"副总"**+**this**.getName()+**"正在审批"**+request.getEname()+**"的"**+request.getReason()+**"的请假申请"**);  **if**(request.getLeaveDays()<10){  System.***out***.println(**"审核意见:请假"**+request.getLeaveDays()+**"天,同意"**);  }**else**{  **if**(**null** !=**this**.getNextLeader()){  System.***out***.println(**"审核意见:请假"**+request.getLeaveDays()+**"天,请上级继续审批"**);  **this**.getNextLeader().handleLeaveRequest(request);  }  }  } } **class** Manager **extends** Leader{  **public** Manager(String name){  **super**(name);  }  @Override  **public void** handleLeaveRequest(LeaveRequest request) {  System.***out***.println(**"老总"**+**this**.getName()+**"正在审批"**+request.getEname()+**"的"**+request.getReason()+**"的请假申请"**);  **if**(request.getLeaveDays()<30){  System.***out***.println(**"审核意见:请假"**+request.getLeaveDays()+**"天,同意"**);  }**else**{  System.***out***.println(**"审核意见:请假"**+request.getLeaveDays()+**"天,请本人过来面谈"**);  }  } } |

测试代码

|  |
| --- |
| **package** com.msb.chainRespon;  **public class** Test1 {  **public static void** main(String[] args) {  *// 实例化一个请求* LeaveRequest lr =**new** LeaveRequest(**"小明"**,20,**"回家看媳妇儿"**);  *// 准备多个处理者* Leader a =**new** Director(**"张三"**);  Leader b =**new** ViceManager(**"李四"**);  Leader b2 =**new** ManagerAssis(**"小红"**);  Leader c =**new** Manager(**"王五"**);   *//组织处理者的关系* a.setNextLeader(b);  b.setNextLeader(b2);  b2.setNextLeader(c);   *// 处理请求* a.handleRequest(lr);  } } */\*一个具体的请求\*/* **class** LeaveRequest{  *// 请假人员* **private** String **ename**;  *// 请假天数* **private int leaveDays**;  *// 请假原因* **private** String **reason**;  **public** LeaveRequest() {  }   **public** LeaveRequest(String ename, **int** leaveDays, String reason) {  **this**.**ename** = ename;  **this**.**leaveDays** = leaveDays;  **this**.**reason** = reason;  }   **public** String getEname() {  **return ename**;  }   **public void** setEname(String ename) {  **this**.**ename** = ename;  }   **public int** getLeaveDays() {  **return leaveDays**;  }   **public void** setLeaveDays(**int** leaveDays) {  **this**.**leaveDays** = leaveDays;  }   **public** String getReason() {  **return reason**;  }   **public void** setReason(String reason) {  **this**.**reason** = reason;  } }  */\*一个抽象的处理者\*/* **abstract class** Leader{  *// 处理者名字* **private** String **name**;  *// 下一个处理者 最后的处理者的该属性为null* **private** Leader **nextLeader**;  *// 处理请求的核心方法* **public abstract void** handleRequest(LeaveRequest request);   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public** Leader getNextLeader() {  **return nextLeader**;  }   **public void** setNextLeader(Leader nextLeader) {  **this**.**nextLeader** = nextLeader;  }   **public** Leader(String name) {  **this**.**name** = name;  }   **public** Leader(Leader nextLeader) {  **this**.**nextLeader** = nextLeader;  }   **public** Leader(String name, Leader nextLeader) {  **this**.**name** = name;  **this**.**nextLeader** = nextLeader;  }   **public** Leader() {  } }  */\*多个实际的处理者\*/* **class** Director **extends** Leader{  **public** Director(String name) {  **super**(name);  }   **public** Director(Leader nextLeader) {  **super**(nextLeader);  }   **public** Director(String name, Leader nextLeader) {  **super**(name, nextLeader);  }   **public** Director() {  }   @Override  **public void** handleRequest(LeaveRequest request) {  System.***out***.println(**"主任"**+**this**.getName()+**"正在处理"**+request.getEname()+**"的因"**+request.getReason()+**"请假"**+request.getLeaveDays()+**"天的请求"**);  **if**(request.getLeaveDays()<3){  System.***out***.println(**"请假小于3天,同意"**);  }**else**{  System.***out***.println(**"请假不少于3天,请上级继续审批"**);  **if**(**null** !=**this**.getNextLeader()){  **this**.getNextLeader().handleRequest(request);  }  }  } }  **class** ViceManager **extends** Leader{  **public** ViceManager(String name) {  **super**(name);  }   **public** ViceManager(Leader nextLeader) {  **super**(nextLeader);  }   **public** ViceManager(String name, Leader nextLeader) {  **super**(name, nextLeader);  }   **public** ViceManager() {  }   @Override  **public void** handleRequest(LeaveRequest request) {  System.***out***.println(**"副总"**+**this**.getName()+**"正在处理"**+request.getEname()+**"的因"**+request.getReason()+**"请假"**+request.getLeaveDays()+**"天的请求"**);  **if**(request.getLeaveDays()<10){  System.***out***.println(**"请假小于10天,同意"**);  }**else**{  System.***out***.println(**"请假不少于10天,请上级继续审批"**);  **if**(**null** !=**this**.getNextLeader()){  **this**.getNextLeader().handleRequest(request);  }  }  } }  **class** ManagerAssis **extends** Leader{  **public** ManagerAssis(String name) {  **super**(name);  }   **public** ManagerAssis(Leader nextLeader) {  **super**(nextLeader);  }   **public** ManagerAssis(String name, Leader nextLeader) {  **super**(name, nextLeader);  }   **public** ManagerAssis() {  }   @Override  **public void** handleRequest(LeaveRequest request) {  System.***out***.println(**"老总助理"**+**this**.getName()+**"正在处理"**+request.getEname()+**"的因"**+request.getReason()+**"请假"**+request.getLeaveDays()+**"天的请求"**);  **if**(request.getLeaveDays()<20){  System.***out***.println(**"请假小于20天,同意"**);  }**else**{  System.***out***.println(**"请假不少于20天,请上级继续审批"**);  **if**(**null** !=**this**.getNextLeader()){  **this**.getNextLeader().handleRequest(request);  }  }  } }  **class** Manager **extends** Leader{  **public** Manager(String name) {  **super**(name);  }   **public** Manager(Leader nextLeader) {  **super**(nextLeader);  }   **public** Manager(String name, Leader nextLeader) {  **super**(name, nextLeader);  }   **public** Manager() {  }   @Override  **public void** handleRequest(LeaveRequest request) {  System.***out***.println(**"老总"**+**this**.getName()+**"正在处理"**+request.getEname()+**"的因"**+request.getReason()+**"请假"**+request.getLeaveDays()+**"天的请求"**);  **if**(request.getLeaveDays()<30){  System.***out***.println(**"请假小于30天,同意"**);  }**else**{  System.***out***.println(**"请假不少于30天,请申请人过来面谈"**);  }  } } |

实际开发中的应用:过滤器Filter

### 4.2 观察者模式



何谓观察者模式？观察者模式定义了对象之间的一对多依赖关系，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并且自动更新。

生活举例:

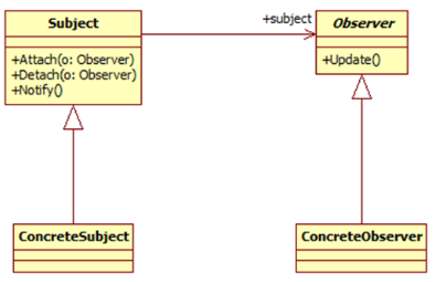
1微信公众号向我们很多人同时推送消息

2 天气预报开始播放,我们收看天气预报

3上课铃声响了,所有学生进入教室听课

… …

在这里，发生改变的对象称之为观察目标，而被通知的对象称之为观察者。一个观察目标可以对应多个观察者，而且这些观察者之间没有相互联系，所以么可以根据需要增加和删除观察者，使得系统更易于扩展。所以观察者提供了一种对象设计，让主题和观察者之间以松耦合的方式结合。



观察者模式包含如下角色：

1 Subject: 目标(抽象的被观察物)  
2 ConcreteSubject: 具体目标(具体的观察物)  
3 Observer: 抽象的观察者  
4 ConcreteObserver: 具体观察者

举例:天气预报节目向所有的订阅者发送天气信息

1 Subject: 目标(抽象的被观察物)

|  |
| --- |
| */\*\*  \* 一个抽象的被观察的主题  \*/* **public interface** Subject {  */\*\*  \* 注册观察者  \** ***@param observer*** *\*/* **public void** registObserver(Observer observer);   */\*\*  \* 删除观察者  \** ***@param observer*** *\*/* **public void** removeObserver(Observer observer);   */\*\*  \* 通知观察者  \*/* **public void** nofityObserver(); } |

2 ConcreteSubject: 具体目标(具体的观察物)

|  |
| --- |
| **import** java.util.ArrayList; **import** java.util.List;  **public class** WeatherSubject **implements** Subject {  *// 所有观察者列表* **private** List<Observer> **observers**;  {  **observers**=**new** ArrayList<Observer>();  }  @Override  **public void** registObserver(Observer observer) {  **if**(**null** !=observer){  **observers**.add(observer);  }  }   @Override  **public void** removeObserver(Observer observer) {  **if**(**null** != observer){  **int** i = **observers**.indexOf(observer);  **if**(i>=0){  **observers**.remove(i);  }  }  }   @Override  **public void** nofityObserver() {  **int** i =(**int**)(Math.*random*()\*3+1);  String message =**""**;  **switch** (i){  **case** 1:  message=**"下雨"**;  **break**;  **case** 2:  message=**"下雪"**;  **break**;  **case** 3:  message=**"下冰雹"**;  **break**;  **default**:  message=**"晴天"**;  }  **for**(Observer ob:**observers**){  ob.update(**new** Event(message,**this**));  }  } } |

3 Observer: 抽象的观察者

|  |
| --- |
| **public interface** Observer {  */\*\*  \* 关注的主体发送消息时要调用观察者的方法  \* 通知观察者的方法  \** ***@param event*** *\*/* **public void** update(Event event); } |

4 ConcreteObserver: 具体观察者

|  |
| --- |
| **public class** PersonObserver **implements** Observer {  *// 观察者名称* **private** String **name**;  *// 观察这所关注的主体* **private** Subject **subject**;   **public** PersonObserver() {  }   **public** PersonObserver(String name, Subject subject) {  **this**.**name** = name;  **this**.**subject** = subject;  *// 将当前对象注册到主题的通知列表* **this**.**subject**.registObserver(**this**);  }   @Override  **public void** update(Event event) {  System.***out***.println(**name**+**"观察到今天要"**+event.getMessage());  } } |

5 一个用户传递信息的事件类

|  |
| --- |
| */\*\*  \* 观察的主体通知观察者时,将信息封装如至该类对象并传递给观察者  \*/* **public class** Event {  **private** String **message**;  **private** Subject **resource**;   **public** String getMessage() {  **return message**;  }   **public void** setMessage(String message) {  **this**.**message** = message;  }   **public** Subject getResource() {  **return resource**;  }   **public void** setResource(Subject resource) {  **this**.**resource** = resource;  }   **public** Event() {  }   **public** Event(String message, Subject resource) {  **this**.**message** = message;  **this**.**resource** = resource;  } } |

6 测试代码

|  |
| --- |
| **public class** Test1 {  **public static void** main(String[] args) {  Subject subject=**new** WeatherSubject();  Observer ob1 =**new** PersonObserver(**"张三"**,subject);  Observer ob2 =**new** PersonObserver(**"李四"**,subject);  Observer ob3 =**new** PersonObserver(**"王五"**,subject);  Observer ob4 =**new** PersonObserver(**"赵六"**,subject);  *//发送消息* subject.nofityObserver();  *// 取消关注* subject.removeObserver(ob1);  subject.removeObserver(ob2);  *// 再次发送消息* subject.nofityObserver();   }  } |

# 说一说代码块以及代码块和构造方法的执行顺序

在java中用{}括起来的称为代码块，代码块可分为以下四种：

**1.普通代码块：**

类中方法的方法体

**2.构造代码块**：

构造块会在创建对象时被调用，每次创建时都会被调用，优先于类构造函数执行。

**3.静态代码块：**

用static{}包裹起来的代码片段，只会执行一次。静态代码块优先于构造块执行。

**4.同步代码块：**

使用synchronized（）{}包裹起来的代码块，在多线程环境下，对共享数据的读写操作是需要互斥进行的，否则会导致数据的不一致性。同步代码块需要写在方法中。

继承下执行顺序测试

|  |
| --- |
|  |

内部类下代码块测试

|  |
| --- |
|  |

# 如何确定垃圾

引用计数法

在 Java 中，引用和对象是有关联的。如果要操作对象则必须用引用进行。因此，很显然一个简单的办法是通过引用计数来判断一个对象是否可以回收。简单说，即一个对象如果没有任何与之关联的引用，即他们的引用计数都不为 0，则说明对象不太可能再被用到，那么这个对象就是可回收对象。

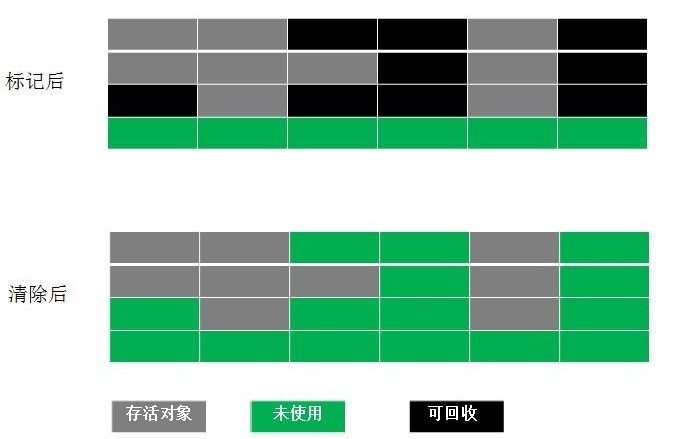
可达性分析

为了解决引用计数法的循环引用问题，Java 使用了可达性分析的方法。通过一系列的“GC roots” 对象作为起点搜索。如果在“GC roots”和一个对象之间没有可达路径，则称该对象是不可达的。要注意的是，不可达对象不等价于可回收对象，不可达对象变为可回收对象至少要经过两次标记过程。两次标记后仍然是可回收对象，则将面临回收。

# 如何清除垃圾

## 标记清除算法（Mark-Sweep）

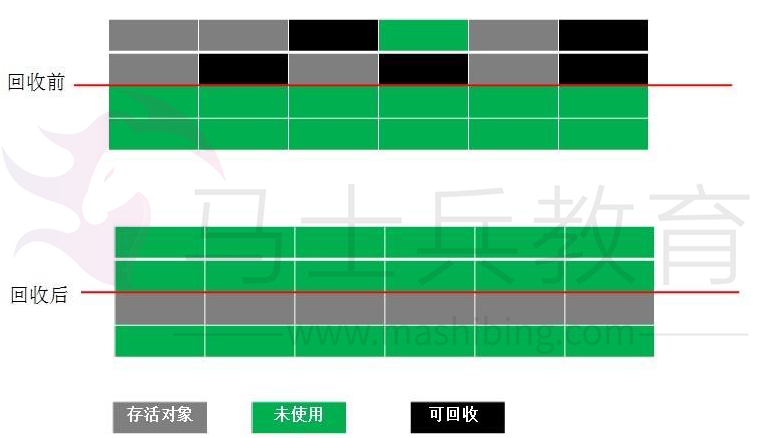
最基础的垃圾回收算法，分为两个阶段，标注和清除。标记阶段标记出所有需要回收的对象，清除阶段回收被标记的对象所占用的空间。如图



我们就可以发现，该算法最大的问题是内存碎片化严重，后续可能发生大对象不能找到可利用空间的问题。

## 复制算法（copying）

为了解决 Mark-Sweep 算法内存碎片化的缺陷而被提出的算法。按内存容量将内存划分为等大小的两块。每次只使用其中一块，当这一块内存满后将尚存活的对象复制到另一块上去，把已使用的内存清掉，如图：



算法虽然实现简单，内存效率高，不易产生碎片，但是最大的问题是可用内存被压缩到了原本的一半。且存活对象增多的话，Copying 算法的效率会大大降低。

## 标记整理算法(Mark-Compact)

结合了以上两个算法，为了避免缺陷而提出。标记阶段和 Mark-Sweep 算法相同，标记后不是清理对象，而是将存活对象移向内存的一端。然后清除端边界外的对象。如图：



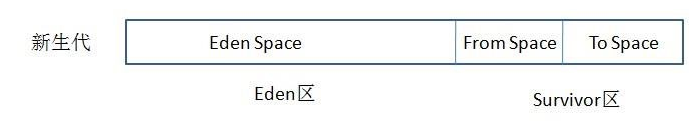
## 分代收集算法

分代收集法是目前大部分 JVM 所采用的方法，其核心思想是根据对象存活的不同生命周期将内存划分为不同的域，一般情况下将 GC 堆划分为老生代(Tenured/Old Generation)和新生代(Young Generation)。老生代的特点是每次垃圾回收时只有少量对象需要被回收，新生代的特点是每次垃圾回收时都有大量垃圾需要被回收，因此可以根据不同区域选择不同的算法。

### 新生代与复制算法

目前大部分 JVM 的 GC 对于新生代都采取 Copying 算法，因为新生代中每次垃圾回收都要回收大部分对象，即要复制的操作比较少，但通常并不是按照 1：1 来划分新生代。一般将新生代划分为一块较大的 Eden 空间和两个较小的 Survivor 空间(From Space, To Space)，每次使用

Eden 空间和其中的一块 Survivor 空间，当进行回收时，将该两块空间中还存活的对象复制到另一块 Survivor 空间中。



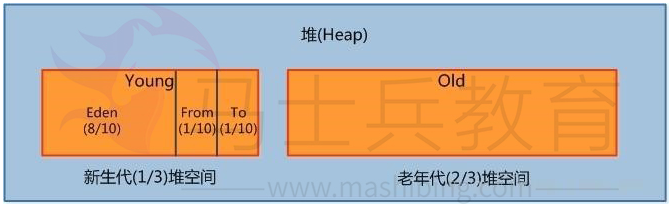
### 老年代与标记复制算法

而老年代因为每次只回收少量对象，因而采用 Mark-Compact 算法。

1. JAVA 虚拟机提到过的处于方法区的永生代(Permanet Generation)，它用来存储 class 类，常量，方法描述等。对永生代的回收主要包括废弃常量和无用的类。
2. 对象的内存分配主要在新生代的 Eden Space 和 Survivor Space 的 From Space(Survivor 目前存放对象的那一块)，少数情况会直接分配到老生代。
3. 当新生代的 Eden Space 和 From Space 空间不足时就会发生一次 GC，进行 GC 后，Eden Space 和 From Space 区的存活对象会被挪到 To Space，然后将 Eden Space 和 From Space 进行清理。
4. 如果 To Space 无法足够存储某个对象，则将这个对象存储到老生代。
5. 在进行 GC 后，使用的便是 Eden Space 和 To Space 了，如此反复循环。
6. 当对象在 Survivor 区躲过一次 GC 后，其年龄就会+1。默认情况下年龄到达 15 的对象会被移到老生代中。

# java堆内存分代

Java 堆从 GC 的角度还可以细分为: 新生代(Eden区、From Survivor区和To Survivor区)和老年代。



## 新生代

是用来存放新生的对象。一般占据堆的1/3空间。由于频繁创建对象，所以新生代会频繁触发

MinorGC 进行垃圾回收。新生代又分为 Eden 区、ServivorFrom、ServivorTo 三个区。

1. Eden 区,Java新对象的出生地（如果新创建的对象占用内存很大，则直接分配到老年代）。当Eden区内存不够的时候就会触发MinorGC，对新生代区进行一次垃圾回收。

2. ServivorFrom,上一次 GC 的幸存者，作为这一次 GC 的被扫描者。

3. ServivorTo 保留了一次 MinorGC 过程中的幸存者。

4. MinorGC 的过程（复制->清空->互换）

首先，把Eden和ServivorFrom区域中存活的对象复制到ServicorTo区域（如果有对象的年龄以及达到了老年的标准，则赋值到老年代区），同时把这些对象的年龄+1（如果 ServicorTo 不够位置了就放到老年区）；

然后，清空 Eden 和 ServicorFrom 中的对象；

最后，ServicorTo 和 ServicorFrom 互换，原 ServicorTo 成为下一次 GC 时的 ServicorFrom 区。

## 老年代

主要存放应用程序中生命周期长的内存对象。

老年代的对象比较稳定，所以 MajorGC 不会频繁执行。在进行 MajorGC 前一般都先进行了一次 MinorGC，使得有新生代的对象晋身入老年代，导致空间不够用时才触发。当无法找到足够大的连续空间分配给新创建的较大对象时也会提前触发一次 MajorGC 进行垃圾回收腾出空间。

MajorGC 采用标记清除算法：首先扫描一次所有老年代，标记出存活的对象，然后回收没有标记的对象。MajorGC 的耗时比较长，因为要扫描再回收。MajorGC 会产生内存碎片，为了减少内存损耗，我们一般需要进行合并或者标记出来方便下次直接分配。当老年代也满了装不下的时候，就会抛出 OOM（Out of Memory）异常。

## 永久代

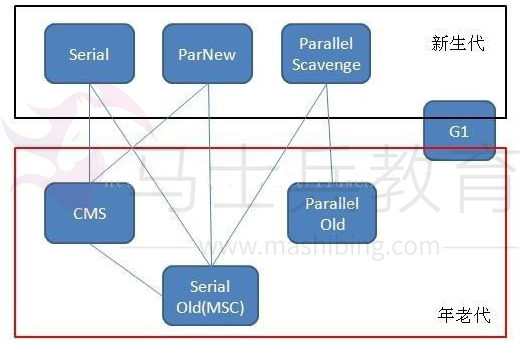
指内存的永久保存区域，主要存放 Class 和 Meta（元数据）的信息,Class 在被加载的时候被放入永久区域，它和和存放实例的区域不同,GC 不会在主程序运行期对永久区域进行清理。所以这也导致了永久代的区域会随着加载的 Class 的增多而胀满，最终抛出 OOM 异常。

## JAVA8与元数据

在Java8中，永久代已经被移除，被一个称为“元数据区”（元空间）的区域所取代。元空间的本质和永久代类似，元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制。类的元数据放入 native memory, 字符串池和类的静态变量放入 java 堆中，这样可以加载多少类的元数据就不再由MaxPermSize 控制, 而由系统的实际可用空间来控制。

# GC垃圾收集器

Java 堆内存被划分为新生代和年老代两部分，新生代主要使用复制和标记-清除垃圾回收算法年老代主要使用标记-整理垃圾回收算法，因此 java 虚拟中针对新生代和年老代分别提供了多种不同的垃圾收集器，JDK1.6 中 Sun HotSpot 虚拟机的垃圾收集器如下：



## Serial 垃圾收集器（单线程、复制算法）

Serial（英文连续）是最基本垃圾收集器，使用复制算法，曾经是JDK1.3.1之前新生代唯一的垃圾收集器。Serial 是一个单线程的收集器，它不但只会使用一个 CPU 或一条线程去完成垃圾收集工作，并且在进行垃圾收集的同时，必须暂停其他所有的工作线程，直到垃圾收集结束。

Serial 垃圾收集器虽然在收集垃圾过程中需要暂停所有其他的工作线程，但是它简单高效，对于限定单个 CPU 环境来说，没有线程交互的开销，可以获得最高的单线程垃圾收集效率，因此 Serial 垃圾收集器依然是 java 虚拟机运行在 Client 模式下默认的新生代垃圾收集器。

## ParNew 垃圾收集器（Serial+多线程）

ParNew 垃圾收集器其实是 Serial 收集器的多线程版本，也使用复制算法，除了使用多线程进行垃圾收集之外，其余的行为和 Serial 收集器完全一样，ParNew垃圾收集器在垃圾收集过程中同样也要暂停所有其他的工作线程。

ParNew 收集器默认开启和 CPU 数目相同的线程数，可以通过-XX:ParallelGCThreads 参数来限制垃圾收集器的线程数。【Parallel：平行的】

ParNew虽然是除了多线程外和Serial收集器几乎完全一样，但是ParNew垃圾收集器是很多java

虚拟机运行在 Server 模式下新生代的默认垃圾收集器。

## Parallel Scavenge 收集器（多线程复制算法、高效）

Parallel Scavenge 收集器也是一个新生代垃圾收集器，同样使用复制算法，也是一个多线程的垃圾收集器，它重点关注的是程序达到一个可控制的吞吐量（Thoughput，CPU 用于运行用户代码的时间/CPU 总消耗时间，即吞吐量=运行用户代码时间/(运行用户代码时间+垃圾收集时间)），高吞吐量可以最高效率地利用 CPU 时间，尽快地完成程序的运算任务，主要适用于在后台运算而不需要太多交互的任务。自适应调节策略也是 ParallelScavenge 收集器与 ParNew 收集器的一个重要区别。

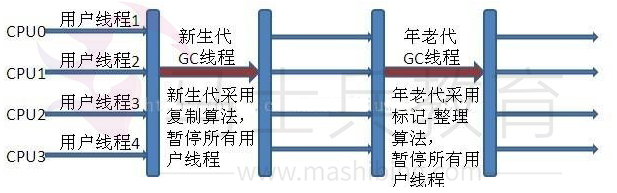
## Serial Old 收集器（单线程标记整理算法 ）

Serial Old 是 Serial 垃圾收集器年老代版本，它同样是个单线程的收集器，使用标记-整理算法，这个收集器也主要是运行在 Client 默认的 java 虚拟机默认的年老代垃圾收集器。

在 Server 模式下，主要有两个用途：

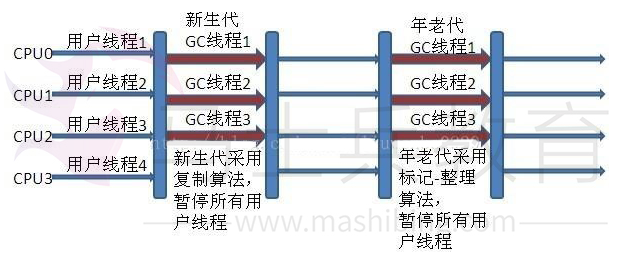
1在 JDK1.5 之前版本中与新生代的 Parallel Scavenge 收集器搭配使用。

2作为年老代中使用 CMS 收集器的后备垃圾收集方案。新生代 Serial 与年老代 Serial Old 搭配垃圾收集过程图：



Parallel Scavenge 收集器与 ParNew 收集器工作原理类似，都是多线程的收集器，都使

用的是复制算法，在垃圾收集过程中都需要暂停所有的工作线程。新生代 Parallel Scavenge/ParNew 与年老代 Serial Old 搭配垃圾收集过程图：



## CMS 收集器（多线程标记清除算法）

Concurrent mark sweep(CMS)收集器是一种年老代垃圾收集器，其最主要目标是获取最短垃圾回收停顿时间，和其他年老代使用标记-整理算法不同，它使用多线程的标记-清除算法。

最短的垃圾收集停顿时间可以为交互比较高的程序提高用户体验。

CMS 工作机制相比其他的垃圾收集器来说更复杂，整个过程分为以下 4 个阶段：

**初始标记**

只是标记一下 GC Roots 能直接关联的对象，速度很快，仍然需要暂停所有的工作线程。

**并发标记**

进行 GC Roots 跟踪的过程，和用户线程一起工作，不需要暂停工作线程。

**重新标记**

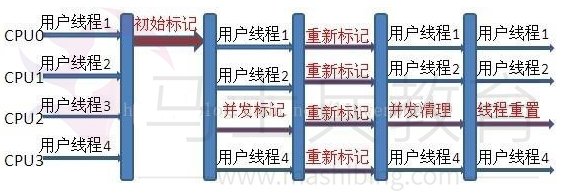
为了修正在并发标记期间，因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，仍然需要暂停所有的工作线程。

**并发清除**

清除 GC Roots 不可达对象，和用户线程一起工作，不需要暂停工作线程。由于耗时最长的并发标记和并发清除过程中，垃圾收集线程可以和用户现在一起并发工作，所以总体上来看

CMS 收集器的内存回收和用户线程是一起并发地执行。

CMS 收集器工作过程：



## G1 收集器

Garbage first 垃圾收集器是目前垃圾收集器理论发展的最前沿成果，相比与 CMS 收集器，G1 收集器两个最突出的改进是：

1. 基于标记-整理算法，不产生内存碎片。
2. 可以非常精确控制停顿时间，在不牺牲吞吐量前提下，实现低停顿垃圾回收。

G1 收集器避免全区域垃圾收集，它把堆内存划分为大小固定的几个独立区域，并且跟踪这些区域的垃圾收集进度，同时在后台维护一个优先级列表，每次根据所允许的收集时间，优先回收垃圾最多的区域。区域划分和优先级区域回收机制，确保 G1 收集器可以在有限时间获得最高的垃圾收集效率。

# 事务

什么是事务: 事务逻辑上的一组操作,组成这组操作的各个逻辑单元,要么一起成功,要么一起失败.

## 事务特性（4种）:

* 原子性 （atomicity）:强调事务的不可分割.
* 一致性 （consistency）:事务的执行的前后数据的完整性保持一致.
* 隔离性 （isolation）:一个事务执行的过程中,不应该受到其他事务的干扰
* 持久性（durability） :事务一旦结束,数据就持久到数据库

## 引发安全性问题:

脏读 :一个事务读到了另一个事务的未提交的数据

不可重复读 :一个事务读到了另一个事务已经提交的 update 的数据导致多次查询结果不一致.

虚幻读 :一个事务读到了另一个事务已经提交的 insert 的数据导致多次查询结果不一致.

## 解决读问题: 设置事务隔离级别

DEFAULT 这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔 离级别.

未提交读（read uncommited） :脏读，不可重复读，虚读都有可能发生

已提交读 （read commited）:避免脏读。但是不可重复读和虚读有可能发生

可重复读 （repeatable read） :避免脏读和不可重复读.但是虚读有可能发生.

串行化的 （serializable） :避免以上所有读问题.

Mysql 默认:可重复读

Oracle 默认:读已提交

## 事务的传播行为

PROPAGION\_XXX :事务的传播行为

* 保证同一个事务中

PROPAGATION\_REQUIRED 支持当前事务，如果不存在 就新建一个(默认)

PROPAGATION\_SUPPORTS 支持当前事务，如果不存在，就不使用事务

PROPAGATION\_MANDATORY 支持当前事务，如果不存在，抛出异常

* 保证没有在同一个事务中

PROPAGATION\_REQUIRES\_NEW 如果有事务存在，挂起当前事务，创建一个新的事务

PROPAGATION\_NOT\_SUPPORTED 以非事务方式运行，如果有事务存在，挂起当前事务

PROPAGATION\_NEVER 以非事务方式运行，如果有事务存在，抛出异常

PROPAGATION\_NESTED 如果当前事务存在，则嵌套事务执行

# Mybatis一级缓存原理

第一次发出一个查询 sql，sql 查询结果写入 sqlsession 的一级缓存中，缓存使用的数据结构是一个 map。

key：MapperID+offset+limit+Sql+所有的入参

value：用户信息

同一个 sqlsession 再次发出相同的 sql，就从缓存中取出数据。如果两次中间出现 commit 操作（修改、添加、删除），本 sqlsession 中的一级缓存区域全部清空，下次再去缓存中查询不到所以要从数据库查询，从数据库查询到再写入缓存。

# Mybatis二级缓存原理

二级缓存的范围是mapper 级别（mapper同一个命名空间），mapper以命名空间为单位创建缓存数据结构，结构是 map。mybatis 的二级缓存是通过 CacheExecutor 实现的。CacheExecutor 其实是 Executor 的代理对象。所有的查询操作，在 CacheExecutor 中都会先匹配缓存中是否存在，不存在则查询数据库。

key：MapperID+offset+limit+Sql+所有的入参

具体使用需要配置：

1. Mybatis 全局配置中启用二级缓存配置
2. 在对应的 Mapper.xml 中配置 cache 节点
3. 在对应的 select 查询节点中添加 useCache=true

