# Java Web学习笔记

## Java基础

① **静态代码块、构造代码块，构造方法的执行顺序**：

父类静态代码块→子类静态代码块→父类构造代码块→父类构造方法→子类构造代码块→子类构造方法

**② final关键字**

final是一个关键字，可以用于修饰类，成员变量，成员方法。

**特点**：

1. 它修饰的类不能被继承。
2. 它修饰的成员变量是一个常量。
3. 它修饰的成员方法是不能被子类重写的。

final修饰的常量定义一般都有书写规范,被final修饰的常量名称,所有字母都**大写**。

final修饰成员变量,必须初始化,初始化有两种

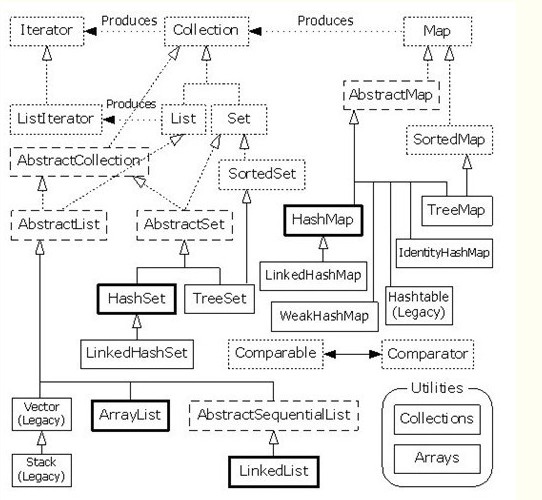
* 显示初始化；
* 构造方法初始化。

但是不能两个一起初始化

③ **final和private的区别**

1. final修饰的类可以访问；  
   private不可以修饰外部类，但可以修饰内部类（其实把外部类私有化是没有意义的）。
2. final修饰的方法不可以被子类重写；  
   private修饰的方法表面上看是可以被子类重写的，其实不可以，子类是看不到父类的私有方法的。
3. final修饰的变量只能在显示初始化或者构造函数初始化的时候赋值一次，以后不允许更改；  
   private修饰的变量，也不允许直接被子类或一个包中的其它类访问或修改，但是他可以通过set和get方法对其改值和取值。

### Java 集合



#### ① Collection接口

Collection接口是 Set 、List 和 Queue 接口的父接口，提供了多数集合常用的方法声明，包括 add()、remove()、contains() 、size() 、iterator() 等。

1. List

List 关心的是索引，与其他集合相比，List特有的就是和索引相关的一些方法：get(int index) 、 add(int index,Object o) 、 indexOf(Object o) ；List是有序的Collection，使用此接口能够精确的控制每个元素插入的位置。用户能够使用索引（元素在List中的位置，类似于数组下标）来访问List中的元素，这类似于Java的数组；另外List允许有相同的元素。

**ArrayList** 可以将它理解成一个可增长的数组，默认初始大小为10. 成1.5倍加1增长。它提供快速迭代和快速随机访问的能力，其底层使用数组实现的，查询效率较高，但是增删效率低且是线程不安全的。

**LinkedList** 中的元素之间是双链接的，当需要快速插入和删除时LinkedList成为List中的不二选择，其底层使用双向循环链表实现，查询效率低，但增删元素效率比较高。

**Vector** 是ArrayList的线程安全版本，性能比ArrayList要低，现在已经很少使用。

1. Set

**Set关心唯一性，它不允许重复。**

HashSet**当不希望集合中有重复值，并且不关心元素之间的顺序时可以使用此类，使用哈希**[算法](http://lib.csdn.net/base/datastructure)**实现。**

LinkedHashset**当不希望集合中有重复值，并且希望按照元素的插入顺序进行迭代遍历时可采用此类。**

TreeSet**当不希望集合中有重复值，并且希望按照元素的自然顺序进行排序时可以采用此类。（自然顺序意思是某种和插入顺序无关，而是和元素本身的内容和特质有关的排序方式，譬如“abc”排在“abd”前面。）**

1. Queue

Queue用于保存将要执行的任务列表。

**LinkedList** 同样实现了Queue接口，可以实现先进先出的队列。

**PriorityQueue** 用来创建自然排序的优先级队列。

#### ② Map接口

Map关心的是唯一的标识符。他将唯一的键映射到某个元素。当然键和值都是对象。

**HashMap** 当需要键值对表示，又不关心顺序时可采用HashMap，线程不安全，允许key或value为NULL。

**Hashtable** 注意Hashtable中的t是小写的，它是HashMap的线程安全版本，不允许key或value为NULL。

**LinkedHashMap** 当需要键值对，并且关心插入顺序时可采用它。

**TreeMap** 当需要键值对，并关心元素的自然排序时可采用它。

### JAVA 枚举（Enum）



enum是特殊的class，实际上enum声明定义的类型就是一个类。 是 (java.lang.Enum<E>)的子类

1、Color枚举类就是class，而且是一个不可以被继承的final类。其枚举值(RED,BLUE...)都是Color类型的类静态常量， 我们可以通过下面的方式来得到Color枚举类的一个实例：

Color c=Color.RED;

注意：这些枚举值都是public static final的，也就是我们经常所定义的常量方式，因此枚举类中的枚举值最好全部大写。

2、即然枚举类是class，当然在枚举类型中有构造器，方法和数据域。但是，枚举类的构造器有很大的不同：

(1) **构造器只是在构造枚举值的时候被调用**。

(2) **构造器只能私有private，绝对不允许有public构造器**。 这样可以保证外部代码无法新构造枚举类的实例。这也是完全符合情理的，因为我们知道枚举值是public static final的常量而已。但枚举类的方法和数据域可以允许外部访问。

3、所有枚举类都继承了Enum的方法:  
       **(1)  ordinal()方法**: 返回枚举值在枚举类种的顺序。这个顺序根据枚举值声明的顺序而定。          Color.RED.ordinal();  //返回结果：0

                 Color.BLUE.ordinal();  //返回结果：1

**(2)  compareTo()方法**: Enum实现了java.lang.Comparable接口，因此可以比较对象与指定对象的顺序。Enum中的compareTo返回的是两个枚举值的顺序之差。当然，前提是两个枚举值必须属于同一个枚举类，否则会抛出ClassCastException()异常。(具体可见源代码)  
                 Color.RED.compareTo(Color.BLUE);  //返回结果 -1

**(3)  values()方法**： 静态方法，返回一个包含全部枚举值的数组。  
                 Color[] colors = Color.values();

                 for(Color c:colors){

                        System.out.print(c+",");

                 }//返回结果：RED,BLUE,BLACK YELLOW,GREEN,

  (4)  **toString()方法**： 返回枚举常量的名称。

                 Color c = Color.RED;

                 System.out.println(c);//返回结果: RED

 (5)  **valueOf()方法**： 这个方法和toString方法是相对应的，返回带指定名称的指定枚举类型的枚举常量。

                 Color.valueOf("BLUE");   //返回结果: Color.BLUE

     (6)  **equals()方法**： 比较两个枚举类对象的引用。

### JAVA 泛型

### JAVA 注解（Annotation）

#### 什么是注解

注解（Annotation），也叫（metadata）元数据。一种代码级别的说明。它是JDK1.5及以后版本引入的一个特性，与类、接口、枚举是在同一个层次。它可以声明在包、类、字段、方法、局部变量、方法参数等的前面，用来对这些元素进行说明，注释。这些信息被存储在Annotation的“name=value”结构对中。

#### 什么是（metadata）元数据

“数据的数据”。那它有什么作用呢，如果要对于元数据的作用进行分类，还没有明确的定义，不过我们可以根据它所起的作用，大致可分为三类：

编写文档：通过代码里标识的元数据生成文档。

代码分析：通过代码里标识的元数据对代码进行分析。

编译检查：通过代码里标识的元数据让编译器能实现基本的编译检查

#### 注解的分类

根据注解的参数个数分类：

* 标记注解。一个没有成员的 Annotation，这种类型仅仅使用自身的存在与否来为我们提供信息。标记注解非常常见，比如上面所说的 @Override
* 单值注解。成员的参数为单个参数
* 完整注解。成员的参数为多个参数

根据注解使用的方法和用途分类：

* JDK内置系统注解
* 元注解
* 自定义注解

##### 内置注解

* @Override
* @Deprecated
* @SuppressWarnings

**1、@Override**

它的作用是对覆盖超类中方法的方法进行标记，如果被标记的方法并没有实际覆盖超类中的方法，则编译器会发出错误警告。换句话理解就是重写父类方法，方法前的标记。

@Override

protected void onStart() {

super.onStart();

}

**2、@Deprecated**

它的作用是对不应该再使用的方法添加注解，当编程人员使用这些方法时，将会在编译时显示提示信息，它与javadoc里的@deprecated标记有相同的功能。

public class User {

@Deprecated

public static String getName(){

return "user";

}

}

**调用 getName 方法**：

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

//@Deprecated 方法已经过时 出现中划线

User.getName();

}

}

**3、@SuppressWarnings**

其参数有：

* deprecation 使用了过时的类或方法时的警告
* unchecked 执行了未检查的转换时的警告
* fallthrough 当 switch 程序块直接通往下一种情况而没有 break 时的警告
* path 在类路径、源文件路径等中有不存在的路径时的警告
* serial 当在可序列化的类上缺少serialVersionUID 定义时的警告
* finally 任何 finally 子句不能正常完成时的警告
* all 关于以上所有情况的警告

public class User {

@SuppressWarnings("deprecation")

public static int getAge() {

return 18;

}

}

##### 元注解

元注解就是定义注解的注解，由 java API 提供，分别有四个元注解：

* @Target
* @Retention
* @Documented
* @Inherited

**1、@Target**

用于描述注解的使用范围。修饰的对象范围：packages、types（类、接口、枚举、Annotation类型）、类型成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如 catch 参数）。

它的值在枚举类 ElemenetType 中：

* CONSTRUCTOR： 用于描述构造器
* FIELD ： 用于描述域
* LOCAL\_VARIABLE： 用于描述局部变量
* METHOD ： 用于描述方法
* PACKAGE ： 用于描述包
* PARAMETER ： 用于描述参数
* TYPE ： 用于描述类、接口(包括注解类型) 或enum声明

Mode类可以注解类的成员变量：

@Target(ElementType.FIELD)

@Documented

public @interface Mode {

public int value() default 0;

}

Person可以注解类、接口（包括注解类型）、或者enum声明：

@Target(ElementType.TYPE)

public @interface People {

public String value() default "";

}

**2、@Retention**

表示需要在什么级别保存该注释信息，用于描述注解的生命周期（即：被描述的注解在什么范围内有效），定义了该Annotation被保留的时间长短：某些Annotation仅出现在源代码中，而被编译器丢弃；而另一些却被编译在class文件中；编译在class文件中的Annotation可能会被虚拟机忽略，而另一些在class被装载时将被读取。

RetentionPoicy取值：

* SOURCE ： 在源文件中有效（即源文件保留）
* CLASS ： 在class文件中有效（即class保留）
* RUNTIME ： 在运行时有效（即运行时保留）

Page 注解的RetentionPolicy 的值为 RUNTIME，这样注解处理器可以通过反射，获取到该注解的属性，从而做一些运行时的逻辑处理。

**3、@Documented**

用于描述其它类型的annotation应该被作为被标注的程序成员的公共API，Documented 是一个标记注解，没有成员。将此注解包含在 javadoc 中 ，它代表着此注解会被javadoc工具提取成文档。在doc文档中的内容会因为此注解的信息内容不同而不同。

@Documented

public @interface Page {

public int value() default 0;

}

**4、@Inherited**

允许子类继承父类中的注解，是一个标记注解，@Inherited阐述了某个被标注的类型是被继承的。如果一个使用了@Inherited修饰的annotation类型被用于一个class，则这个annotation将被用于该class的子类。

@Inherited annotation类型是被标注过的class的子类所继承。类并不从它所实现的接口继承annotation，方法并不从它所重载的方法继承annotation

当@Inherited annotation类型标注的annotation的Retention是RetentionPolicy.RUNTIME，则反射API增强了这种继承性。如果我们使用java.lang.reflect去查询一个@Inherited annotation类型的annotation时，反射代码检查将展开工作：检查class和其父类，直到发现指定的annotation类型被发现，或者到达类继承结构的顶层。

##### 自定义注解

使用@interface自定义注解时，自动继承了java.lang.annotation.Annotation接口，由编译程序自动完成其他细节。在定义注解时，不能继承其他的注解或接口。@interface用来声明一个注解，其中的每一个方法实际上是声明了一个配置参数。方法的名称就是参数的名称，返回值类型就是参数的类型（返回值类型只能是基本类型、Class、String、enum）。可以通过default来声明参数的默认值。以上所有例子都属于自定义注解。自定义注解具有以下固定格式：

public @interface 注解名{注解体}

* 所有基本数据类型（int,float,boolean,byte,double,char,long,short)
* String类型
* Class类型
* enum类型
* Annotation类型
* 以上所有类型的数组

注意：只能有public或默认(default)这两个访问权修饰，参数成员只能用以上6种类型，如果只有一个参数成员，最好把参数名称设为”value”。

**Shade 形状注解**：

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @interface Shade {

public enum ShadeType {

Triangle("三角"),

Four("四边"),

Five("五角");

private String type;

ShadeType(String type) {

this.type = type;

}

@Override

public String toString() {

return type;

}

}

public ShadeType shader() default ShadeType.Triangle;

}

**读取注解**

这里我们将使用反射去读取注解。Java在java.lang.reflect 包下新增了AnnotatedElement接口，该接口代表程序中可以接受注解的程序元素。该接口主要有如下几个方法：

default boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)

判断该程序元素上是否包含指定类型的注解，存在则返回true，否则返回false。

<T extends Annotation> T getAnnotation(Class<T> var1);

返回该程序元素上存在的、指定类型的注解，如果该类型注解不存在，则返回null。

Annotation[] getAnnotations();

返回该程序元素上存在的所有注解。

Annotation[] getDeclaredAnnotations();

### JAVA 异常（Exception）

**① JAVA异常结构**：



**② 异常说明：**

**Throwable：**有两个重要的子类：Exception（异常）和 Error（错误）

**Error（错误）:** 是程序无法处理的错误，表示运行应用程序中较严重问题，Java虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError

**Exception（异常）:**是程序本身可以处理的异常。Java的异常(包括Exception和Error)分为**可查的异常（checked exceptions）和不可查的异常（unchecked exceptions）**。

**运行时异常：**都是RuntimeException类及其子类异常，如NullPointerException(空指针异常)、IndexOutOfBoundsException(下标越界异常)等，这些异常是不检查异常

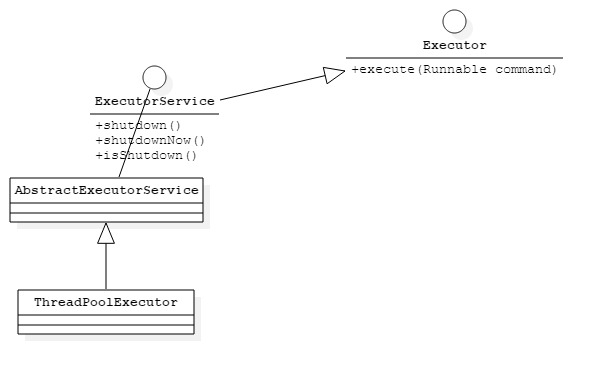
**非运行时异常 （编译异常）：**是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常。

### JAVA IO、NIO

### JAVA 线程（Thread）

## JAVA 高级

### JAVA线程池



Java通过Executors提供四种线程池，分别为：  
**newCachedThreadPool**创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。  
**newFixedThreadPool** 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

**newScheduledThreadPool** 创建一个定长线程池，支持定时及周期性任务执行。  
**newSingleThreadExecutor** 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

#### newCachedThreadPool

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。示例代码如下：

1. ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
2. **for** (**int** i = 0; i < 10; i++) {
3. **final** **int** index = i;
4. **try** {
5. Thread.sleep(index \* 1000);
6. } **catch** (InterruptedException e) {
7. e.printStackTrace();
8. }
10. cachedThreadPool.execute(**new** Runnable() {
12. @Override
13. **public** **void** run() {
14. System.out.println(index);
15. }
16. });
17. }

线程池为无限大，当执行第二个任务时第一个任务已经完成，会复用执行第一个任务的线程，而不用每次新建线程。

#### newFixedThreadPool

创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。示例代码如下:

1. ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);
2. **for** (**int** i = 0; i < 10; i++) {
3. **final** **int** index = i;
4. fixedThreadPool.execute(**new** Runnable() {
6. @Override
7. **public** **void** run() {
8. **try** {
9. System.out.println(index);
10. Thread.sleep(2000);
11. } **catch** (InterruptedException e) {
12. // TODO Auto-generated catch block
13. e.printStackTrace();
14. }
15. }
16. });
17. }

因为线程池大小为3，每个任务输出index后sleep 2秒，所以每两秒打印3个数字。  
定长线程池的大小最好根据系统资源进行设置。如Runtime.getRuntime().availableProcessors()。可参考PreloadDataCache。

#### newScheduledThreadPool

创建一个定长线程池，支持定时及周期性任务执行。延迟执行示例代码如下：

1. ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);
2. scheduledThreadPool.schedule(**new** Runnable() {
4. @Override
5. **public** **void** run() {
6. System.out.println("delay 3 seconds");
7. }
8. }, 3, TimeUnit.SECONDS);

表示延迟3秒执行。  
   
定期执行示例代码如下：

scheduledThreadPool.scheduleAtFixedRate(**new** Runnable() {

1. @Override
2. **public** **void** run() {
3. System.out.println("delay 1 seconds, and excute every 3 seconds");
4. }
5. }, 1, 3, TimeUnit.SECONDS);

表示延迟1秒后每3秒执行一次。  
ScheduledExecutorService比Timer更安全，功能更强大。

#### newSingleThreadExecutor

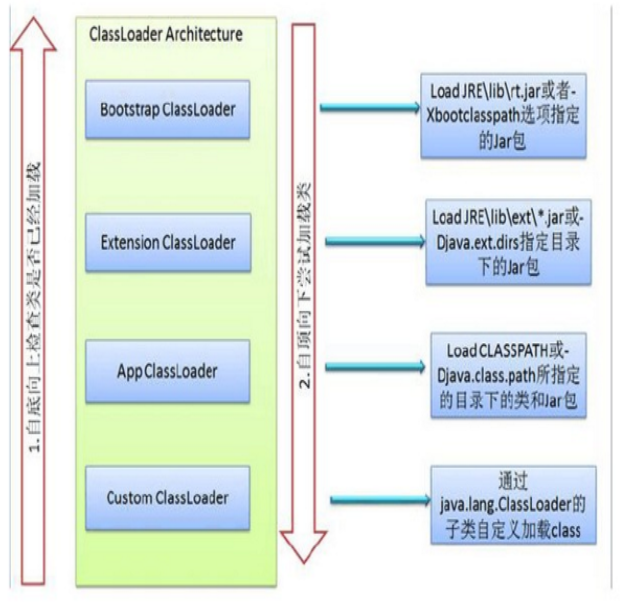
创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。示例代码如下：

1. ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();
2. **for** (**int** i = 0; i < 10; i++) {
3. **final** **int** index = i;
4. singleThreadExecutor.execute(**new** Runnable() {
6. @Override
7. **public** **void** run() {
8. **try** {
9. System.out.println(index);
10. Thread.sleep(2000);
11. } **catch** (InterruptedException e) {
12. // TODO Auto-generated catch block
13. e.printStackTrace();
14. }
15. }
16. });
17. }

结果依次输出，相当于顺序执行各个任务。  
现行大多数GUI程序都是单线程的。Android中单线程可用于数据库操作，文件操作，应用批量安装，应用批量删除等不适合并发但可能IO阻塞性及影响UI线程响应的操作。

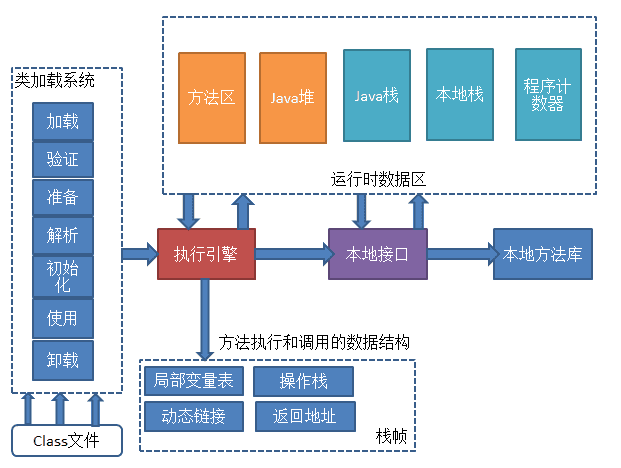
### ****类加载机制****

**双亲委派机制**。类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。



### 反射

### JVM



**① JVM内存模型**

描述主内存和工作内存之间的通信规则，避免数据不一致。所有线程共享JVM内存区域main memory。而每个单独线程又有自己的工作内存。为了保证从工作内存写到主内存的数据的一致性，JVM定义了一系列的规则：

* 所有变量都在主内存中，对所有线程共享（此处的变量与Java编程时所说的变量不一样，指包括了实例字段、静态字段和构成数组对象的元素，但是不包括局部变量与方法参数，后者是线程私有的，不会被共享。）
* 每条线程都有自己的工作内存，保存主内存中变量的拷贝，线程对变量的操作只能在工作内存中完成
* 线程无法直接访问对方的工作内存

**内存之间的8个交互指令**

lock(main) unlock(main) read(main) load(work) use(work) assign(work) store(work) write(main)  
必须顺序执行，不必连续执行

* + 不许read load 和 store write单独出现
  + 不许丢弃assign结果，必须同步回主内存
  + 不许未assign，直接同步主内存
  + 变量只能在主内存诞生。在使用use 和 store前，必须先read和load
  + 一个变量同一时刻只能由一个lock操作，与unlock必须成对出现
  + 如果lock操作，那么这个变量需要重新执行load 和 assign操作
  + 如果没lock，不许unlock
  + unlock前，必须执行store和write

**② JVM内存管理**

Java虚拟机在运行时会把它所管理的内存分为若干不同个数据区域  
主内存: 方法区+堆, 由线程共享  
工作内存：栈 + 程序计数器，线程私有

* 方法区： 存放类信息，常量，静态变量，即时编译后的代码。
* 堆：存放对象，细分为新生代（Eden，From Survivor， To Survivor）+ 老年代, 也可以划分出多个线程私有的分配缓存区TLAB
* 栈：局部变量表 + 操作数栈 + 动态链接 + 方法出参+其他（debug）
* 程序计数器：当前线程所执行的代码的行号指示器

**③ JVM垃圾回收**

对象已死分析：引用计数法 Vs 可达性分析法  
GC roots：栈中引用的对象，类静态变量引用的对象，常量引用的对象，本地方法JNI引用的对象  
垃圾回收算法：复制，标记-整理，标记-清除，分代回收  
垃圾回收时间：安全点，安全区域  
垃圾回收器：  
新生代：serial parNew ParallelScavenge  
老年代：serialOld parallelOld CMS  
G1  
 CMS垃圾回收器：初始标记 -》 并发标记-》重新标记-》并发清除

④ **JVM参数调优**

* 标准参数 （-） -verbose
* 非标准参数 （-X） -Xmx20m  
  -Xmx 堆最大值  
  -Xms 堆最小值  
  -Xmn 新生代内存  
  -Xss 栈内存
* 非稳定参数 （-XX） -XX：SurvivorRatio=8
  + 行为参数：  
    DisableExplicitGC 禁止显示调用system.gc  
    UseConcMarkSweepGC  
    UseSearialGC  
    UseParallelGC
  + 性能调优：  
    PermSize 方法区内存  
    MaxPermSize  
    SurvivorRatio 新生代中Eden和Survivor的容量比值，默认8：1  
    PretenureSizeThreshold 直接晋升到老年代的对象大小阈值  
    MaxTenuringThreshold 晋升到老年代的年龄  
    UseAdaptiveSizePolicy 动态调整Java堆各区域的大小及进入老年代的年龄  
    HandlePromotionFailure 是否允许担保失败  
    ParallelGCThreads GC内存回收的线程数  
    GCTimeRatio GC占总时间的比例，默认99， 只允许1%,只在parallelScavenge生效  
    MaxGCPauseMillis GC最大停顿时间，只在parallelScavenge生效  
    CMSInitiatingOccupancyFraction 老年代在空间占用多少后触发回收，默认68%，只在CMS生效  
    UseCMSCompactionAtFullCollection 是否需要整理，仅在CMS生效  
    CMSFullGCsBeforeCompaction 几次后再整理  
    CompileThreshold JIT编译阈值 client默认1500 server默认10000
  + 调试：  
    printGCDetails 打印内存回收日志  
    HeapDumpOnOutOfMemoryError  
    TraceClassLoading  
    TraceClassUnloading

**⑤ java工具**

jps: 打印java进程  
jstack: 查看线程信息  
jmap: 查看堆信息  
jconsole， jinfo， jhat， javap， btrace

⑥ **HeapOutOfMemory**

当堆上分配的对象大于指定堆的最大值时，抛出该错。  
可以使用-XX:+HeapDumpOnOutOfMemoryError 查看内存快照进行分析

MethodArea OutOfMemory  
方法区内存不足，存放类信息，常量，静态变量，即时编译后的代码，检查这几个信息是否有异常  
大多的原因是因为动态产生过多的类。

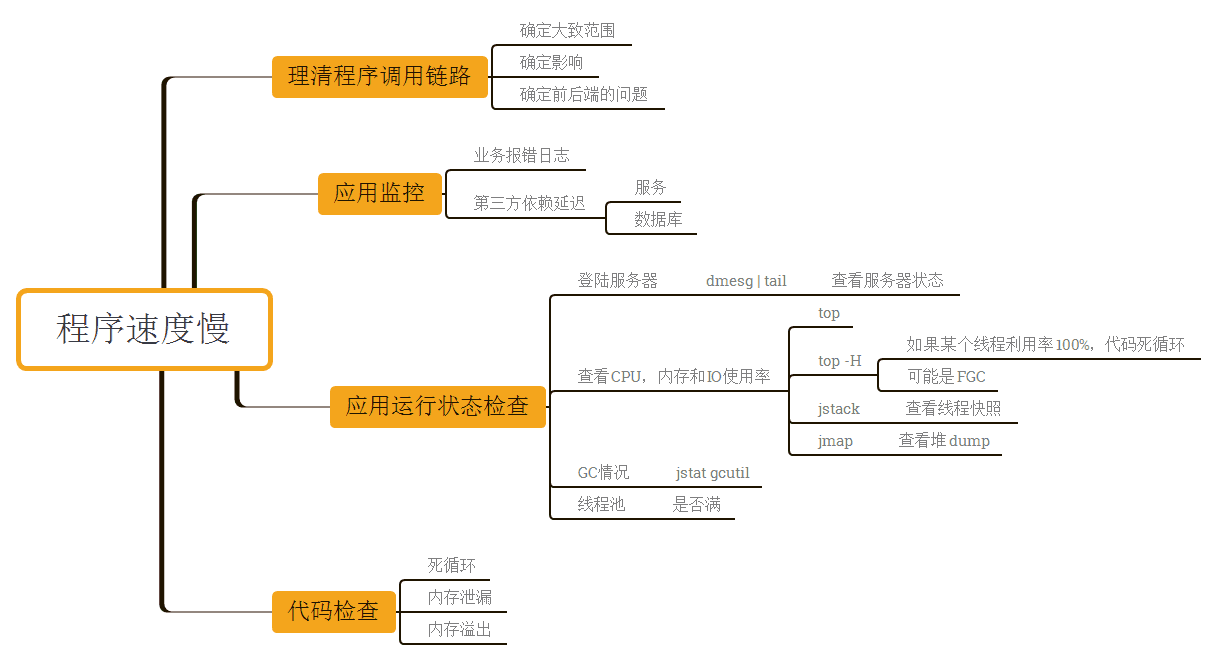
ConstantPool OutOfMemory  
常量池溢出，查看是否intern使用不当

DirectMemory OutOfMemory  
本机直接内存溢出，容量可通过-XX:MaxDirectMemorySize指定，如果不指定，默认和堆最大值相同。  
这个溢出发生在系统进行直接内存分配。例如：unsafe.allocateMemory()  
特征为：OOM后发现Dump问价你很小，程序中直接或间接使用了NIO

Stack OutOfMemory  
扩展栈时无法获取足够的内存空间，在创建线程时  
解决方法之一：减少最大堆

Stack OverFlow  
栈深度大于虚拟机所允许的深度，经常是由于死循环的递归调用

当一个Java程序响应很慢时如何查找问题

****

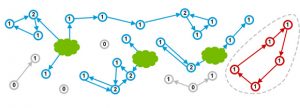
### 垃圾回收

#### 引用计数



绿色的云代表的是程序中仍在使用的对象。蓝色的圆圈代表的是内存中的对象，可以看到有多少对象引用了它们。灰色圆圈的对象是已经没有任何人引用的了。因此，它们属于垃圾对象，可以被垃圾回收器清理掉。

这里个重大的缺陷。很容易会出现一些孤立的环，它们中的对象都不在任何域内，但彼此却互相引用导致引用数不为0。下面便是一个例子：



#### 标记删除

JVM通过标记删除的算法来记录所有可达（存活）对象，同时确保不可达对象的那些内存能够被重用。这包含两个步骤：

* 标记是指遍历所有可达对象，然后在本地内存中记录这些对象的信息
* 删除会确保不可达对象的内存地址可以在下一次内存分配中使用。

缺点就是应用程序的线程需要被暂停才能完成回收，如果引用一直在变的话你是无法进行计数的。这个应用程序被暂停以便JVM可以收拾家务的情况又被称为Stop The World pause(STW)。这种暂停被触发的可能性有很多，不过垃圾回收应该是最常见的一种。

### 设计模式

① **静态代理和动态代理**

动态代理相对于静态代理在使用上的优点主要是能够对一个对象的所有方法进行统一包装，而且后期被代理的类添加方法的时候动态代理类不需要改动。

缺点是要求被代理的类必须实现了接口，因为动态代理类在实现的时候继承了Proxy类，java不支持多继承，因此动态代理类只能根据接口来定义方法。

最后动态代理之所以叫做动态代理是因为java在实现动态代理的时候，动态代理类是在运行时动态生成和加载的，相对的，静态代理类和其他普通类一下，在类加载阶段就加载了。

② **状态模式**

状态模式的角色主要有三个：抽象状态角色，具体状态角色，环境角色（例如：电梯开门、关门）

状态模式的优点：

* 结构清晰，避免了过多的switch…case或者if…else语句的使用
* 遵循设计原则，很多地体现了开闭原则和单一职责原则，每个状态都是一个子类。
* 封装性比较好

状态模式的缺点：

* 类膨胀

### JAVA 并发编程

**① 并发编程的原则和技巧**

 1. **单一职责原则**：分离并发相关代码和其他代码（并发相关代码有自己的开发、修改和调优生命周期）。

  2. **限制数据作用域**：两个线程修改共享对象的同一字段时可能会相互干扰，导致不可预期的行为，解决方案之一是构造临界区，但是必须限制临界区的数量。

  3. **使用数据副本**：数据副本是避免共享数据的好方法，复制出来的对象只是以只读的方式对待。Java 5的java.util.concurrent包中增加一个名为CopyOnWriteArrayList的类，它是List接口的子类型，所以你可以认为它是ArrayList的线程安全的版本，它使用了写时复制的方式创建数据副本进行操作来避免对共享数据并发访问而引发的问题。

  4. **线程应尽可能独立**：让线程存在于自己的世界中，不与其他线程共享数据。有过Java Web开发经验的人都知道，Servlet就是以单实例多线程的方式工作，和每个请求相关的数据都是通过Servlet子类的service方法（或者是doGet或doPost方法）的参数传入的。只要Servlet中的代码只使用局部变量，Servlet就不会导致同步问题。Spring MVC的控制器也是这么做的，从请求中获得的对象都是以方法的参数传入而不是作为类的成员，很明显Struts 2的做法就正好相反，因此Struts 2中作为控制器的Action类都是每个请求对应一个实例。

**② 并发模型**

| **概念** | **解释** |
| --- | --- |
| 临界资源 | 并发环境中有着固定数量的资源 |
| 互斥 | 对资源的访问是排他式的 |
| 饥饿 | 一个或一组线程长时间或永远无法取得进展 |
| 死锁 | 两个或多个线程相互等待对方结束 |
| 活锁 | 想要执行的线程总是发现其他的线程正在执行以至于长时间或永远无法执行 |

**③ 生产者-消费者**

  一个或多个生产者创建某些工作并将其置于缓冲区或队列中，一个或多个消费者会从队列中获得这些工作并完成之。这里的缓冲区或队列是临界资源。当缓冲区或队列放满的时候，生产这会被阻塞；而缓冲区或队列为空的时候，消费者会被阻塞。生产者和消费者的调度是通过二者相互交换信号完成的。

**④ 读者-写者**

  当存在一个主要为读者提供信息的共享资源，它偶尔会被写者更新，但是需要考虑系统的吞吐量，又要防止饥饿和陈旧资源得不到更新的问题。在这种并发模型中，如何平衡读者和写者是最困难的，当然这个问题至今还是一个被热议的问题，恐怕必须根据具体的场景来提供合适的解决方案而没有那种放之四海而皆准的方法。

#### Synchronized

**Synchronized特性总结:**

   - **只能锁定对象，不能锁定基本数据类型。**

   - **被锁定的对象数组中的单个对象不会被锁定。**

   - **同步方法可以视为包含整个方法的Synchronized(this) { ... }代码块。**

   - **静态同步方法会锁定它的Class对象。**

   - **内部类的同步是独立于外部类的。**

   - **synchronized修饰符并不是方法签名的组成部分，所以不能出现在接口的方法声明中。**

   - **非同步的方法不关心锁的状态，它们在同步方法运行时仍然可以得以运行。**

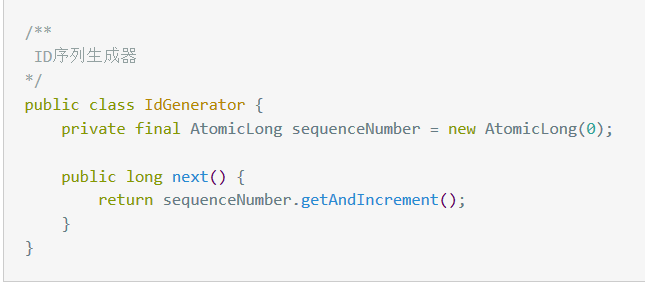
   - **synchronized实现的锁是可重入的锁**。

在JVM内部，为了提高效率，同时运行的每个线程都会有它正在处理的数据的缓存副本，当我们使用Synchronzied进行同步的时候，真正被同步的是在不同线程中表示被锁定对象的内存块（副本数据会保持和主内存的同步，现在知道为什么要用同步这个词汇了吧），简单的说就是在同步块或同步方法执行完后，对被锁定的对象做的任何修改要在释放锁之前写回到主内存中；在进入同步块得到锁之后，被锁定对象的数据是从主内存中读出来的，持有锁的线程的数据副本一定和主内存中的数据视图是同步的 。

#### java.util.concurrent

##### 原子类

Java 5中的java.util.concurrent包下面有一个atomic子包，其中有几个以Atomic打头的类，例如AtomicInteger和AtomicLong。它们利用了现代处理器的特性，可以用非阻塞的方式完成原子操作，代码如下所示：



##### 显示锁

**基于synchronized关键字的锁机制有以下问题**：

   - 锁只有一种类型，而且对所有同步操作都是一样的作用

   - 锁只能在代码块或方法开始的地方获得，在结束的地方释放

   - 线程要么得到锁，要么阻塞，没有其他的可能性

**Java 5对锁机制进行了重构，提供了显示的锁，这样可以在以下几个方面提升锁机制：**

   - 可以添加不同类型的锁，例如读取锁和写入锁。  
   - 可以在一个方法中加锁，在另一个方法中解锁。  
   - 可以使用tryLock方式尝试获得锁，如果得不到锁可以等待、回退或者干点别的事情，当然也可以在超时之后放弃操作。

**显示的锁都实现了java.util.concurrent.Lock接口，主要有两个实现类**：

   - ReentrantLock - 比synchronized稍微灵活一些的重入锁  
   - ReentrantReadWriteLock - 在读操作很多写操作很少时性能更好的一种重入锁

##### CountDownLatch

CountDownLatch是一种简单的同步模式，它让一个线程可以等待一个或多个线程完成它们的工作从而避免对临界资源并发访问所引发的各种问题。下面演示CountDownLatch是如何工作的。

import java.util.concurrent.CountDownLatch;

/\*\*

\* 工人类

\* @author 骆昊

\*/

class Worker {

private String name; // 名字

private long workDuration; // 工作持续时间

/\*\*

\* 构造器

\*/

public Worker(String name, long workDuration) {

this.name = name;

this.workDuration = workDuration;

}

/\*\*

\* 完成工作

\*/

public void doWork() {

System.out.println(name + " begins to work...");

try {

Thread.sleep(workDuration); // 用休眠模拟工作执行的时间

} catch (InterruptedException ex) {

ex.printStackTrace();

}

System.out.println(name + " has finished the job...");

}

}

/\*\*

\* 测试线程

\* @author 骆昊

\*/

class WorkerTestThread implements Runnable {

private Worker worker;

private CountDownLatch cdLatch;

public WorkerTestThread(Worker worker, CountDownLatch cdLatch) {

this.worker = worker;

this.cdLatch = cdLatch;

}

@Override

public void run() {

worker.doWork(); // 让工人开始工作

cdLatch.countDown(); // 工作完成后倒计时次数减1

}

}

class CountDownLatchTest {

private static final int MAX\_WORK\_DURATION = 5000; // 最大工作时间

private static final int MIN\_WORK\_DURATION = 1000; // 最小工作时间

// 产生随机的工作时间

private static long getRandomWorkDuration(long min, long max) {

return (long) (Math.random() \* (max - min) + min);

}

public static void main(String[] args) {

CountDownLatch latch = new CountDownLatch(2); // 创建倒计时闩并指定倒计时次数为2

Worker w1 = new Worker("骆昊", getRandomWorkDuration(MIN\_WORK\_DURATION, MAX\_WORK\_DURATION));

Worker w2 = new Worker("王大锤", getRandomWorkDuration(MIN\_WORK\_DURATION, MAX\_WORK\_DURATION));

new Thread(new WorkerTestThread(w1, latch)).start();

new Thread(new WorkerTestThread(w2, latch)).start();

try {

latch.await(); // 等待倒计时闩减到0

System.out.println("All jobs have been finished!");

} catch (InterruptedException e) {

e.printStackTrace();

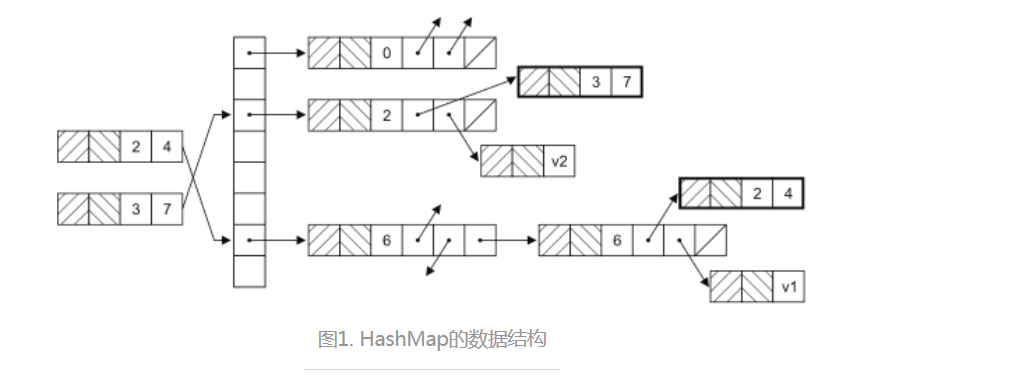
}

}

}

##### ConcurrentHashMap

ConcurrentHashMap是HashMap在并发环境下的版本，大家可能要问，既然已经可以通过Collections.synchronizedMap获得线程安全的映射型容器，为什么还需要ConcurrentHashMap呢？因为通过Collections工具类获得的线程安全的HashMap会在读写数据时对整个容器对象上锁，这样其他使用该容器的线程无论如何也无法再获得该对象的锁，也就意味着要一直等待前一个获得锁的线程离开同步代码块之后才有机会执行。实际上，HashMap是通过哈希函数来确定存放键值对的桶（桶是为了解决哈希冲突而引入的），修改HashMap时并不需要将整个容器锁住，只需要锁住即将修改的“桶”就可以了。HashMap的数据结构如下图所示。



 此外，ConcurrentHashMap还提供了原子操作的方法，如下所示：

   - putIfAbsent：如果还没有对应的键值对映射，就将其添加到HashMap中

   - remove：如果键存在而且值与当前状态相等（equals比较结果为true），则用原子方式移除该键值对映射

   - replace：替换掉映射中元素的原子操作

##### CopyOnWriteArrayList

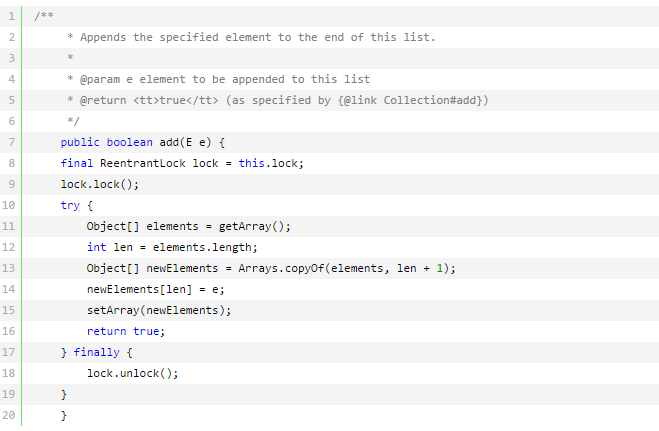
Copy-On-Write简称COW，是一种用于程序设计中的优化策略。其基本思路是，从一开始大家都在共享同一个内容，当某个人想要修改这个内容的时候，才会真正把内容Copy出去形成一个新的内容然后再改，这是一种延时懒惰策略。从JDK1.5开始Java并发包里提供了两个使用CopyOnWrite机制实现的并发容器,它们是CopyOnWriteArrayList和CopyOnWriteArraySet。CopyOnWrite容器非常有用，可以在非常多的并发场景中使用到。

**什么是CopyOnWrite容器**

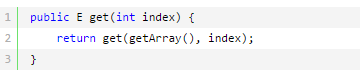
CopyOnWrite容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。

**CopyOnWriteArrayList的实现原理**

在使用CopyOnWriteArrayList之前，我们先阅读其源码了解下它是如何实现的。以下代码是向CopyOnWriteArrayList中add方法的实现（向CopyOnWriteArrayList里添加元素），可以发现在添加的时候是需要加锁的，否则多线程写的时候会Copy出N个副本出来。



读的时候不需要加锁，如果读的时候有多个线程正在向CopyOnWriteArrayList添加数据，读还是会读到旧的数据，因为写的时候不会锁住旧的CopyOnWriteArrayList。



JDK中并没有提供CopyOnWriteMap，我们可以参考CopyOnWriteArrayList来实现一个，基本代码如下：



**CopyOnWrite的应用场景**

CopyOnWrite并发容器用于读多写少的并发场景。比如白名单，黑名单，商品类目的访问和更新场景，假如我们有一个搜索网站，用户在这个网站的搜索框中，输入关键字搜索内容，但是某些关键字不允许被搜索。这些不能被搜索的关键字会被放在一个黑名单当中，黑名单每天晚上更新一次。当用户搜索时，会检查当前关键字在不在黑名单当中，如果在，则提示不能搜索

**CopyOnWrite的缺点:**

CopyOnWrite容器有很多优点，但是同时也存在两个问题，即内存占用问题和数据一致性问题。所以在开发的时候需要注意一下。

**内存占用问题**。因为CopyOnWrite的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，旧的对象和新写入的对象（注意:在复制的时候只是复制容器里的引用，只是在写的时候会创建新对象添加到新容器里，而旧容器的对象还在使用，所以有两份对象内存）。如果这些对象占用的内存比较大，比如说200M左右，那么再写入100M数据进去，内存就会占用300M，那么这个时候很有可能造成频繁的Yong GC和Full GC。之前我们系统中使用了一个服务由于每晚使用CopyOnWrite机制更新大对象，造成了每晚15秒的Full GC，应用响应时间也随之变长。

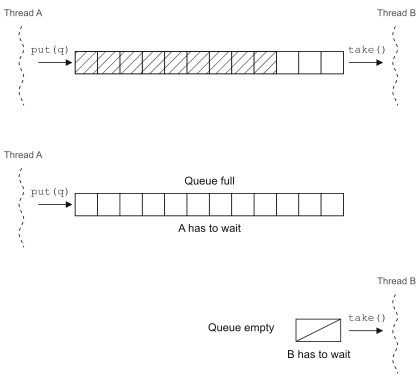
针对内存占用问题，可以通过压缩容器中的元素的方法来减少大对象的内存消耗，比如，如果元素全是10进制的数字，可以考虑把它压缩成36进制或64进制。或者不使用CopyOnWrite容器，而使用其他的并发容器，如ConcurrentHashMap。

**数据一致性问题**。CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的的数据，马上能读到，请不要使用CopyOnWrite容器。

##### Queue

队列是一个无处不在的美妙概念，它提供了一种简单又可靠的方式将资源分发给处理单元（也可以说是将工作单元分配给待处理的资源，这取决于你看待问题的方式）。实现中的并发编程模型很多都依赖队列来实现，因为它可以在线程之间传递工作单元。

   Java 5中的BlockingQueue就是一个在并发环境下非常好用的工具，在调用put方法向队列中插入元素时，如果队列已满，它会让插入元素的线程等待队列腾出空间；在调用take方法从队列中取元素时，如果队列为空，取出元素的线程就会阻塞。



#### Volatile

**锁的两大特性”互斥” 与 “可见性”**。 互斥是说，每次都得等先拿到锁的线程把锁释放了，其他线程才有资格访问共享资源，期间线程对共享资源是独占的。而可见性是指，一个线程修改了某个共享变量的值，其他线程能读取到最新的值，而不至于读取到脏数据。

**CPU有自己的高速缓存**。 CPU执行指令的速度是相当快的，如果每次获取数据时，都需要从操作系统的内存中获取的话，会影响指令的执行速度的。因此CPU会把从内存读取到的数据缓存起来，之后都直接从缓存中获取数据，等数据处理完后，再更新到操作系统的内存中。

**定义**

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

**变量特性**

volatile变量拥有”可见性”的特性。 也即是说，线程可以看到volatile变量的最新值，而不会看到修改前的值或者不正确的值。这个特性在某些场景下，可以让volatile变量是线程安全的。

volatile变量不具备”原子性”。 举个例子，volatile变量i，进行i++操作时，是不能保证i++操作的原子性的，在多线程并发的情况下，仍然可能有线程安全问题。这个也是volatile必须小心谨慎使用的原因。

**volatile性能**

跟synchronized不同，volatile不会引起线程挂起，导致线程上下文切换和调度的开销。

#### Java细粒度锁

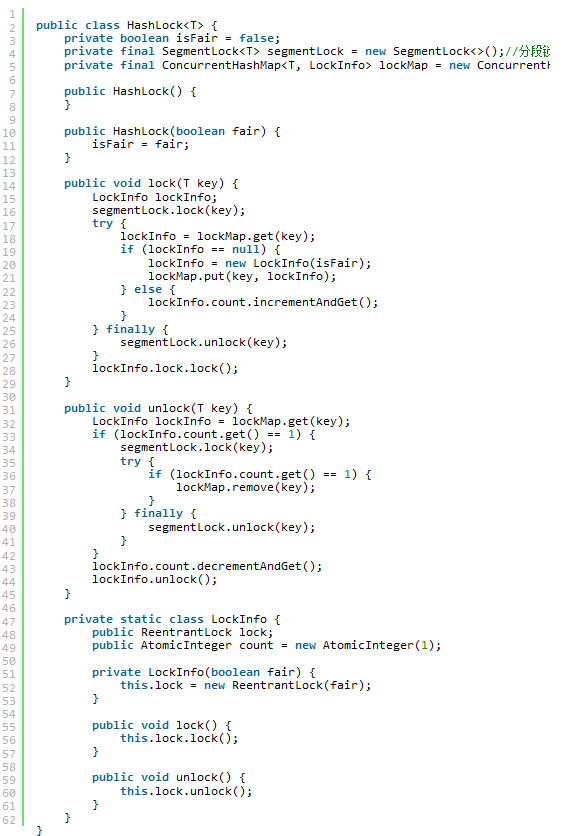
**① 分段锁**

借鉴concurrentHashMap的分段思想，先生成一定数量的锁，具体使用的时候再根据key来返回对应的lock。这是几个实现里最简单，性能最高，也是最终被采用的锁策略，代码如下：

****

**② 哈希锁**

分段锁的基础上发展起来的第二种锁策略，目的是实现真正意义上的细粒度锁。每个哈希值不同的对象都能获得自己独立的锁。在测试中，在被锁住的代码执行速度飞快的情况下，效率比分段锁慢 30% 左右。如果有长耗时操作，感觉表现应该会更好。代码如下：

****

**③ 弱引用锁**

哈希锁因为引入的分段锁来保证锁创建和销毁的同步，总感觉有点瑕疵，所以写了第三个锁来寻求更好的性能和更细粒度的锁。这个锁的思想是借助java的弱引用来创建锁，把锁的销毁交给jvm的垃圾回收，来避免额外的消耗。

## JAVA框架

### Spring

#### 控制反转（IOC）

#### 面向切面（AOP）

### Spring MVC

### Struts2

### MyBatis

### Hibernate

## Web前端

## 数据库（MySQL、Oracle）

### SQL基础

① **SQL语句执行顺序**

1、查询中用到的关键词主要包含六个，并且他们的**声明顺序**：

select--from--where--group by--having--order by。

**执行顺序**：from--where--group by--having--select--order by,

from:需要从哪个数据表检索数据

where:过滤表中数据的条件

group by:如何将上面过滤出的数据分组

having:对上面已经分组的数据进行过滤的条件

select:查看结果集中的哪个列，或列的计算结果

order by :按照什么样的顺序来查看返回的数据

2、from后面的表关联，是自右向左解析的

而where条件的解析顺序是自下而上的。

也就是说，在写SQL文的时候，尽量把数据量大的表放在最右边来进行关联，

而把能筛选出大量数据的条件放在where语句的最下面。

3、具体执行顺序：

**from 子句--执行顺序为从后往前、从右到左**

oracle 的解析器按照从右到左的顺序处理，FROM 子句中的表名，FROM 子句中写在最后的表(基础表 driving table)将被最先处理，即最后的表为驱动表，在FROM 子句中包含多个表的情况下,你必须选择记录条数最少的表作为基础表。如果有3 个以上的表连接查询, 那就需要选择交叉表(intersection table)作为基础表, 交叉表是指被其他表所引用的表

多表连接时，使用表的别名并把别名前缀于每个Column上。可以减少解析的时间并减少那些由Column 歧义引起的语法错误.

**where子句--执行顺序为自下而上、从右到左**

ORACLE 采用自下而上从右到左的顺序解析Where 子句,根据这个原理,表之间的连接必须写在其他Where 条件之前, 可以过滤掉最大数量记录的条件必须写在Where 子句的末尾。

**group by--执行顺序从左往右分组**

提高GROUP BY 语句的效率, 可以通过将不需要的记录在GROUP BY 之前过滤掉。即在GROUP BY前使用WHERE来过虑，而尽量避免GROUP BY后再HAVING过滤。

**having 子句----很耗资源，尽量少用**

避免使用HAVING 子句, HAVING 只会在检索出所有记录之后才对结果集进行过滤. 这个处理需要排序,总计等操作.

如果能通过Where 子句在GROUP BY前限制记录的数目,那就能减少这方面的开销.

(非oracle 中)on、where、having 这三个都可以加条件的子句中，on 是最先执行，where 次之，having 最后，因为on 是先把不符合条件的记录过滤后才进行统计，它就可以减少中间运算要处理的数据，按理说应该速度是最快的，

where 也应该比having 快点的，因为它过滤数据后才进行sum，在两个表联接时才用on 的，所以在一个表的时候，就剩下where 跟having比较了。

在这单表查询统计的情况下，如果要过滤的条件没有涉及到要计算字段，那它们的结果是一样的，只是where 可以使用rushmore 技术，而having 就不能，在速度上后者要慢。

如果要涉及到计算的字段，就表示在没计算之前，这个字段的值是不确定的，where 的作用时间是在计算之前就完成的，而having 就是在计算后才起作用的，所以在这种情况下，两者的结果会不同。

在多表联接查询时，on 比where 更早起作用。系统首先根据各个表之间的联接条件，把多个表合成一个临时表后，再由where 进行过滤，然后再计算，计算完后再由having 进行过滤。

由此可见，要想过滤条件起到正确的作用，首先要明白这个条件应该在什么时候起作用，然后再决定放在那里。

**select子句--少用\*号，尽量取字段名称**。

ORACLE 在解析的过程中, 会将依次转换成所有的列名, 这个工作是通过查询数据字典完成的, 使用列名意味着将减少消耗时间。

**sql 语句用大写的**；因为 oracle 总是先解析 sql 语句，把小写的字母转换成大写的再执行

**order by子句--执行顺序为从左到右排序,很耗资源**

#### 索引

#### 视图

#### 存储过程

#### 触发器

### SQL 优化

### 乐观锁和悲观锁

### 分区和分表

### 事务

 ① 事务的ACID属性：

    1. 原子性（Atomicity）

原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。  
    2. 一致性（Consistency）

 事务必须使数据库从一个一致性状态变换到另外一个一致性状态。

3. 隔离性（Isolation）

 事务的隔离性是指一个事务的执行不能被其他事务干扰，即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

4. 持久性（Durability）  
    持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来的其他操作和数据库故障不应该对其有任何影响。

② 事务隔离级别：

**读操作未提交（Read Uncommitted）**：读取未提交的数据是允许的。说明一个事务在提交前，其变化对于其他事务来说是可见的。这样脏读、不可重读和幻读都是允许的。当一个事务已经写入一行数据但未提交，其他事务都不能再写入此行数据；但是，任何事务都可以读任何数据。这个隔离级别使用排写锁实现。

**读操作已提交（Read Committed）**：读取未提交的数据是不允许的，它使用临时的共读锁和排写锁实现。这种隔离级别不允许脏读，但不可重读和幻读是允许的。

**可重读（Repeatable Read）**：说明事务保证能够再次读取相同的数据而不会失败。此隔离级别不允许脏读和不可重读，但幻读会出现。

**可串行化（Serializable）**：提供最严格的事务隔离。这个隔离级别不允许事务并行执行，只允许串行执行。这样，脏读、不可重读或幻读都可发生。

**1. 1事务隔离与隔离级别的关系**

|  |  |  |  |
| --- | --- | --- | --- |
| **隔离级别** | **脏读（Dirty Read）** | **不可重读（Unrepeatable read）** | **幻读（Phantom Read）** |
| 读操作未提交（Read Uncommitted） | 可能 | 可能 | 可能 |
| 读操作已提交（Read Committed） | 不可能 | 可能 | 可能 |
| 可重读（Repeatable Read） | 不可能 | 不可能 | 可能 |
| 可串行化（Serializable） | 不可能 | 不可能 | 不可能 |

③ 事务的传播机制：

|  |  |
| --- | --- |
| **事务传播行为类型** | **说明** |
| **PROPAGATION\_REQUIRED** | **如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。这是 最常见的选择。** |
| **PROPAGATION\_SUPPORTS** | **支持当前事务，如果当前没有事务，就以非事务方式执行。** |
| **PROPAGATION\_MANDATORY** | **使用当前的事务，如果当前没有事务，就抛出异常。** |
| **PROPAGATION\_REQUIRES\_NEW** | **新建事务，如果当前存在事务，把当前事务挂起。** |
| **PROPAGATION\_NOT\_SUPPORTED** | **以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。** |
| **PROPAGATION\_NEVER** | **以非事务方式执行，如果当前存在事务，则抛出异常。** |
| **PROPAGATION\_NESTED** | **如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与 PROPAGATION\_REQUIRED 类似的操作。** |

当使用 PROPAGATION\_NESTED 时， 底层的数据源必须基于 JDBC 3.0 ，并且实现者需要支持保存点事务机制。

**readOnly**事务属性中的readOnly标志表示对应的事务应该被最优化为只读事务。这是一个最优化提示 。在一些情况下，一些事务策略能够起到显著的最优化效果，例如在使用Object/Relational映射工具 （如：Hibernate或TopLink）时避免dirty checking（试图“刷新”）。

**Timeout** 在事务属性中还有定义“timeout”值的选项，指定事务超时为几秒。在JTA中，这将被简单地传递到J2EE服务器的事务协调程序，并据此得到相应的解释。

## 缓存（Redis）

## 消息中间件（MQ）

### RabbitMQ

**1.什么是AMPQ？**

AMQP，即Advanced Message Queuing Protocol，高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。基于此协议的客户端与消息中间件可传递消息，并不受客户端/中间件不同产品，不同的开发语言等条件的限制。Erlang中的实现有 RabbitMQ等。

**2.AMPQ的特征**

AMQP的主要特征是面向消息、队列、路由（包括点对点和发布/订阅）、可靠性、安全。

**3.AMPQ的实现**

**3.1 OpenAMQ**

AMQP的开源实现，用C语言编写，运行于Linux、AIX、Solaris、Windows、OpenVMS

**3.2 Apache Qpid**

Apache的开源项目，支持C++、Ruby、Java、JMS、[**Python**](http://lib.csdn.net/base/11)和.NET

**3.3 Redhat Enterprise MRG**

实现了AMQP的最新版本0-10，提供了丰富的特征集，比如完全管理、联合、Active-Active集群，有Web控制台，还有许多企业级特征，客户端支持C++、Ruby、Java、JMS、Python和.NET

**3.4 RabbitMQ**

一个独立的开源实现，服务器端用Erlang语言编写，支持多种客户端，如：Python、Ruby、.NET、Java、JMS、C、PHP、ActionScript、XMPP、STOMP等，支持AJAX。RabbitMQ发布在Ubuntu、FreeBSD、Centos中的实现

**4.RabbitMQ是什么**

RabbitMQ是一个遵循AMQP协议的消息中间件，它从生产者接收消息并递送给消费者，在这个过程中，根据规则进行路由，缓存与持久化。

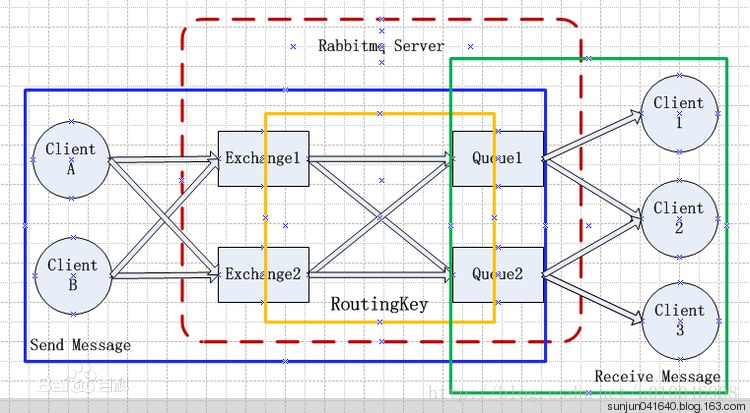
**5.RabbitMQ能为你做些什么？**

消息系统允许软件、应用相互连接和扩展．这些应用可以相互链接起来组成一个更大的应用，或者将用户设备和数据进行连接。消息系统通过将消息的发送和接收分离来实现应用程序的异步和解偶。

或许你正在考虑进行数据投递，非阻塞操作或推送通知。或许你想要实现发布／订阅，异步处理，或者工作队列。所有这些都可以通过消息系统实现。

在项目中，将一些无需即时返回且耗时的操作提取出来，进行了异步处理，而这种异步处理的方式大大的节省了服务器的请求响应时间，从而提高了系统的吞吐量。

**6.RabbitMQ的结构图**



几个概念说明：

* Broker：简单来说就是消息队列服务器实体。
* Exchange：消息交换机，它指定消息按什么规则，路由到哪个队列。
* Queue：消息队列载体，每个消息都会被投入到一个或多个队列。
* Binding：绑定，它的作用就是把exchange和queue按照路由规则绑定起来。
* Routing Key：路由关键字，exchange根据这个关键字进行消息投递。
* vhost：虚拟主机，一个broker里可以开设多个vhost，用作不同用户的权限分离。
* Producer：消息生产者，就是投递消息的程序。
* Consumer：消息消费者，就是接受消息的程序。
* Channel：消息通道，在客户端的每个连接里，可建立多个channel，每个channel代表一个会话任务。

消息队列的使用过程大概如下：

1. 客户端连接到消息队列服务器，打开一个channel。
2. 客户端声明一个exchange，并设置相关属性。
3. 客户端声明一个queue，并设置相关属性。
4. 客户端使用routing key，在exchange和queue之间建立好绑定关系。
5. 客户端投递消息到exchange。
6. exchange接收到消息后，就根据消息的key和已经设置的binding，进行消息路由，将消息投递到一个或多个队列里

exchange也有几个类型，完全根据key进行投递的叫做Direct交换机，例如，绑定时设置了routing key为”abc”，那么客户端提交的消息，只有设置了key为”abc”的才会投递到队列。对key进行模式匹配后进行投递的叫做Topic交换机，符号”#”匹配一个或多个词，符号”*”匹配正好一个词。例如”abc.#”匹配”abc.def.ghi”，”abc.*”只匹配”abc.def”。还有一种不需要key的，叫做Fanout交换机，它采取广播模式，一个消息进来时，投递到与该交换机绑定的所有队列。

RabbitMQ支持消息的持久化，也就是数据写在磁盘上，为了数据安全考虑，我想大多数用户都会选择持久化。消息队列持久化包括3个部分：

1. exchange持久化，在声明时指定durable => 1
2. queue持久化，在声明时指定durable => 1
3. 消息持久化，在投递时指定delivery\_mode => 2（1是非持久化）

如果exchange和queue都是持久化的，那么它们之间的binding也是持久化的。如果exchange和queue两者之间有一个持久化，一个非持久化，就不允许建立绑定。

**7、RabbitMQ之交换机**

消息(Message)由Client发送，RabbitMQ接收到消息之后通过交换机转发到对应的队列上面。Worker会从队列中获取未被读取的数据处理。

RabbitMQ包含四种不同的交换机类型：

1. Direct exchange：直连交换机，转发消息到routigKey指定的队列，如果消息的routigKey和binding的routigKey直接匹配的话，消息将会路由到该队列
2. Fanout exchange：扇形交换机，转发消息到所有绑定队列（速度最快），不管消息的routigKey息和binding的参数表头部信息和值是什么，消息将会路由到所有的队列
3. Topic exchange：主题交换机，按规则转发消息（最灵活），如果消息的routigKey和binding的routigKey符合通配符匹配的话，消息将会路由到该队列
4. Headers exchange：首部交换机 ，如果消息的头部信息和binding的参数表中匹配的话，消息将会路由到该队列。

## 网络编程（WebService）

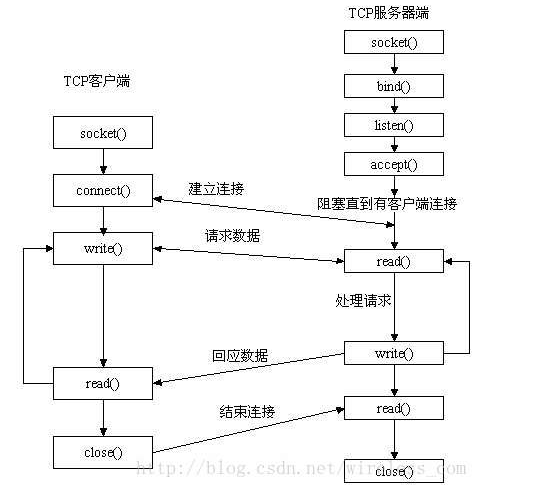
### TCP

### UDP

### Socket

**Socket（套接字）是通信的基石**，是支持TCP/IP协议网络通信的基本操作单元。Socket实质上提供了进程通信的端点。进程通信之前，双方首先必须各自创建一个端点，否则是没有办法建立联系并相互通信的。一个完整的socket有一个本地唯一的socket号，这是由操作系统分配的。

从设计模式的角度看， Socket其实是一个**外观模式**，它把复杂的TCP/IP协议栈隐藏在Socket接口后面，对用户来说，一组简单的Socket接口就是全部



### RPC（Remote Procedure Call Protocol）

RPC（Remote Procedure Call Protocol）——远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。RPC协议假定某些传输协议的存在，如TCP或UDP，为通信程序之间携带信息数据。在OSI网络通信模型中，RPC跨越了传输层和应用层。RPC使得开发包括网络分布式多程序在内的应用程序更加容易。

RPC采用客户机/服务器模式。请求程序就是一个客户机，而服务提供程序就是一个服务器。首先，客户机调用进程发送一个有进程参数的调用信息到服务进程，然后等待应答信息。在服务器端，进程保持睡眠状态直到调用信息到达为止。当一个调用信息到达，服务器获得进程参数，计算结果，发送答复信息，然后等待下一个调用信息，最后，客户端调用进程接收答复信息，获得进程结果，然后调用执行继续进行。

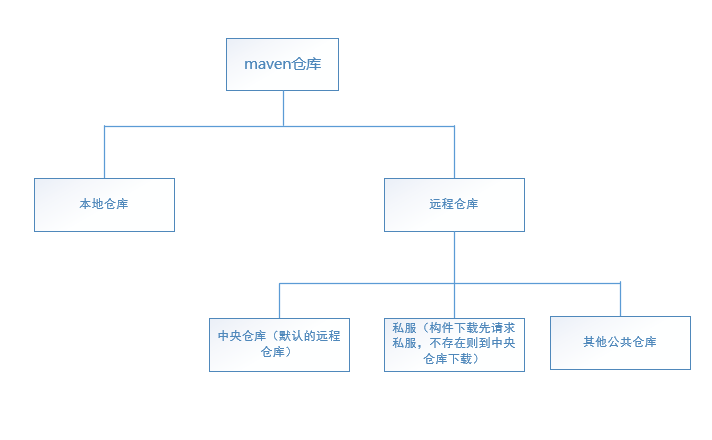
## 工具（Maven,Git）

① maven依赖范围Scope：compile、test、provided、runtime、system。

依赖原则：

1. 路径优先原则
2. 第一声明优先

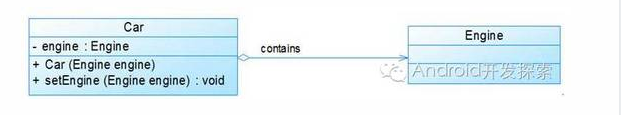
② maven仓库的结构



## UML基础

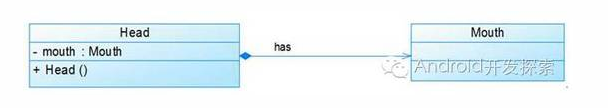
① 聚合关系和组合关系的区别：

聚合关系



上 图中的Car类与Engine类就是聚合关系（Car类中包含一个Engine类型的成员变量）。由上图我们可以看到，UML中 聚合关系用带空心菱形和 箭头的直线表示。聚合关系强调是“整体”包含“部分”，但是“部分”可以脱离“整体”而单独存在。比如上图中汽车包含了发动机，而发动机脱离了汽车也能够 单独存在。

组合关系



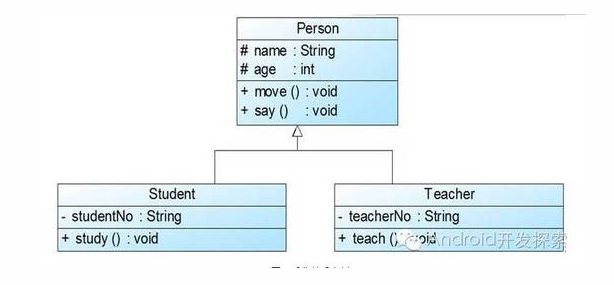
组合关系与聚合关系间的最大不同在于：这里的“部分”脱离了“整体”便不复存在。比如下图：

显然，嘴是头的一部分且不能脱离了头而单独存在。在UML类图中，组合关系用一个带实心菱形和箭头的直线表示。

② 继承关系和接口实现关系

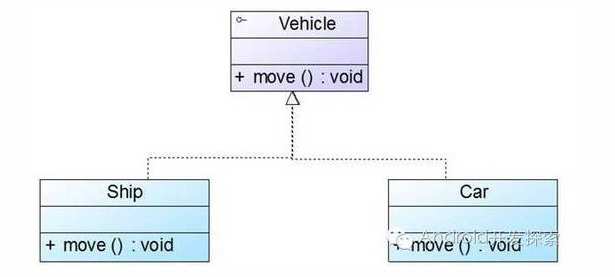
继承关系：

继承关系对应的是extend关键字，在UML类图中用带空心三角形的直线表示，如下图所示中，Student类与Teacher类继承了Person类。

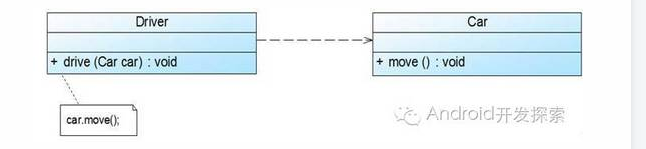


接口实现关系：

这种关系对应implements关键字，比如下图中，Car类与Ship类都实现了Vehicle接口。



③ 依赖关系



从上图中我们可以看到，Driver的drive方法只有传入了一个Car对象才能发挥作用，因此我们说Driver类依赖于Car类。在UML类图中，依赖关系用一条带有箭头的虚线表示。