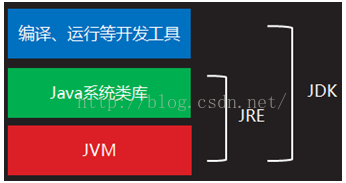
# JDK、JRE、JVM



**JDK**(Java Development Kit) 是 Java 语言的软件开发工具包(SDK)。在JDK的安装目录下有一个jre目录，同时还包含里编译java源码的编译器javac，还有很多java程序调试和分析的工具等

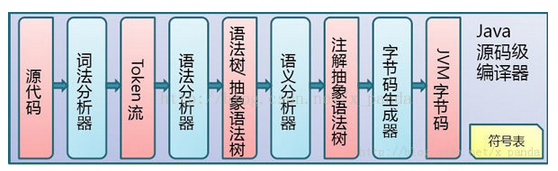
**JRE**（Java Runtime Environment，Java运行环境），包含JVM及Java基础类库。JRE是Java运行环境，里面有两个文件夹bin和lib，在这里可以认为bin里的就是jvm，lib中则是jvm工作所需要的类库，而jvm和 lib合起来就称为jre。



**JVM**是Java Virtual Machine（Java虚拟机）的缩写，JVM是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

# Java 源码编译机制

Java代码编译是由Java源码编译器来完成，也就是Java代码到JVM字节码（.class文件）的过程。比如JDK中的编译器javac

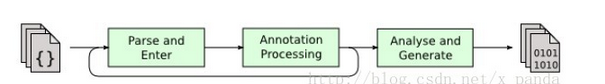


Java 源码编译由以下三个过程组成：

①分析和输入到符号表

②注解处理

③语义分析和生成class文件



最后生成的class文件由以下部分组成：

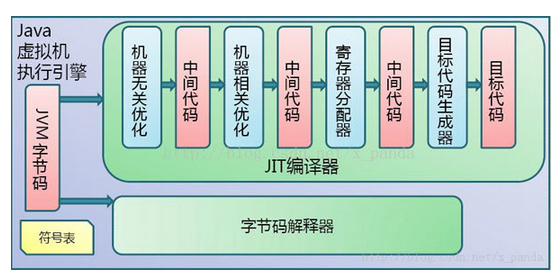
①结构信息：包括class文件格式版本号及各部分的数量与大小的信息

②元数据：对应于Java源码中声明与常量的信息。包含类/继承的超类/实现的接口的声明信息、域与方法声明信息和常量池

③方法信息：对应Java源码中语句和表达式对应的信息。包含字节码、异常处理器表、求值栈与局部变量区大小、求值栈的类型记录、调试符号信息

# Java 类执行机制

Java字节码的执行是由JVM执行引擎来完成。对于一个Java程序来说，它的运行就是通过对堆栈的操作来完成的。堆栈以帧为单位保存线程的状态。JVM对堆栈只进行两种操作:以帧为单位的压栈和出栈操作。



**解释器**：一条一条地读取，解释并且执行字节码指令。因为它一条一条地解释和执行指令，所以它可以很快地解释字节码，但是执行起来会比较慢。这是解释执行的语言的一个缺点。字节码这种“语言”基本来说是解释执行的。

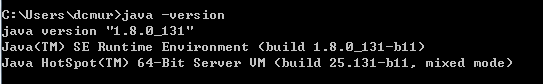
**即时（Just-In-Time)编译器**：即时编译器被引入用来弥补解释器的缺点。执行引擎首先按照解释执行的方式来执行，然后在合适的时候，即时编译器把整段字节码编译成本地代码。然后，执行引擎就没有必要再去解释执行方法了，它可以直接通过本地代码去执行它。执行本地代码比一条一条进行解释执行的速度快很多。编译后的代码可以执行的很快，因为本地代码是保存在缓存里的。

不过，用JIT编译器来编译代码所花的时间要比用解释器去一条条解释执行花的时间要多。因此，如果代码只被执行一次的话，那么最好还是解释执行而不是编译后再执行。因此，内置了JIT编译器的JVM都会检查方法的执行频率，如果一个方法的执行频率超过一个特定的值的话，那么这个方法就会被编译成本地代码。

Oracle Hotspot VM使用一种叫做热点编译器的JIT编译器。它之所以被称作”热点“是因为热点编译器通过分析找到最需要编译的“热点”代码，然后把热点代码编译成本地代码。如果已经被编译成本地代码的字节码不再被频繁调用了，换句话说，这个方法不再是热点了，那么Hotspot VM会把编译过的本地代码从cache里移除，并且重新按照解释的方式来执行它。大部分Java程序的性能都是通过提升执行引擎的性能来达到的。正如JIT编译器一样，很多优化的技术都被引入进来使得JVM的性能一直能够得到提升。最原始的JVM和最新的JVM最大的差别之处就是在于执行引擎。

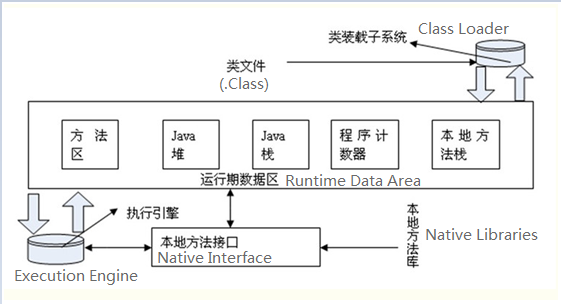
# JVM

Oracle公司分别收购了BEA公司（2008年）和Sun公司（2009年），这样Oracle就同时拥有了两款优秀的Java虚拟机：JRockit VM和HotSpot VM。JDK1.7及以前版本我们使用的都是Sun公司的HotSpot，JDK1.8的jvm是两者的融合。



## 组成

Java虚拟机是内存中的虚拟机，意味是jvm的存储就是内存，我们所写的类，常量，变量、方法就是在内存中。由下图可以看出，jvm主要由class loader、runtime data area、execution engine、native interface 4个部分组成



Class Loader: 依据特定格式，加载class文件到内存

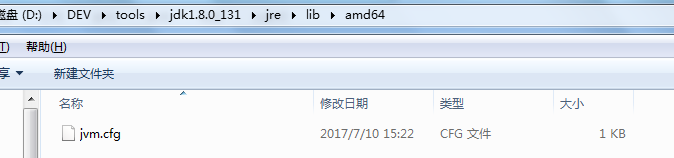
Execution Engine: 对命令进行解析

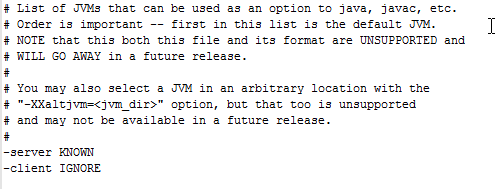
Native interface:本地接口，调用其他语言外部类库。融合不同的开发语言的原生库为java所用

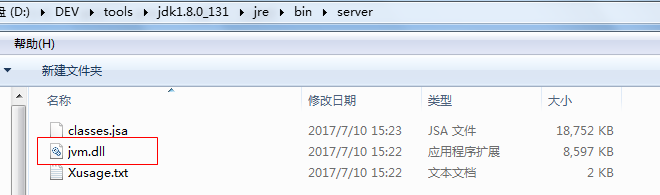
untime data area：jvm内存空间结构模型

jvm.cfg ——> jvm.dll

执行java –jar xxx，此时java根据系统版本找到jvm.cfg, 其中-server KNOWN就表示名称为server的jvm可用。如果这时你搜索一下你电脑上jvm.dll，你就会发现它一定在你的某个server目录下，jvm.dll则是java虚拟机的主要实现。







class文件(字节码)被jvm装载以后，经过jvm的内存空间调配，最终是由执行引擎完成class文件的执行。

jvm把.class文件加载到内存，

## Java 类的装载过程

1. 类加载
2. 链接（验证-准备-解析）
3. 初始化

当我们使用命令来执行某一个Java程序的时候：java -jar gateway-service.jar

(1) java.exe 会帮助我们找到JRE ，接着找到位于JRE内部的 jvm.dll ，这才是真正的Java虚拟机器 , 最后加载动态库，激活Java虚拟机器。

(2) 虚拟机器激活以后，会先做一些初始化的动作，比如说读取系统参数等。一旦初始化动作完成之后，就会产生第一个类装载器 —— Bootstrap Loader(启动类装载器 ) 。

(3) Bootstrap Loader所做的初始工作中，除了一些基本的初始化动作之外，最重要的就是加载 Launcher.java 之中的 ExtClassLoader(扩展类装载器) ,并设定其 Parent为null,代表其父加载器为 BootstrapLoader 。

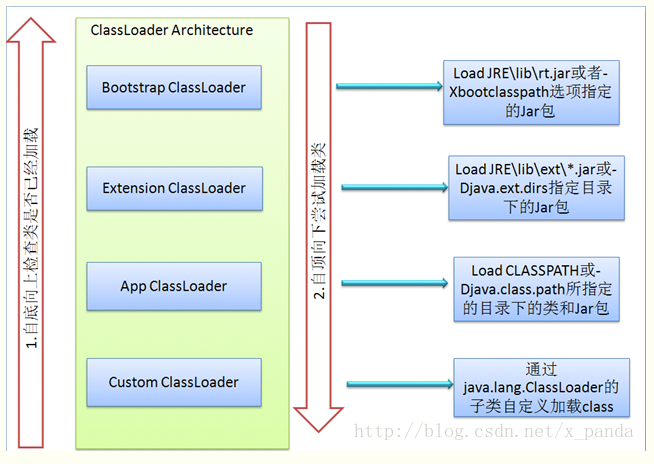
(4) 然后Bootstrap Loader再要求加载 Launcher.java之中的 AppClassLoader(用户自定义类装载器 ) ，并设定其 Parent为之前产生的 ExtClassLoader 实体。这两个加载器都是以静态类的形式存在的。

要注意的是，Launcher$ExtClassLoader.class和Launcher$AppClassLoader.class

都是由Bootstrap Loader所加载，所以Parent和由哪个类加载器加载没有关系。

### ClassLoader

将字节码输入到内存中，这个过程叫加载，JVM的类加载是通过ClassLoader及其子类来完成的，类的层次关系和加载顺序可以由下图来描述：



①Bootstrap ClassLoader

负责加载$JAVA\_HOME中jre/lib/rt.jar里所有的class，由C++实现，不是ClassLoader子类

②Extension ClassLoader

负责加载java平台中扩展功能的一些jar包，包括$JAVA\_HOME中jre/lib/\*.jar或-Djava.ext.dirs指定目录下的jar包

③Application ClassLoader

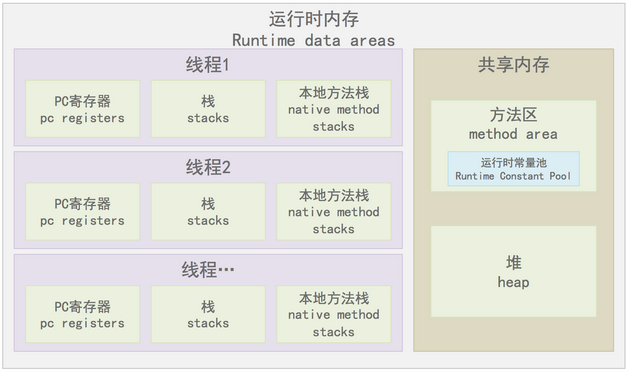
负责记载classpath中指定的jar包及目录中class

④Custom ClassLoader 自定义类

属于应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader

加载过程中会先检查类是否被已加载，检查顺序是自底向上，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个classloader已加载就视为已加载此类，保证此类只所有ClassLoader加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

## Runtime data areas



线程私有

其中，方法区只是JVM规范定义，而永久代（PermGen:Permanent Generation space）为具体的实现，元空间（MetaSpace）也是方法区在jdk1.8中的一种实现。

1.8同1.7比，最大的差别就是：元数据区取代了永久代，元空间的本质和永久代类似，都是对JVM规范中方法区的实现。最大的区别在于：元数据空间并不在虚拟机中，而是使

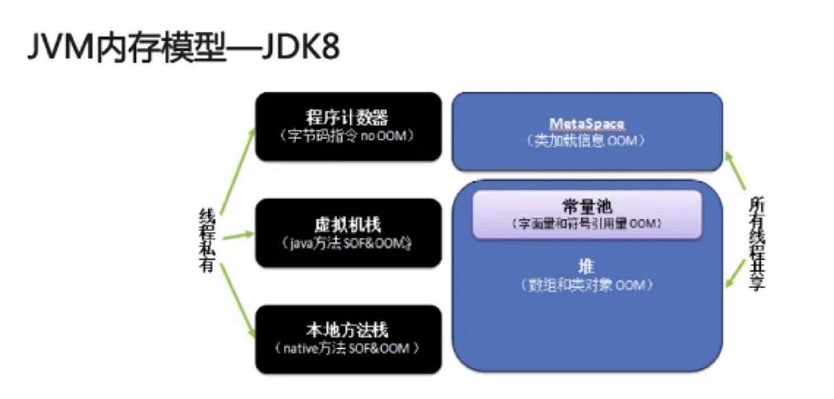
用本地内存。

**常量池：**

ava6和6之前，常量池是存放在方法区中的。

Java7，将常量池是存放到了堆中，常量池就相当于是在永久代中，所以永久代存放在堆中。

Java8之后，取消了整个永久代区域，取而代之的是元空间。没有再对常量池进行调整。



### 程序计数器（Program Counter Register，PC寄存器）：

和线程是一对一关系线程私有，

是逻辑计数器，

不会发生内存泄露

对java方法计数，如果是Native方法，则计数器值为Undefined

类已经加载，实例对象、方法、静态变量都去了自己改去的地方，那么问题来了，程序该怎么执行，哪个方法先执行，哪个方法后执行，这些指令执行的顺序就是PC寄存器在管，它的作用就是控制程序指令的执行顺序。

### 本地方法栈：Native method stacks

中登记native方法，在Excution Engine执行的时候加载Native Libraries。角色和java栈类似，只不过它是用来表示执行本地方法的，本地方法栈存放的方法调用本地方法接口，最终调用本地方法库，实现与操作系统、硬件交互的目的。

### 方法区

各线程共享的区域，存放类信息、常量、静态变量。

#### MetaSpace相比PermGen的优势

字符串常量池存在永久代中，容易出现性能问题和内存溢出

类和方法的大小难确定，给永久代的大小指定带来困难

永久代为GC带来不必要的复杂度

方便HotSport和其它JVM的如Jrockit的集成

### Java虚拟机栈（Stack）：

是每个线程私有的区域，是一块连续的内存区域，能从栈获得的空间较小，

它的生命周期与线程相同，一个线程对应一个java栈，方法执行的内存模型

用来存储栈帧

每执行一个方法，会创建一个栈帧（即方法运行期间的基础数据结构），对应从入栈到出栈的过程

它类似一个集合，是有固定容量的，是由多个栈帧合起来的。

“栈帧”，程序运行时，每调用一个方法，java虚拟机就会分配一块空间，那就是一个栈帧了，用于存储局部变量表、操作数栈、动态链接、方法出口等。方法调用结束后，栈帧就会被自动释放掉。即栈的内存不需要GC回收

#### 特点

Java虚拟机栈会出现两种异常

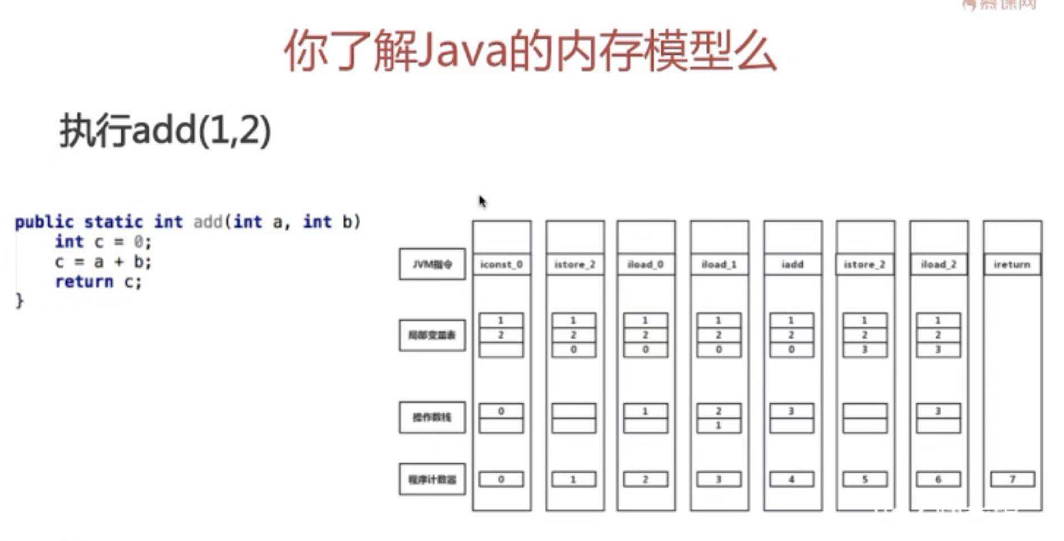
StackOverFlowError 若Java虚拟机栈的内存大小不允许动态扩展,那么当线程请求的栈深度大于虚拟机允许的最大深度时(但内存空间可能还有很多),就抛出此异常。每个线程的虚拟机栈深度是固定的，递归如果深度很深，栈帧数超出虚拟机栈深度，斐波那契（fibonacci）

OutOfMemoryError 若Java虚拟机栈的内存大小允许动态扩展,且当线程请求栈时内存用完了,无法再动态扩展了,此时抛出OutOfMemoryError异常

#### 局部变量表和操作数栈

局部变量表：方法执行过程中的所有变量

操作数栈：入栈、出栈、复制、交换、产生消费变量



总结：局部变量表是为操作数栈提供数据支撑的

### Java堆（Heap）

线程共享，物理上不连续

堆是jvm内存管理的最大的一块区域，

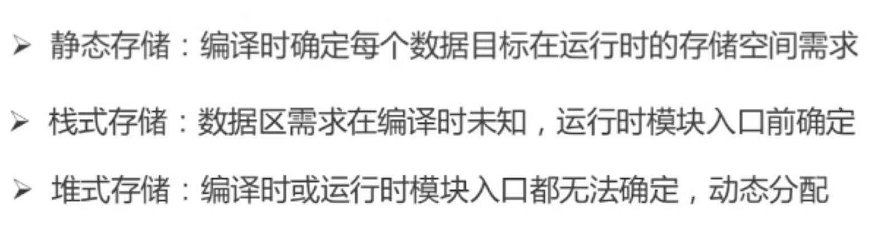
此内存区域的唯一目的就是存放对象的实例，所有对象实例与数组都要在堆上分配内存。它也是垃圾收集器的主要管理区域

如果java堆空间不足了，程序会抛出OutOfMemoryError异常。

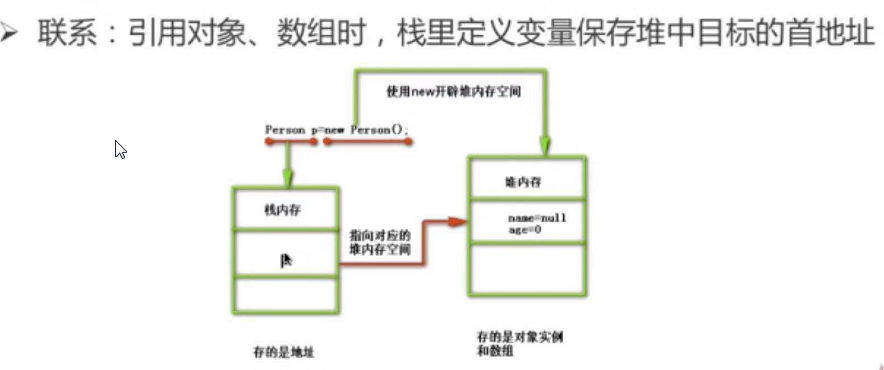
# 堆内存与栈内存的区别

## 内存分配策略

静态式、堆式、栈式



## 联系



## 区别

管理方式：栈自动释放，堆需要GC

空间大小：栈比堆小

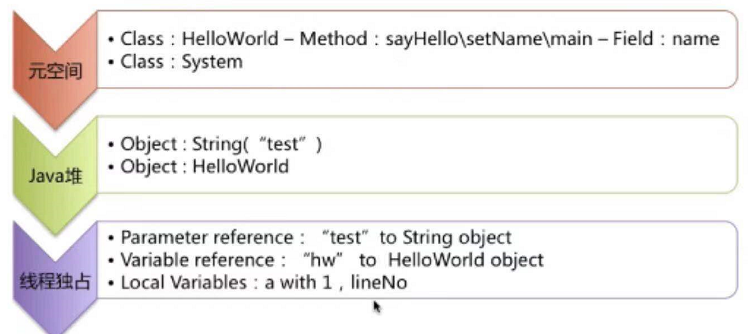
碎片相关：栈产生的碎片远小于堆

分配方式：栈支持静态分配和动态分配，堆仅支持动态分配(是一个运行时级别的内存)

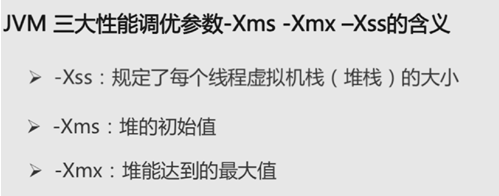
效率：栈比堆高。堆空间最大的优点就是动态分配，比较灵活，但是效率会下降

# MetaSpace、heap、线程独占的联系





# JVM三大性能调优参数



Java –Xms128m –Xmx128m –Xss256k –jar xxx.jar

通常–Xms和–Xmx设置的一样大，因为heap不够用要扩容时，会发生内存抖动，影响程序运行的稳定性

# Java GC

无能为力：JVM的垃圾回收机制属于Java内存管理的一部分，它只是负责回收堆内存中分配出来的内存，至于程序中打开的物理资源，垃圾回收机制管不了，如数据库连接、网络连接、磁盘文件。

对象可被回收：

对象没有被引用

作用域发生未捕获异常

程序在作用域正常执行完毕程序执行完毕

堆

Heap

新生代

Young

老年代

Old

Eden

Survivor

Survivor1(FromSpace)

Survivor2(ToSpace)

默认分配是8:1:1

程序执行了System.exit()

程序发生意外终止（被杀线程）

每次调用Eden和Survivor1，当发生回收，将Eden和Survivor1存活的对象复制到Survivor2中，然后直接清理掉Eden和Survivor1的空间。

Java对象是否存活的判断算法——根搜索算法：把内存中的每一个对象都看做一个节点，并且定义了一些对象作为根节点“GC Roots”。JVM会启一个线程从所有的GC Roots开始往下遍历，遍历完之后一些对象不可到达，则认为对象已经没用，可被回收

# 内存泄露

## 各种连接

比如数据库连接（dataSourse.getConnection()），网络连接(socket)和io连接，除非其显式的调用了其close（）方法将其连接关闭，否则是不会自动被GC 回收的。对于Resultset 和Statement 对象可以不进行显式回收，但Connection 一定要显式回收，因为Connection 在任何时候都无法自动回收，而Connection一旦回收，Resultset 和Statement 对象就会立即为NULL。但是如果使用连接池，情况就不一样了，除了要显式地关闭连接，还必须显式地关闭Resultset Statement 对象（关闭其中一个，另外一个也会关闭），否则就会造成大量的Statement 对象无法释放，从而引起内存泄漏。这种情况下一般都会在try里面去的连接，在finally里面释放连接。

## 单例模式

# Java 引用

Java 引用分为4种：强引用、软引用、弱引用、虚引用

强引用

1. 自我介绍，项目介绍
2. java常用的数据结构有哪些?哪些是线程安全的?是怎么保证线程安全的？

安全：ConcurrentHashMap、HashTable

不安全：HashMap、ArrayList、LinkedList

1. 常见的数据结构有哪些。
2. HashMap在jdk1.7和1.8的区别，为什么引入这个概念？hash碰撞怎么解决，为什么1.8要比1.7更好，好在哪？

1.7用的头插法，1.8用的尾插法。

因为JDK1.7是用单链表进行的纵向延伸，当采用头插法时会容易出现逆序且环形链表死循环问题。但是在JDK1.8之后是因为加入了红黑树使用尾插法，能够避免出现逆序且链表死循环的问题。

1. 说说HashMap的原理, 以及HashMap如何扩充bucket的大小

它内部是基于哈希表实现的键值对存储，继承 AbstractMap 并且实现了 Map 接口。HashMap默认容量是16，在不指定大小的情况下，是默认容量\*加载因子（默认0.75），即在元素数量大于容量\*加载因子（12）的时候，就开始自动扩容一倍，变成32。然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作，所以如果我们已经预知hashmap中元素的个数，那么预设元素的个数能够有效的提高hashmap的性能

1. Mysql索引的分类(Btree, hash)，各自使用什么情况 。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*索尼\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. 你有没有用过Spring的AOP? 是用来干嘛的? 大概会怎么使用？

面向切面编程，

1. mysql数据库调优。
2. java对象四种引用。
3. Java的垃圾回收机制，Java文件加载机制，tomcat类加载机制，锁机制，jvm原理及线上调优，jvm内存模型。
4. SpringMVC如果希望把输出的Object(例如XXResult或者XXResponse)这种包装为JSON输出, 应该怎么处理?
5. 线程池内部工作原理可以说一下么？
6. 死锁是什么意思，形成条件是什么？出现死锁是可以通过什么方式去排查。

获取锁操作之后，又不释放锁，导致后面的线程无法操作。Redis远程指令拍它时一直存在，导致后续远程操作无法执行。

1. 讲一下怎么使用分布式锁。

可以通过Redis来实现分布式锁

1. 在交易过程中如何放在用户在支付时的重复支付（交叉支付），请写出你了解的方案或使用的过的方案。

用户带上标记支付记录的id,每次进行支付操作的时候，条件带上标记id,有这条用户记录再去做支付操作，更新数据库

1. 数据库索引有哪几种，他们之间的区别。
2. 了解哪些设计模式，用伪代码实现一个你熟悉的设计模式。。
3. 说说Myisam, Innodb区别。
4. 知道哪些负载均衡算法。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*普华永道\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

建组合索引

user表 name重复，取工资最大值

反射

服务发现要注意什么

哪些造成内存泄露

读取加载本地一个文件

sql注入

group by join要注意什么

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* 自我介绍，项目介绍
* java常用的数据结构有哪些?哪些是线程安全的?是怎么保证线程安全的？

安全：ConcurrentHashMap、HashTable

不安全：HashMap、ArrayList、LinkedList

* 常见的数据结构有哪些。
* HashMap在jdk1.7和1.8的区别，为什么引入这个概念？hash碰撞怎么解决，为什么1.8要比1.7更好，好在哪？

1.7用的头插法，1.8用的尾插法。

因为JDK1.7是用单链表进行的纵向延伸，当采用头插法时会容易出现逆序且环形链表死循环问题。但是在JDK1.8之后是因为加入了红黑树使用尾插法，能够避免出现逆序且链表死循环的问题。

* 说说HashMap的原理, 以及HashMap如何扩充bucket的大小

它内部是基于哈希表实现的键值对存储，继承 AbstractMap 并且实现了 Map 接口。HashMap默认容量是16，在不指定大小的情况下，是默认容量\*加载因子（默认0.75），即在元素数量大于容量\*加载因子（12）的时候，就开始自动扩容一倍，变成32。然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作，所以如果我们已经预知hashmap中元素的个数，那么预设元素的个数能够有效的提高hashmap的性能

* Mysql索引的分类(Btree, hash)，各自使用什么情况 。

* 你有没有用过Spring的AOP? 是用来干嘛的? 大概会怎么使用？

面向切面编程，

* mysql数据库调优。
* 缓存
* 当只需要一条数据时，使用limit
* 避免select \*
* java对象四种引用。

（1）强引用: 创建一个对象并把这个对象赋给一个引用变量。强引用有引用变量指向时永远不会被垃圾回收，JVM宁愿抛出OutOfMemory错误也不会回收这种对象。可以显示地将引用赋值为null，这样一来的话，JVM在合适的时间就会回收该对象。比如Vector类的clear方法中就是通过将引用赋值为null来实现清理工作的：

* Java的垃圾回收机制，Java文件加载机制，tomcat类加载机制，锁机制，jvm原理及线上调优，jvm内存模型。
* SpringMVC如果希望把输出的Object(例如XXResult或者XXResponse)这种包装为JSON输出, 应该怎么处理?
* 线程池内部工作原理可以说一下么？
* 死锁是什么意思，形成条件是什么？出现死锁是可以通过什么方式去排查。

获取锁操作之后，又不释放锁，导致后面的线程无法操作。Redis远程指令拍它时一直存在，导致后续远程操作无法执行。

* 讲一下怎么使用分布式锁。

可以通过Redis来实现分布式锁

* 在交易过程中如何放在用户在支付时的重复支付（交叉支付），请写出你了解的方案或使用的过的方案。

用户带上标记支付记录的id,每次进行支付操作的时候，条件带上标记id,有这条用户记录再去做支付操作，更新数据库

* 数据库索引有哪几种，他们之间的区别。
* 了解哪些设计模式，用伪代码实现一个你熟悉的设计模式。。
* 说说Myisam, Innodb区别。
* 知道哪些负载均衡算法。