阿里面经

2016 年4月23日 14:35

1. 自我介绍
2. 做过的项目
   （Java 基础）
3. Java 的四个基本特性（抽象、封装、继承，多态），对多态的理解 (多态的实现方式)以及在项目中那些地方用到多态
   - Java 的四个基本特性
     - 抽象：抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。抽象只关注对象有哪些属性和行为，并不关注这些行为的细节是什么。
     - 继承：继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类、基类）；得到继承信息的类被称为子类（派生类）。继承让变化中的软件系统有了一定的延续性，同时继承也是封装程序中可变因素的重要手段。
     - 封装：通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。面向对象的本质就是将现实世界描绘成一系列完全自治、封闭的对象。我们在类中编写的方法就是对实现细节的一种封装；我们编写一个类就是对数据和数据操作的封装。可以说，封装就是隐藏一切可隐藏的东西，只向外界提供最简单的编程接口。
     - 多态性是指允许不同子类型的对象对同一消息作出不同的响应。
   - 多态的理解 (多态的实现方式)
     - 方法重载（ overload ）实现的是编译时的多态性（也称为前绑定）。
     - 方法重写（ override ）实现的是运行时的多态性（也称为后绑定）。运行时的多态是面向对象最精髓的东西。
     - 要实现多态需要做两件事： 1). 方法重写（子类继承父类并重写父类中已有的或抽象的方法）； 2). 对象造型（用父类型引用引用子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。
   - 项目中对多态的应用
     - 举一个简单的例子，在物流信息管理系统中，有两种用户：订购客户和卖房客户，两个客户都可以登录系统，他们有相同的方法 Login，但登陆之后他们会进入到不同的页面，也就是在登录的时候会有不同的操作，两种客户都继承父类的 Login 方法，但对于不同的对象，拥有不同的操作。
4. 面向对象和面向过程的区别？用面向过程可以实现面向对象吗？那是不是不能面向对象？
   - 面向对象和面向过程的区别
     - 面向过程就像是一个细心的管家，事无具细的都要考虑到。而面向对象就像是个家用电器，你只需要知道他的功能，不需要知道它的工作原理。
     - 面向过程 ”是一种是事件为中心的编程思想。就是分析出解决问题所需的步骤，然后用函数把这些步骤实现，并按顺序调用。面向对象是以 “对象 ”为中心的编程思想。
       - 简单的举个例子：汽车发动、汽车到站
         - 这对于 面向过程 来说，是两个事件，汽车启动是一个事件，汽车到站是另一个事件，面向过程编程的过程中我们关心的是事件，而不是汽车本身。针对上述两个事件，形成两个函数，之 后依次调用。
         - 然而这对于面向对象来说，我们关心的是汽车这类对象，两个事件只是这类对象所具有的行为。而且对于这两个行为的顺序没有强制要求。
   - 用面向过程可以实现面向对象吗
   - 那是不是不能面向对象
5. 重载和重写，如何确定调用哪个函数
   - 重载：重载发生在同一个类中，同名的方法如果有不同的参数列表（参数类型不同、参数个数不同或者二者都不同）则视为重载。
   - 重写：重写发生在子类与父类之间，重写要求子类被重写方法与父类被重写方法有相同的返回类型，比父类被重写方法 更好访问，不能比父类被重写方法声明更多的异常（里氏代换原则）。根据不同的子类对象确定调用的那个方法。
6. 面向对象开发的六个基本原则 (单一职责、开放封闭、里氏替换、依赖倒置、合成聚合复用、接口隔离)，迪米特法则。在项目中用过哪些原则
   - 六个基本原则
     - 单一职责：一个类只做它该做的事情 (高内聚)。在面向对象中，如果只让一个类完成它该做的事，而不涉及与它无关的领域就是践行了高内聚的原则，这个类就只有单一职责。
     - 开放封闭：软件实体应当对扩展开放，对修改关闭。要做到开闭有两个要点：①抽象是关键，一个系统中如果没有抽象类或接口系统就没有扩展点；②封装可变性，将系统中的各种可变因素封装到一个继承结构中，如果多个可变因素混杂在一起，系统将变得复杂而换乱。
     - 里氏替换：任何时候都可以用子类型替换掉父类型。子类一定是增加父类的能力而不是减少父类的能力，因为子类比父类的能力更多，把能力多的对象当成能力少的对象来用当然没有任何问题。
     - 依赖倒置：面向接口编程。（该原则说得直白和具体一些就是声明方法的参数类型、方法的返回类型、变量的引用类型时，尽可能使用抽象类型而不用具体类型，因为抽象类型可以被它的任何一个子类型所替代）
     - 合成聚和复用：优先使用聚合或合成关系复用代码。
     - 接口隔离：接口要小而专，绝不能大而全。臃肿的接口是对接口的污染，既然接口表示能力，那么一个接口只应该描述一种能力，接口也应该是高度内聚的。
   - 迪米特法则
     - 迪米特法则又叫最少知识原则，一个对象应当对其他对象有尽可能少的了解。
   - 项目中用到的原则
     - 单一职责、开放封闭、合成聚合复用 (最简单的例子就是 String 类)、接口隔离
7. static 和final 的区别和用途
   - Static
     - 修饰变量：静态变量随着类加载时被完成初始化，内存中只有一个，且 JVM 也只会为它分配一次内存，所有类共享静态变量。
     - 修饰方法：在类加载的时候就存在，不依赖任何实例； static 方法必须实现，不能用 abstract 修饰。

        - 修饰代码块：在类加载完之后就会执行代码块中的内容。
        - 父类静态代码块 ->子类静态代码块 ->父类非静态代码块 ->父类构造方法 ->子类非静态代码块 ->子类构造方法
    - Final
        - 修饰变量：
            - 编译期常量：类加载的过程完成初始化，编译后带入到任何计算式中。只能是基本类型。
            - 运行时常量：基本数据类型或引用数据类型。引用不可变，但引用的对象内容可变。
        - 修饰方法：不能被继承，不能被子类修改。
        - 修饰类：不能被继承。
        - 修饰形参： final 形参不可变
8. Hash Map 和Hash Table 的区别， Hash Map 中的key 可以是任何对象或数据类型吗？ HashTable 是线程安全的么？
    - Hash Map 和 Hash Table 的区别
        - Hashtable 的方法是同步的， HashMap 未经同步，所以在多线程场合要手动同步 HashMap 这个区别就像 Vector 和 ArrayList 一样。
        - Hashtable 不允许 null 值(key 和 value 都不可以)， HashMap 允许 null 值(key 和value 都可以)。
        - 两者的遍历方式大同小异， Hashtable 仅仅比 HashMap 多一个 elements 方法。
          Hashtable 和 HashMap 都能通过 values() 方法返回一个 Collection ，然后进行遍历处理。
          两者也都可以通过 entrySet() 方法返回一个 Set ， 然后进行遍历处理。
        - HashTable 使用 Enumeration ， HashMap 使用 Iterator 。
        - 哈希值的使用不同， Hashtable 直接使用对象的 hashCode 。而HashMap 重新计算 hash 值，而且用于代替求模。
        - Hashtable 中hash 数组默认大小是 11，增加的方式是 old*2+1 。HashMap 中hash 数组的默认大小是 16，而且一定是 2的指数。
        - HashTable 基于 Dictionary 类，而 HashMap 基于 AbstractMap 类
    - Hash Map 中的 key 可以是任何对象或数据类型吗
        - 可以为 null，但不能是可变对象，如果是可变对象的话，对象中的属性改变，则对象 HashCode 也进行相应的改变，导致下次无法查找到已存在 Map 中的数据。
        - 如果可变对象在 HashMap 中被用作键，那就要小心在改变对象状态的时候，不要改变它的哈希值了。我们只需要保证成员变量的改变能保证该对象的哈希值不变即可。
    - HashTable 是线程安全的么
        - HashTable 是线程安全的，其实现是在对应的方法上添加了 synchronized 关键字进行修饰，由于在执行此方法的时候需要获得对象锁，则执行起来比较慢。所以现在如果为了保证线程安全的话，使用 CurrentHashMap 。
9. HashMap 和Concurrent HashMap 区别， Concurrent HashMap 线程安全吗， Concurrent HashMap 如何保证 线程安全？
    - HashMap 和Concurrent HashMap 区别？
        - HashMap 是非线程安全的， CurrentHashMap 是线程安全的。
        - ConcurrentHashMap 将整个 Hash 桶进行了分段 segment ，也就是将这个大的数组分成了几个小的片段 segment ，而且每个小的片段 segment 上面都有锁存在，那么在插入元素的时候就需要先找到应该插入到哪一个片段 segment ，然后再在这个片段上面进行插入，而且这里还需要获取 segment 锁。
        - ConcurrentHashMap 让锁的粒度更精细一些，并发性能更好。
    - Concurrent HashMap 线程安全吗， Concurrent HashMap 如何保证 线程安全？
        - HashTable 容器在竞争激烈的并发环境下表现出效率低下的原因是所有访问 HashTable 的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是 ConcurrentHashMap 所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。
        - get操作的高效之处在于整个 get过程不需要加锁，除非读到的值是空的才会加锁重读。 get 方法里将要使用的共享变量都定义成 volatile ，如用于统计当前 Segement 大小的 count 字段和用于存储值的 HashEntry 的value 。定义成 volatile 的变量，能够在线程之间保持可见性，能够被多线程同时读，并且保证不会读到过期的值，但是只能被单线程写（有一种情况可以被多线程写，就是写入的值不依赖于原值），在 get操作里只需要读不需要写共享变量 count 和value ，所以可以不用加锁。
        - Put方法首先定位到 Segment ，然后在 Segment 里进行插入操作。插入操作需要经历两个步骤，第一步判断是否需要对 Segment 里的HashEntry 数组进行扩容，第二步定位添加元素的位置然后放在 HashEntry 数组里。

10. 因为别人知道源码怎么实现的，故意构造相同的 hash 的字符串进行攻击，怎么处理？那 jdk7 怎么办？
    - 怎么处理构造相同 hash 的字符串进行攻击 ？
        - 当客户端提交一个请求并附带参数的时候， web 应用服务器会把我们的参数转化成一个 HashMap 存储，这个HashMap 的逻辑结构如下： key1-->value1?
        - 但是物理存储结构是不同的， key 值会被转化成 Hashcode ，这个 hashcode 有会被转成数组的下标： 0-->value1 ；
        - 不同的 string 就会产生相同 hashcode 而导致碰撞，碰撞后的物理存储结构可能如下： 0-->value1-->value2?
        - 1、限制 post 和get 的参数个数，越少越好
          2、限制 post 数据包的大小
          3、 WAF
    - Jdk7 如何处理 hashcode 字符串攻击
        - HashMap 会动态的使用一个专门的 treemap 实现来替换掉它。
11. String 、 StringBuffer 、 StringBuilder 以及对 String 不变性的理解
    - String 、 StringBuffer 、 StringBuilder
        - 都是 final 类, 都不允许被继承 ？
        - String 长度是不可变的 , StringBuffer 、 StringBuilder 长度是可变的 ？
        - StringBuffer 是线程安全的 , StringBuilder 不是线程安全的，但它们两个中的所有方法都是相同的， StringBuffer 在StringBuilder 的方法之上添加了 synchronized 修饰，保证线程安全。
        - StringBuilder 比StringBuffer 拥有更好的性能。
        - 如果一个 String 类型的字符串，在编译时就可以确定是一个字符串常量，则编译完成之后，字符串会自动拼接成一个常量。此时 String 的速度比 StringBuffer 和StringBuilder 的性能好的多。
    - String 不变性的理解
        - String 类是被 final 进行修饰的，不能被继承。
        - 在用 +号链接字符串的时候会创建新的字符串。

- String s = new String("Hello world")? 可能创建两个对象也可能创建一个对象。如果静态区中有 “ Hello world字符串常量对象的话，则仅仅在堆中创建一个对象。如果静态区中没有 “ Hello world对象，则堆上和静态区中都需要创建对象。
- 在 java 中，通过使用 "+" 符号来串联字符串的时候 ，实际上底层会转成通过 StringBuilder 实例的 append() 方法来实现。

12. String 有重写 Object 的hashcode 和 toString 吗？如果重写 equals 不重写 hashcode 会出现什么问题？
    - String 有重写 Object 的hashcode 和toString 吗？
        - String 重写了 Object 类的 hashcode 和 toString 方法。
    - 当 equals 方法被重写时，通常有必要重写 hashCode 方法，以维护 hashCode 方法的常规协定，该协定声明相对等的两个对象必须有相同的 hashCode
        - object1.euqal(object2) 时为 true， object1.hashCode() == object2.hashCode() 为true
        - object1.hashCode() == object2.hashCode() 为false 时， object1.euqal(object2) 必定为 false
        - object1.hashCode() == object2.hashCode() 为true 时，但 object1.euqal(object2) 不一定定为 true
    - 重写 equals 不重写 hashcode 会出现什么问题
        - 在存储散列集合时 (如 Set 类)，如果原对象 .equals(新对象)，但没有对 hashCode 重写，即两个对象拥有不同的 hashCode ，则在集合中将会存储两个值相同的对象，从而导致混淆。因此在重写 equals 方法时，必须重写 hashCode 方法。

13. Java 序列化，如何实现序列化和反序列化，常见的序列化协议有哪些
    - Java 序列化定义
        - 将那些实现了 Serializable 接口的对象转换成一个字节序列，并能够在以后将这个字节序列完全恢复为原来的对象，序列化可以弥补不同操作系统之间的差异。
    - Java 序列化的作用
        - Java 远程方法调用（ RMI ）
        - 对 JavaBeans 进行序列化
    - 如何实现序列化和反序列化
        - 实现序列化方法
            - 实现 Serializable 接口
                - 该接口只是一个可序列化的标志，并没有包含实际的属性和方法。
                - 如果不在改方法中添加 readObject() 和 writeObject() 方法，则采取默认的序列化机制。如果添加了这两个方法之后还想利用 Java 默认的序列化机制，则在这两个方法中分别调用 defaultReadObject() 和 defaultWriteObject() 两个方法。
                - 为了保证安全性，可以使用 transient 关键字进行修饰不必序列化的属性。因为在反序列化时， private 修饰的属性也能发查看到。
            - 实现 ExternalSerializable 方法
                - 自己对要序列化的内容进行控制，控制那些属性能被序列化，那些不能被序列化。
        - 反序列化
            - 实现 Serializable 接口的对象在反序列化时不需要调用对象所在类的构造方法，完全基于字节。
            - 实现 externalSerializable 接口的方法在反序列化时会调用构造方法。
        - 注意事项
            - 被static 修饰的属性不会被序列化
            - 对象的类名、属性都会被序列化，方法不会被序列化
            - 要保证序列化对象所在类的属性也是可以被序列化的
            - 当通过网络、文件进行序列化时，必须按照写入的顺序读取对象。
            - 反序列化时必须有序列化对象时的 class 文件
            - 最好显示的声明 serializableID ，因为在不同的 JVM 之间，默认生成 serializableID 可能不同，会造成反序列化失败。
    - 常见的序列化协议有哪些
        - COM 主要用于 Windows 平台，并没有真正实现跨平台，另外 COM 的序列化的原理利用了编译器中虚表，使得其学习成本巨大。
        - CORBA 是早期比较好的实现了跨平台，跨语言的序列化协议。 COBRA 的主要问题是参与方过多带来的版本过多，版本之间兼容性较差，以及使用复杂晦涩。
        - XML&SOAP
            - XML 是一种常用的序列化和反序列化协议，具有跨机器，跨语言等优点。
            - SOAP （Simple Object Access protocol ） 是一种被广泛应用的，基于 XML 为序列化和反序列化协议的结构化消息传递协议。 SOAP 具有安全、可扩展、跨语言、跨平台并支持多种传输层协议。
        - JSON （Javascript Object Notation ）
            - 这种 Associative array 格式非常符合工程师对对象的理解。
            - 它保持了 XML 的人眼可读（ Human-readable ）的优点。
            - 相对于 XML 而言，序列化后的数据更加简洁。
            - 它具备 Javascript 的先天性支持，所以被广泛应用于 Web browser 的应用常景中，是 Ajax 的事实标准协议。
            - 与XML 相比，其协议比较简单，解析速度比较快。
            - 松散的 Associative array 使得其具有良好的可扩展性和兼容性。
        - Thrift 是Facebook 开源提供的一个高性能，轻量级 RPC 服务框架，其产生正是为了满足当前大数据量、分布式、跨语言、跨平台数据通讯的需求。 Thrift 在空间开销和解析性能上有了比较大的提升，对于对性能要求比较高的分布式系统，它是一个优秀的 RPC 解决方案；但是由于 Thrift 的序列化被嵌入到 Thrift 框架里面， Thrift 框架本身并没有透出序列化和反序列化接口，这导致其很难和其他传输层协议共同使用
        - Protobuf 具备了优秀的序列化协议的所需的众多典型特征
            - 标准的 IDL 和IDL 编译器，这使得其对工程师非常友好。
            - 序列化数据非常简洁，紧凑，与 XML 相比，其序列化之后的数据量约为 1/3 到1/10 。
            - 解析速度非常快，比对应的 XML 快约 20-100 倍。
            - 提供了非常友好的动态库，使用非常简介，反序列化只需要一行代码。由于其解析性能高，序列化后数据量相对少，非常适合应用层对象的持久化场景
        - Avro 的产生解决了 JSON 的冗长和没有 IDL 的问题， Avro 属于 Apache Hadoop 的一个子项目。 Avro 提供两种序列化格式： JSON 格式或者 Binary 格式。 Binary 格式在空间开销和解析性能方面可以和 Protobuf 媲美， JSON 格式方便测试阶段

的调试。适合于高性能的序列化服务。
- 几种协议的对比
  - XML 序列化（ Xstream ）无论在性能和简洁性上比较差；
  - Thrift 与Protobuf 相比在时空开销方面都有一定的劣势；
  - Protobuf 和Avro 在两方面表现都非常优越。

14. Java 实现多线程的方式及三种方式的区别
    - 实现多线程的方式
      - 继承 Thread 类，重写 run函数。
      - 实现 Runnable 接口
      - 实现 Callable 接口
    - 三种方式的区别
      - 实现 Runnable 接口可以避免 Java 单继承特性而带来的局限；增强程序的健壮性，代码能够被多个线程共享，代码与数据是独立的；适合多个相同程序代码的线程区处理同一资源的情况。
      - 继承 Thread 类和实现 Runnable 方法启动线程都是使用 start 方法，然后 JVM 虚拟机将此线程放到就绪队列中，如果有处理机可用，则执行 run 方法。
      - 实现 Callable 接口要实现 call 方法，并且线程执行完毕后会有返回值。其他的两种都是重写 run方法，没有返回值。

15. 线程安全
    - 定义
      - 某个类的行为与其规范一致。
      - 不管多个线程是怎样的执行顺序和优先级 ,或是 wait,sleep,join 等控制方式 ,，如果一个类在多线程访问下运转一切正常，并且访问类不需要进行额外的同步处理或者协调，那么我们就认为它是线程安全的。
    - 如何保证线程安全？ 6
      - 对变量使用 volitate
      - 对程序段进行加锁 (synchronized,lock)
    - 注意
      - 非线程安全的集合在多线程环境下可以使用，但并不能作为多个线程共享的属性，可以作为某个线程独享的属性。
      - 例如 Vector 是线程安全的， ArrayList 不是线程安全的。如果每一个线程中 new 一个 ArrayList ， 而这个 ArrayList 只是在这一个线程中使用，肯定没问题。
16. 多线程如何进行信息交互
    - Object 中的方法， wait()， notify()， notifyAll()?
17. 多线程共用一个数据变量需要注意什么？
    - 当我们在线程对象（ Runnable ）中定义了全局变量， run方法会修改该变量时，如果有多个线程同时使用该线程对象，那么就会造成全局变量的值被同时修改，造成错误 .
    - ThreadLocal 是JDK引入的一种机制，它用于解决线程间共享变量，使用 ThreadLocal 声明的变量，即使在线程中属于全局变量，针对每个线程来讲，这个变量也是独立的。
    - volatile 变量每次被线程访问时，都强迫线程从主内存中重读该变量的最新值，而当该变量发生修改变化时，也会强迫线程将最新的值刷新回主内存中。这样一来，不同的线程都能及时的看到该变量的最新值。
18. 什么是线程池？如果让你设计一个动态大小的线程池，如何设计，应该有哪些方法？
    - 什么是线程池
      - 线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。
    - 设计一个动态大小的线程池，如何设计，应该有哪些方法
      - 一个线程池包括以下四个基本组成部分：
        - 线程管理器 (ThreadPool) ：用于创建并管理线程池，包括创建线程，销毁线程池，添加新任务；
        - 工作线程 (PoolWorker) ：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；
        - 任务接口 (Task) ：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；
        - 任务队列 (TaskQueue) ：用于存放没有处理的任务。提供一种缓冲机制；
      - 所包含的方法
        - private ThreadPool() 创建线程池
        - public static ThreadPool getThreadPool() 获得一个默认线程个数的线程池
        - public void execute(Runnable task) 执行任务 ,其实只是把任务加入任务队列，什么时候执行有线程池管理器决定
        - public void execute(Runnable[] task) 批量执行任务 ,其实只是把任务加入任务队列，什么时候执行有线程池管理器决定
        - public void destroy() 销毁线程池 ,该方法保证在所有任务都完成的情况下才销毁所有线程，否则等待任务完成才销毁
        - public int getWorkThreadNumber() 返回工作线程的个数
        - public int getFinishedTasknumber() 返回已完成任务的个数 ,这里的已完成是只出了任务队列的任务个数，可能该任务并没有实际执行完成
        - public void addThread() 在保证线程池中所有线程正在执行，并且要执行线程的个数大于某一值时。增加线程池中线程的个数
        - public void reduceThread() 在保证线程池中有很大一部分线程处于空闲状态，并且空闲状态的线程在小于某一值时，减少线程池中线程的个数

19. Java 是否有内存泄露和内存溢出
    - 静态集合类，使用 Set、 Vector 、 HashMap 等集合类的时候需要特别注意。当这些类被定义成静态的时候，由于他们的生命周期跟应用程序一样长，这时候就有可能发生内存泄漏。
      例子

```
class StaticTest
{
    private static Vector v = new Vector( 10 )?

public void init ()
```

```
    {
        for  (int  i =  1? i < 100 ? i++)
       {
          Object     object   = new  Object()?
          v.add(   object  )?
            object  =  null ?
       }
    }
}
```

在上面的代码中，循环申请了 Object 对象，并添加到 Vector 中，然后设置为 null，可是这些对象呗 vector 引用着，因此必能被 GC 回收，因此造成内存泄漏。因此要释放这些对象，还需要将它们从 vector 删除，最简单的方法就是将 vector 设置为 null

- 监听器： 在Java 编程中，我们都需要和监听器打交道，通常一个应用中会用到很多监听器，我们会调用一个控件，诸如 addXXXListener() 等方法来增加监听器，但往往在释放的时候却没有去删除这些监听器，从而增加了内存泄漏的机会。

- 物理连接：一些物理连接，比如数据库连接和网络连接，除非其显式的关闭了连接，否则是不会自动被 GC 回收的。 Java 数据库连接一般用 DataSource.getConnection() 来创建，当不再使用时必须用 Close() 方法来释放， 因为这些连接是独立于 JVM 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。。一般情况下，在 try 代码块里创建连接，在 finally 里释放连接，就能够避免此类内存泄漏。

- 内部类和外部模块等的引用：内部类的引用是比较容易遗忘的一种，而且一旦没释放可能导致一系列的后继类对象没有释放。在调用外部模块的时候，也应该注意防止内存泄漏，如果模块 A 调用了外部模块 B的一个方法，如：
  public void register(Object o)
  这个方法有可能就使得 A模块持有传入对象的引用，这时候需要查看 B模块是否提供了出去引用的方法，这种情况容易忽略，而且发生内存泄漏的话，还比较难察觉。

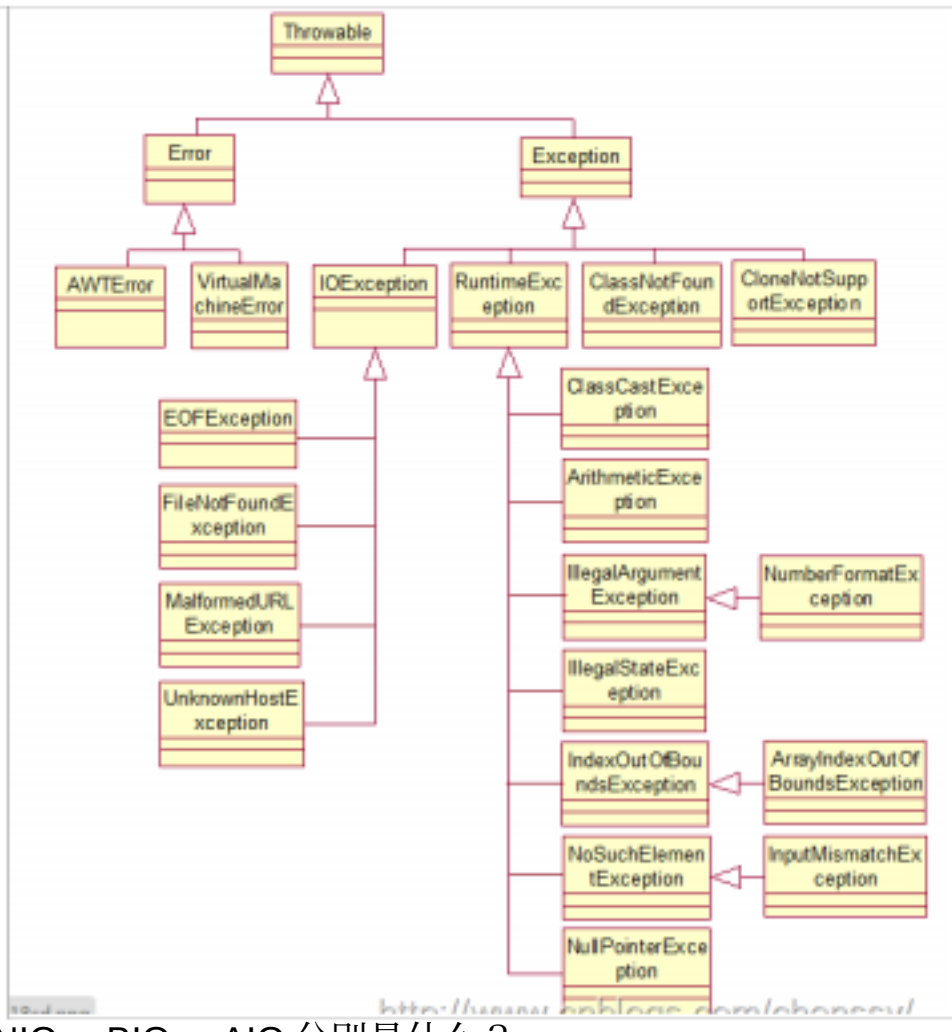
- 单例模式：因为单利对象初始化后将在 JVM 的整个生命周期内存在，如果它持有一个外部对象的（生命周期比较短）引用，那么这个外部对象就不能被回收，从而导致内存泄漏。如果这个外部对象还持有其他对象的引用，那么内存泄漏更严重。

20. concurrent 包下面，都用过什么？
    - concurrent 下面的包
      - Executor 用来创建线程池，在实现 Callable 接口时，添加线程。
      - FeatureTask 此 FutureTask 的 get 方法所返回的结果类型。
      - TimeUnit
      - Semaphore
      - LinkedBlockingQueue
    - 所用过的类
      - Executor

21. volatile 关键字的如何保证内存可见性
    - volatile 关键字的作用
      - 保证内存的可见性
      - 防止指令重排
      - 注意： volatile 并不保证原子性
    - 内存可见性
      - volatile 保证可见性的原理是在每次访问变量时都会进行一次刷新，因此每次访问都是主内存中最新的版本。所以 volatile 关键字的作用之一就是保证变量修改的实时可见性。
    - 当且仅当满足以下所有条件时，才应该使用 volatile 变量
      - 对变量的写入操作不依赖变量的当前值，或者你能确保只有单个线程更新变量的值。
      - 该变量没有包含在具有其他变量的不变式中。
    - volatile 使用建议
      - 在两个或者更多的线程需要访问的成员变量上使用 volatile 。当要访问的变量已在 synchronized 代码块中，或者为常量时，没必要使用 volatile 。
      - 由于使用 volatile 屏蔽掉了 JVM 中必要的代码优化，所以在效率上比较低，因此一定在必要时才使用此关键字。
    - volatile 和synchronized 区别
      - volatile 不会进行加锁操作：
        volatile 变量是一种稍弱的同步机制在访问 volatile 变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此 volatile 变量是一种比 synchronized 关键字更轻量级的同步机制。
      - volatile 变量作用类似于同步变量读写操作：
        从内存可见性的角度看，写入 volatile 变量相当于退出同步代码块，而读取 volatile 变量相当于进入同步代码块。
      - volatile 不如 synchronized 安全：
        在代码中如果过度依赖 volatile 变量来控制状态的可见性，通常会比使用锁的代码更脆弱，也更难以理解。仅当 volatile 变量能简化代码的实现以及对同步策略的验证时，才应该使用它。一般来说，用同步机制会更安全些。
      - volatile 无法同时保证内存可见性和原则性：
        加锁机制（即同步机制）既可以确保可见性又可以确保原子性，而 volatile 变量只能确保可见性，原因是声明

为volatile 的简单变量如果当前值与该变量以前的值相关，那么 volatile 关键字不起作用，也就是说如下的表达式都不是原子操作： “ count++、 “ ”count = count+1。 ”

22. sleep 和wait 分别是那个类的方法，有什么区别
    - sleep 和 wait
        - sleep 是Thread 类的方法
        - wait 是Object 类的方法
    - 有什么区别
        - sleep() 方法（休眠）是线程类（ Thread ）的静态方法，调用此方法会让当前线程暂停执行指定的时间，将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复（线程回到就绪状态）。
        - wait() 是Object 类的方法，调用对象的 wait() 方法导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池（wait pool ），只有调用对象的 notify() 方法（或 notifyAll() 方法）时才能唤醒等待池中的线程进入等锁池（ lock pool），如果线程重新获得对象的锁就可以进入就绪状态。
23. synchronized 与lock的区别，使用场景。看过 synchronized 的源码没？
    - synchronized 与lock 的区别
        - （用法） synchronized(隐式锁)：在需要同步的对象中加入此控制， synchronized 可以加在方法上，也可以加在特定代码块中，括号中表示需要锁的对象。
        - （用法） lock （显示锁）：需要显示指定起始位置和终止位置。一般使用 ReentrantLock 类做为锁，多个线程中必须要使用一个 ReentrantLock 类做为对 象才能保证锁的生效。且在加锁和解锁处需要通过 lock() 和unlock() 显示指出。所以一般会在 finally 块中写 unlock() 以防死锁。
        - （性能） synchronized 是托管给 JVM 执行的，而 lock 是java 写的控制锁的代码。在 Java1.5 中， synchronize 是性能低效的。因为 这是一个重量级操作，需要调用操作接口，导致有可能加锁消耗的系统时间比加锁以外的操作还多。相比之下使用 Java 提供的 Lock 对象，性能更高一些。但 是到了 Java1.6 ， 发生了变化。 synchronize 在语义上很清晰，可以进行很多优化，有适应自旋，锁消除，锁粗化，轻量级锁，偏向锁等等。导致 在Java1.6 上synchronize 的性能并不比 Lock 差。
        - （机制） synchronized 原始采用的是 CPU 悲观锁机制，即线程获得的是独占锁。独占锁意味着其 他线程只能依靠阻塞来等待线程释放锁。 Lock 用的是乐观锁方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁实现的机制就 是CAS 操作（ Compare and Swap ）。
24. synchronized 底层如何实现的？用在代码块和方法上有什么区别？
    - synchronized 底层如何实现的
    - 用在代码块和方法上有什么区别？
        - synchronized 用在代码块锁的是调用该方法的对象（ this），也可以选择锁住任何一个对象。
        - synchronized 用在方法上锁的是调用该方法的对象，
        - synchronized 用在代码块可以减小锁的粒度，从而提高并发性能。
        - 无论用在代码块上还是用在方法上，都是获取对象的锁；每一个对象只有一个锁与之相关联；实现同步需要很大的系统开销作为代价，甚至可能造成死锁，所以尽量避免无谓的同步控制。
    - synchronized 与static synchronized 的区别
        - synchronized 是对类的当前 实例 进行加锁，防止其他线程同时访问该类的该实例的所有 synchronized 块，同一个类的两个不同实例就没有这种约束了。
        - 那么 static synchronized 恰好就是要控制类的所有实例的访问了， static synchronized 是限制线程同时访问 jvm 中该类的所有实例同时访问对应的代码快。
25. 常见异常分为那两种 (Exception ， Error) ，常见异常的基类以及常见的异常
    - Throwable 是java 语言中所有错误和异常的超类（万物即可抛）。它有两个子类： Error 、 Exception 。
    - 异常种类
        - Error ：Error 为错误，是程序无法处理的，如 OutOfMemoryError 、ThreadDeath 等，出现这种情况你唯一能做的就是听之任之，交由 JVM 来处理，不过 JVM 在大多数情况下会选择终止线程。
        - Exception ：Exception 是程序可以处理的异常。它又分为两种 CheckedException （受捡异常），一种是UncheckedException （不受检异常）。
            - CheckException 发生在编译阶段，必须要使用 try …catch（或者 throws ）否则编译不通过。
            - UncheckedException 发生在运行期，具有不确定性，主要是由于程序的逻辑问题所引起的，难以排查，我们一般都需要纵观全局才能够发现这类的异常错误，所以在程序设计中我们需要认真考虑，好好写代码，尽量处理异常，即使产生了异常，也能尽量保证程序朝着有利方向发展。
    - 常见异常的基类
        - IOException
        - RuntimeException
    - 常见的异常



26. Java 中的 NIO， BIO， AIO 分别是什么？
    - BIO
        - 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。
        - BIO 方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4 以前的唯一选择，但程序直观简单易理解。
    - NIO
        - 同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有 I/O 请求时才启动一个线程进行处理。
        - NIO 方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂， JDK1.4 开始支持。
    - AIO
        - 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的 I/O 请求都是由 OS 先完成了再通知服务器应用去启动线程进行处理 .
        - AIO 方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用 OS 参与并发操作，编程比较复杂， JDK7 开始支持。
27. 所了解的设计模式，单例模式的注意事项， jdk 源码哪些用到了你说的设计模式
    - 所了解的设计模式
        - 工厂模式：定义一个用于创建对象的接口，让子类决定实例化哪一个类， Factory Method 使一个类的实例化延迟到了子类。
        - 单例模式：保证一个类只有一个实例，并提供一个访问它的全局访问点；
        - 适配器模式：将一类的接口转换成客户希望的另外一个接口， Adapter 模式使得原本由于接口不兼容而不能一起工作那些类可以一起工作。
        - 装饰者模式：动态地给一个对象增加一些额外的职责，就增加的功能来说， Decorator 模式相比生成子类更加灵活。
        - 代理：为其他对象提供一种代理以控制对这个对象的访问
        - 迭代器模式：提供一个方法顺序访问一个聚合对象的各个元素，而又不需要暴露该对象的内部表示。
    - 单例模式的注意事项
        - 尽量使用懒加载
        - 双重检索实现线程安全
        - 构造方法为 private
        - 定义静态的 Singleton instance 对象和 getInstance() 方法
    - jdk 源码中用到的设计模式
        - 装饰器模式： IO 流中
        - 迭代器模式： Iterator
        - 单利模式： java.lang.Runtime
        - 代理模式： RMI
28. 匿名内部类是什么？如何访问在其外面定义的变量？
    - 匿名内部类是什么？
        - 匿名内部类是没有访问修饰符的。
        - 所以当所在方法的形参需要被匿名内部类使用，那么这个形参就必须为 final
        - 匿名内部类是没有构造方法的。因为它连名字都没有何来构造方法。
    - 如何访问在其外面定义的变量？
        - 所以当所在方法的形参需要被匿名内部类使用，那么这个形参就必须为 final
29. 如果你定义一个类，包括学号，姓名，分数，如何把这个对象作为 key？要重写 equals 和 hashcode 吗
    - 需要重写 equals 方法和 hashcode ，必须保证对象的属性改变时，其 hashcode 不能改变。
30. 为什么要实现内存模型？
    - 内存模型的就是为了在现代计算机平台中保证程序可以正确性的执行，但是不同的平台实现是不同的。
    - 编译器中生成的指令顺序， 可以与源代码中的顺序不同；
    - 编译器可能把变量保存在寄存器而不是内存中；
    - 处理器可以采用乱序或并行等方式来执行指令；
    - 缓存可能会改变将写入变量提交到主内存的次序；

- 保存在处理器本地缓存中的值，对其他处理器是不可见的；

（数据库）

31. 常用的数据库有哪些？ redis 用过吗？
    - 常用的数据库
        - MySQL
        - SQLServer
    - Redis
        - Redis 是一个速度非常快的非关系型数据库，他可以存储键 (key) 与5种不同类型的值（ value ）之间的映射，可以将存储在内存中的键值对数据持久化到硬盘中。
        - 与Memcached 相比
            - 两者都可用于存储键值映射，彼此性能也相差无几
            - Redis 能够自动以两种不同的方式将数据写入硬盘
            - Redis 除了能存储普通的字符串键之外，还可以存储其他 4种数据结构， memcached 只能存储字符串键
            - Redis 既能用作主数据库，由可以作为其他存储系统的辅助数据库
32. 数据库索引的优缺点以及什么时候数据库索引失效
    - 索引的特点
        - 可以加快数据库的检索速度
        - 降低数据库插入、修改、删除等维护的速度
        - 只能创建在表上，不能创建到视图上
        - 既可以直接创建又可以间接创建
        - 可以在优化隐藏中使用索引
        - 使用查询处理器执行 SQL 语句，在一个表上，一次只能使用一个索引
    - 索引的优点
        - 创建唯一性索引，保证数据库表中每一行数据的唯一性
        - 大大加快数据的检索速度，这是创建索引的最主要的原因
        - 加速数据库表之间的连接，特别是在实现数据的参考完整性方面特别有意义
        - 在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间
        - 通过使用索引，可以在查询中使用优化隐藏器，提高系统的性能
    - 索引的缺点
        - 创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加
        - 索引需要占用物理空间，除了数据表占用数据空间之外，每一个索引还要占一定的物理空间，如果建立聚簇索引，那么需要的空间就会更大
        - 当对表中的数据进行增加、删除和修改的时候，索引也需要维护，降低数据维护的速度
    - 索引分类
        - 直接创建索引和间接创建索引
        - 普通索引和唯一性索引
        - 单个索引和复合索引
        - 聚簇索引和非聚簇索引
    - 索引失效
        - 如果条件中有 or，即使其中有条件带索引也不会使用 (这就是问什么尽量少使用 or的原因)
        - 对于多列索引，不是使用的第一部分，则不会使用索引
        - like查询是以 %开头
        - 如果列类型是字符串，那一定要在条件中使用引号引起来，否则不会使用索引
        - 如果 mysql 估计使用全表扫秒比使用索引快，则不适用索引。
    - 各引擎支持索引

MyISAM、InnoDB引擎、Memory三个常用引擎类型比较

索引	MyISAM引擎	InnoDB引擎	Memory引擎
B-Tree 索引	支持	支持	支持
HASH 索引	不支持	不支持	支持
R-Tree 索引	支持	不支持	不支持
Full-text 索引	不支持	暂不支持	不支持

33. 事务隔离级别
    - 串行化 (Serializable) ：所有事务一个接着一个的执行，这样可以避免幻读 (phantom read)，对于基于锁来实现并发控制的数据库来说，串行化要求在执行范围查询的时候，需要获取范围锁，如果不是基于锁实现并发控制的数据库，则检查到有违反串行操作的事务时，需回滚该事务。
    - 可重复读 (Repeated Read) ：所有被 Select 获取的数据都不能被修改，这样就可以避免一个事务前后读取不一致的情况。但是没有办法控 制幻读 ，因为这个时候其他事务不能更改所选的数据，但是可以增加数据，因为强恶意事务没有范围锁
    - 读已提交 (Read Committed) ：被读取的数据可以被其他事务修改，这样可能导致 不可重复读 。也就是说，事务读取的时候获取读锁，但是在读完之后立即释放 (不需要等事务结束)，而写锁则是事务提交之后才释放，释放读锁之后，就可能被其他事务修改数据。改等级也是 SQL Server 默认的隔离等级
    - 读未提交 (Read Uncommitted) ：最低的隔离等级，允许其他事务看到没有提交的数据，会导致 脏读
    - 总结
        - 四个级别逐渐增强，每个级别解决一个问题，每个级别解决一个问题，事务级别遇到，性能越差，大多数环境 (Read committed 就可以用了)

隔离级别	脏读（Dirty Read）	不可重复读（NonRepeatable Read）	幻读（Phantom Read）
未提交读（Read uncommitted）	可能	可能	可能
已提交读（Read committed）	不可能	可能	可能
可重复读（Repeatable read）	不可能	不可能	可能
可串行化（Serializable ）	不可能	不可能	不可能

34. 数据库中的范式有哪些？
    - 目前关系数据库有六种范式：第一范式（ 1NF）、第二范式（ 2NF）、第三范式（ 3NF）、巴斯 -科德范式（BCNF）、第四范式 (4NF）和第五范式（ 5NF，又称完美范式）。满足最低要求的范式是第一范式（ 1NF）。在第一范式的基础上进一步满足更多规范要求的称为第二范式（ 2NF），其余范式以次类推。一般说来，数据库只需满足第三范式 (3NF）就行了。
    - 范式的包含关系。一个数据库设计如果符合第二范式，一定也符合第一范式。如果符合第三范式，一定也符合第二范式 …
    - 范式
        - 1NF ：符合 1NF 的关系中的每个属性都不可再分
        - 2NF ：属性完全依赖于主键 [消除部分子函数依赖]
        - 3NF ：属性不依赖于其它非主属性 [消除传递依赖]
        - BCNF ：在 1NF 基础上，任何非主属性不能对主键子集依赖 [在 3NF 基础上消除对主码子集的依赖]
        - 4NF ：要求把同一表内的多对多关系删除。
        - 5NF ：从最终结构重新建立原始结构。
35. 数据库中的索引的结构？什么情况下适合建索引？
    - 数据库中的索引结构
        - 因为在使用二叉树的时候，由于二叉树的深度过大而造成 I/O 读写过于频繁，进而导致查询效率低下。因此采用多叉树结构。 B 树的各种操作能使 B 树能保持较低的高度。
    - B 树又叫平衡多路查找树，一棵 m 阶的 B 树的特性如下
        - 树中每个结点最多含有 m 个孩子（ m>=2）；
        - 除根结点和叶子结点外，其他每个结点至少有 [ceil(m / 2)] 个孩子（其中 ceil(x) 是一个取上限的函数）；
        - 根结点至少有 2 个孩子（除非 B 树只包含一个结点：根结点）；
        - 所有叶子结点都出现在同一层， 叶子结点不包含任何关键字信息 (可以看做是外部结点或查询失败的结点，指向这些结点的指针都为 null)；（注：叶子节点只是没有孩子和指向孩子的指针，这些节点也存在，也有元素。类似红黑树中，每一个 NULL 指针即当做叶子结点，只是没画出来而已）。
        - 每个非终端结点中包含有 n 个关键字信息： (n， P0， K1， P1， K2， P2， ..... ， Kn， Pn)。其中：

          a) Ki (i=1...n) 为关键字，且关键字按顺序升序排序 K(i-1)< Ki 。

          b) Pi 为指向子树根的结点，且指针 P(i-1) 指向子树种所有结点的关键字均小于 Ki，但都大于 K(i-1)。

          c) 关键字的个数 n 必须满足： [ceil(m / 2)-1]<= n <= m-1 。比如有 j 个孩子的非叶结点恰好有 j-1 个关键码。

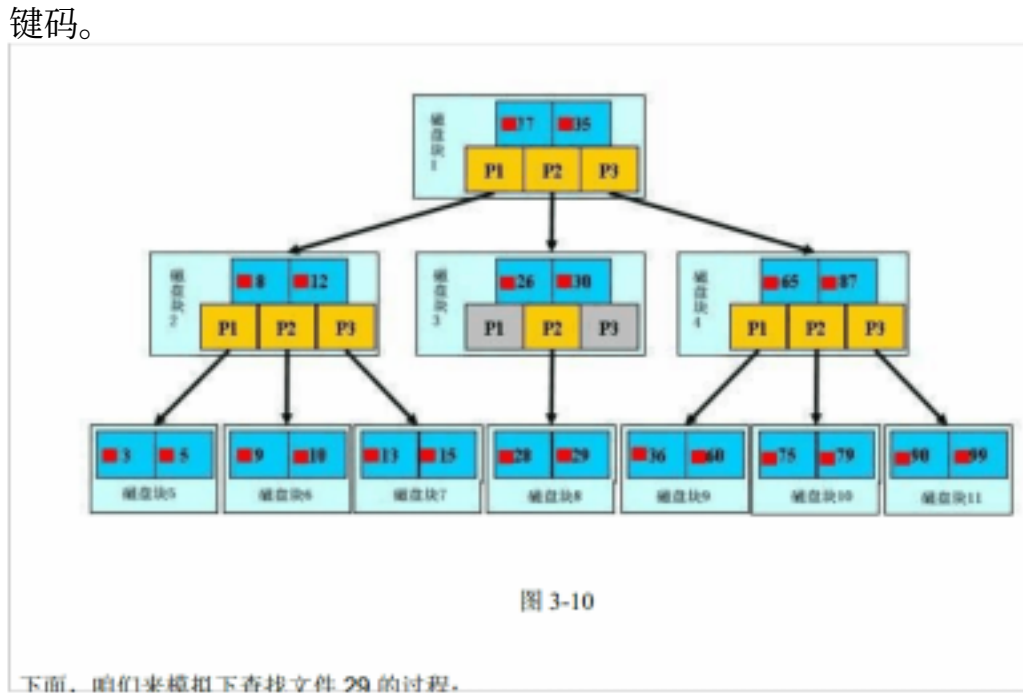
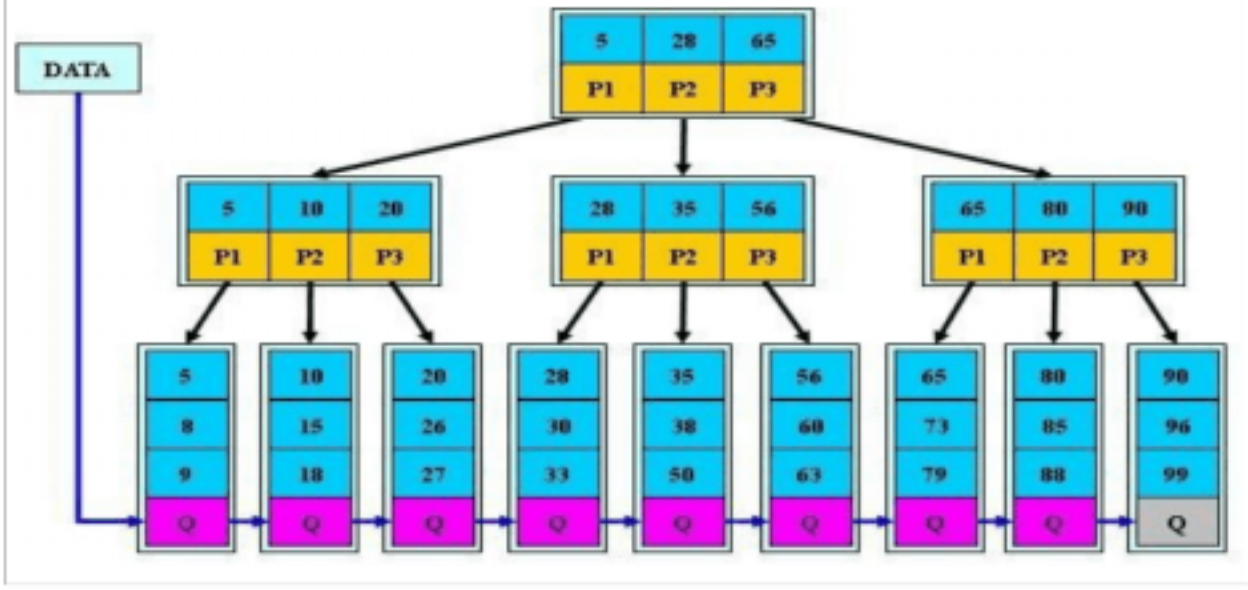


图 3-10

下面，咱们来模拟下查找文件 29 的过程。

- B+ 树



- 在什么情况下适合建立索引

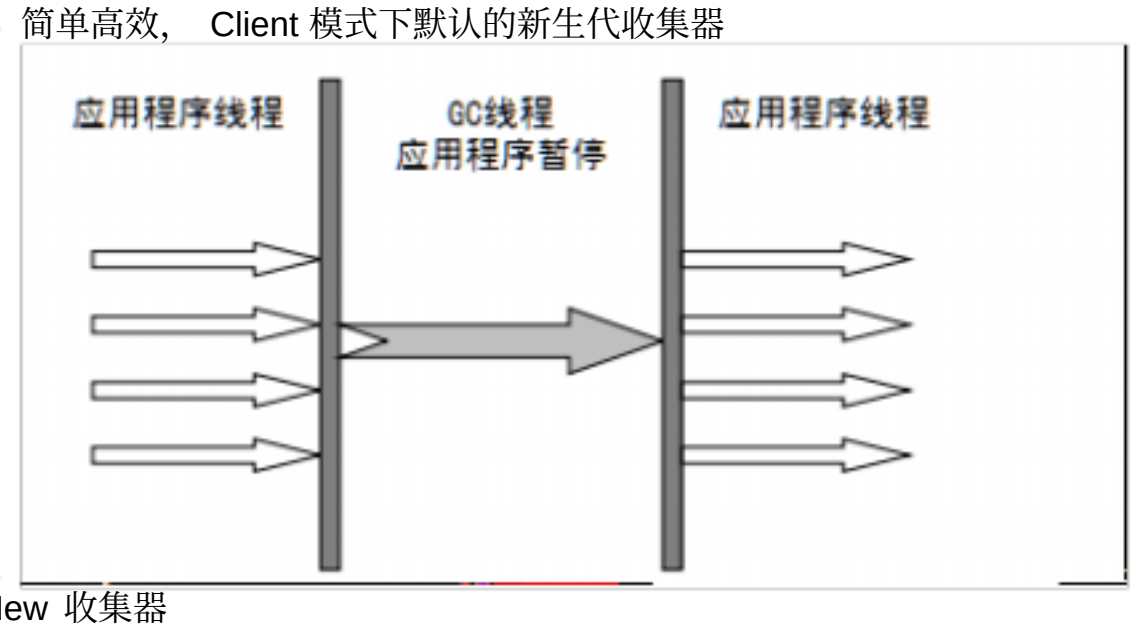
- 为经常出现在关键字 order by 、group by 、distinct 后面的字段，建立索引。
- 在 union 等集合操作的结果集字段上，建立索引。其建立索引的目的同上。
- 为经常用作查询选择的字段，建立索引。
- 在经常用作表连接的属性上，建立索引。
- 考虑使用索引覆盖。对数据很少被更新的表，如果用户经常只查询其中的几个字段，可以考虑在这几个字段上建立索引，从而将表的扫描改变为索引的扫描。

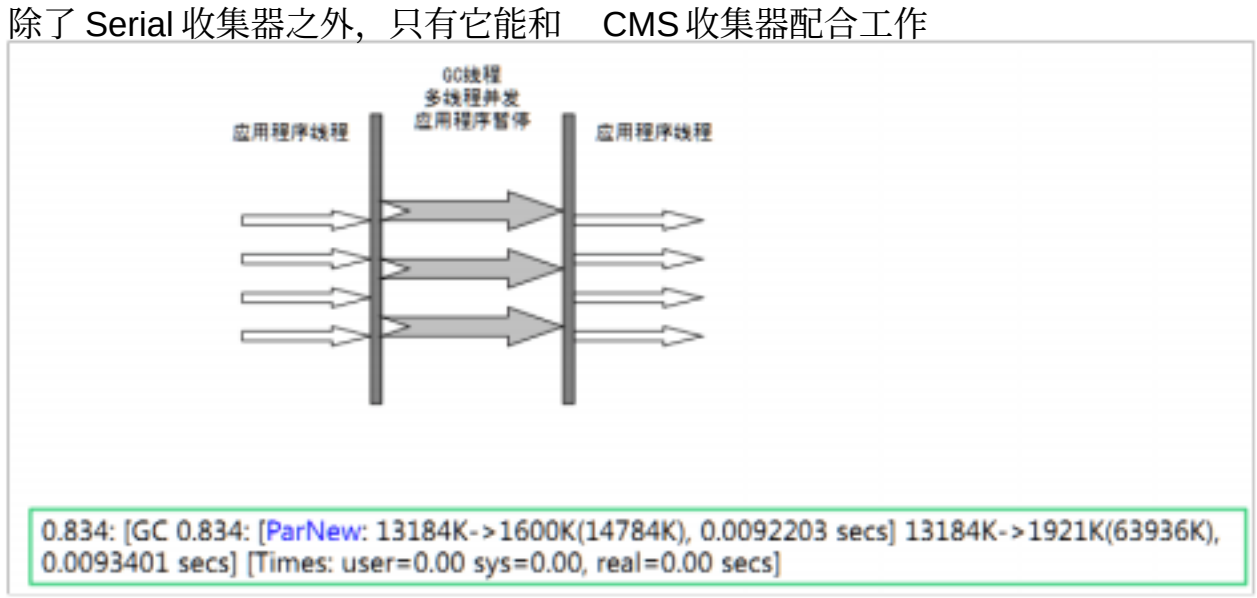
36. Redis 的存储结构，或者说如何工作的，与 mysql 的区别？有哪些数据类型？
    - Redis 的数据结构
        - STRING ：可以是字符串、整数或者浮点数
        - LIST ：一个链表，链表上的每个节点都包含了一个字符串
        - SET ：包含字符串的无序收集器（ unordered collection ），并且被包含的每个字符串都是独一无二、各不相同的
        - HAST ：包含键值对的无序散列表
        - ZSET ：字符串成员（ member ）与浮点数分值（ score ）之间的有序映射，元素的排列顺序由分值的大小决定
37. 数据库中的分页查询语句怎么写？ http://qimo601.iteye.com/blog/1634748
    - Mysql 的 limit用法
        - SELECT * FROM table LIMIT [offset,] rows | rows OFFSET offset
        - LIMIT 接受一个或两个数字参数。参数必须是一个整数常量。如果给定两个参数，第一个参数指定第一个返回记录行的偏移量，第二个参数指定返回记录行的最大数目。初始记录行的偏移量是 0(而不是 1)
    - 最基本的分页方式： SELECT ... FROM ... WHERE ... ORDER BY ... LIMIT ...
    - 子查询的分页方式：
38. 数据库 ACID
    - 原子性 (Atomicity) ：保证事务中的所有操作全部执行或全部不执行
    - 一致性 (Consistency) ：保证数据库始终保持数据的一致性 —— 事务操作之前和之后都是一致的
    - 隔离性 (Isolation) ：多个事务并发执行的话，结果应该与多个事务串行执行效果是一样的
    - 持久性 (Durability) ：事务操作完成之后，对数据库的影响是持久的，即使数据库因故障而受到破坏，数据库也能够恢复 (日志)
39. 脏读、不可重复读和幻读
    - 脏读：事务 T1 更新了一行记录的内容，但是并没有提交所做的修改。事务 T2 读取更新后的行，然后 T1执行了回滚操作，取消了刚才所做的修改。现在 T2 读取的行就无效了（一个事务读取了另一个事务未提交的数据）
    - 不可重复读：事务 T1 读取了一行记录，紧接着 T2修改 了T1 刚才读取的那一行记录，然后 T1 又再次读取这行记录，发现与刚才读取的结果不同。
    - 幻读：事务 T1 读取一条指定的 Where 子句所返回的结果集，然后 T2事务新插入一行记录，这行记录恰好可以满足 T1 所使用的查询条件。然后 T1再次对表进行检索，但又看到了 T2 插入的数据。
40. MyISAM 和InnoDB 引擎的区别
    - 主要区别：
        - MyISAM 是非事务安全型的，而 InnoDB 是事务安全型的。
        - MyISAM 锁的粒度是表级，而 InnoDB 支持行级锁定。
        - MyISAM 支持全文类型索引，而 InnoDB 不支持全文索引。
        - MyISAM 相对简单，所以在效率上要优于 InnoDB ，小型应用可以考虑使用 MyISAM 。
        - MyISAM 表是保存成文件的形式，在跨平台的数据转移中使用 MyISAM 存储会省去不少的麻烦。
        - InnoDB 表比 MyISAM 表更安全，可以在保证数据不会丢失的情况下，切换非事务表到事务表（ alter table tablename type=innodb ）。
    - 应用场景：
        - MyISAM 管理非事务表。它提供高速存储和检索，以及全文搜索能力。如果应用中需要执行大量的 SELECT 查询，那么 MyISAM 是更好的选择。
        - InnoDB 用于事务处理应用程序，具有众多特性，包括 ACID 事务支持。如果应用中需要执行大量的 INSERT 或UPDATE 操作，则应该使用 InnoDB ，这样可以提高多用户并发操作的性能。

（Java 虚拟机）

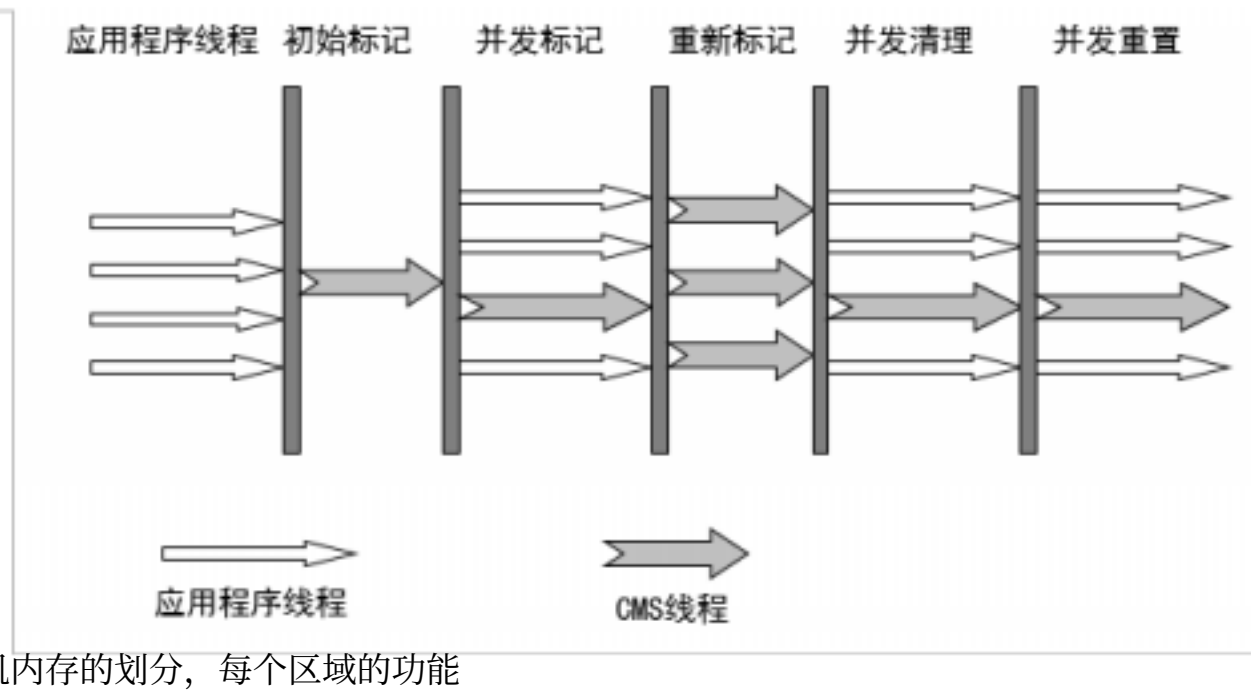
41. JVM 垃圾处理方法 (标记清除、复制、标记整理)
    - 标记 -清除算法
        - 标记阶段：先通过根节点，标记所有从根节点开始的对象，未被标记的为垃圾对象
        - 清除阶段：清除所有未被标记的对象
    - 复制算法
        - 将原有的内存空间分为两块，每次只使用其中一块，在垃圾回收时，将正在使用的内存中的存活对象复制到未使用的内存块中，然后清除正在使用的内存块中的所有对象。
    - 标记 -整理
        - 标记阶段：先通过根节点，标记所有从根节点开始的可达对象，为被标记的为垃圾对象
        - 整理阶段：将所有的存活对象压缩到内存的一段，之后清理边界所有的空间
    - 三种算法的比较
        - 效率：复制算法 > 标记 /整理算法 > 标记 /清除算法（此处的效率只是简单的对比时间复杂度，实际情况不一定如此）。
        - 内存整齐度：复制算法 =标记 /整理算法 >标记 /清除算法。
        - 内存利用率：标记 /整理算法 =标记 /清除算法 >复制算法。
42. JVM 如何 GC，新生代，老年代，持久代，都存储哪些东西，以及各个区的作用？
    - 新生代
        - 在方法中去 new 一个对象，那这方法调用完毕后，对象就会被回收，这就是一个典型的新生代对象。
    - 老年代

    - 在新生代中经历了 N次垃圾回收后仍然存活的对象就会被放到老年代中。而且大对象直接进入老年代
    - 当 Survivor 空间不够用时，需要依赖于老年代进行分配担保，所以大对象直接进入老年代
  - 永久代
    - 即方法区。

43. GC用的引用可达性分析算法中，哪些对象可作为 GC Roots 对象？
  - Java 虚拟机栈中的对象
  - 方法区中的静态成员
  - 方法区中的常量引用对象
  - 本地方法区中的 JNI（Native 方法）引用对象。
44. 什么时候进行 MinGC， FullGC
  - MinGC
    - 新生代中的垃圾收集动作，采用的是复制算法
    - 对于较大的对象，在 Minor GC 的时候可以直接进入老年代
  - FullGC
    - Full GC 是发生在老年代的垃圾收集动作，采用的是标记 -清除 /整理算法。
    - 由于老年代的对象几乎都是在 Survivor 区熬过来的，不会那么容易死掉。因此 Full GC 发生的次数不会有 Minor GC 那么频繁，并且 Time(Full GC)>Time(Minor GC)
45. 各个垃圾收集器是怎么工作的
  - Serial 收集器
    - 是一个单线程的收集器，不是只能使用一个 CPU。在进行垃圾收集时，必须暂停其他所有的工作线程，直到收集结束。
    - 新生代采用复制算法， Stop-The-World
    - 老年代采用标记 -整理算法， Stop-The-World
    - 简单高效， Client 模式下默认的新生代收集器
    - 

  - ParNew 收集器
    - ParNew 收集器是 Serial 收集器的多线程版本
    - 新生代采用复制算法， Stop-The-World
    - 老年代采用标记 -整理算法， Stop-The-World
    - 它是运行在 Server 模式下首选新生代收集器
    - 除了 Serial 收集器之外，只有它能和 CMS 收集器配合工作
    - 

  - ParNew Scanvenge 收集器
    - 类似 ParNew，但更加关注吞吐量。目标是：达到一个可控制吞吐量的收集器。
    - 停顿时间和吞吐量不可能同时调优。我们一方面希望停顿时间少，另外一方面希望吞吐量高，其实这是矛盾的。因为：在GC的时候，垃圾回收的工作总量是不变的，如果将停顿时间减少，那频率就会提高；既然频率提高了，说明就会频繁的进行 GC，那吞吐量就会减少，性能就会降低。
  - G1收集器
    - 是当今收集器发展的最前言成果之一，对垃圾回收进行了划分优先级的操作，这种有优先级的区域回收方式保证了它的高效率
    - 最大的优点是结合了空间整合，不会产生大量的碎片，也降低了进行 gc的频率
    - 让使用者明确指定指定停顿时间
  - CMS 收集器：（ Concurrent Mark Sweep ：并发标记清除老年代收集器）
    - 一种以获得最短回收停顿时间为目标的收集器，适用于互联网站或者 B/S系统的服务器上
    - 初始标记 (Stop-The-World) ：根可以直接关联到的对象
    - 并发标记 (和用户线程一起)：主要标记过程，标记全部对象
    - 重新标记 (Stop-The-World) ：由于并发标记时，用户线程依然运行，因此在正式清理前，再做修正
    - 并发清除 (和用户线程一起)：基于标记结果，直接清理对象
    - 并发收集，低停顿



46. Java 虚拟机内存的划分，每个区域的功能
    - 程序计数器 (线程私有)
        - 线程创建时创建，执行本地方法时其值为 undefined 。
    - 虚拟机栈 (线程私有)
        - （栈内存）为虚拟机执行 java 方法服务：方法被调用时创建栈帧 -->局部变量表 ->局部变量、对象引用
        - 如果线程请求的栈深度超出了虚拟机所允许的深度，就会出现 StackOverFlowError 。-Xss 规定了栈的最大空间
        - 虚拟机栈可以动态扩展，如果扩展到无法申请到足够的内存，会出现 OOM
    - 本地方法栈 (线程私有)
        - java 虚拟机栈是为虚拟机执行 java 方法服务的，而本地方法栈则为虚拟机执使用到的 Native 方法服务。
        - Java 虚拟机没有对本地方法栈的使用和数据结构做强制规定。 Sun HotSpot 把 Java 虚拟机栈和本地方法栈合二为一
        - 会抛出 StackOverFlowError 和 OutOfMemoryError
    - Java 堆
        - 被所有线程共享，在 Java 虚拟机启动时创建，几乎所有的对象实例都存放到堆中
        - GC 的管理的主要区域
        - 物理不连续，逻辑上连续，并可以动态扩展，无法扩展时抛出 OutOfMemoryError
    - 方法区
        - 用于存储已被虚拟机加载的类信息、常量、静态变量、即使编译器编译后的代码的数据
        - Sun HotSpot 虚拟机把方法区叫做永久待 (Permanent Generation)
    - 运行时常量池
        - 受到方法区的限制，抛出 OutOfMemoryError
47. 用什么工具可以查出内存泄漏
    - MemoryAnalyzer ：一个功能丰富的 JAVA 堆转储文件分析工具，可以帮助你发现内存漏洞和减少内存消耗
    - EclipseMAT ：是一款开源的 JAVA内存分析软件，查找内存泄漏，能容易找到大块内存并验证谁在一直占用它，它是基于 Eclipse RCP(Rich Client Platform) ，可以下载 RCP 的独立版本或者 Eclipse 的插件
    - JProbe ：分析 Java 的内存泄漏。
48. JVM 如何加载一个类的过程，双亲委派模型中有哪些方法有没有可能父类加载器和子类加载器，加载同一个类？如果加载同一个类，该使用哪一个类？
    - 双亲委派机制图

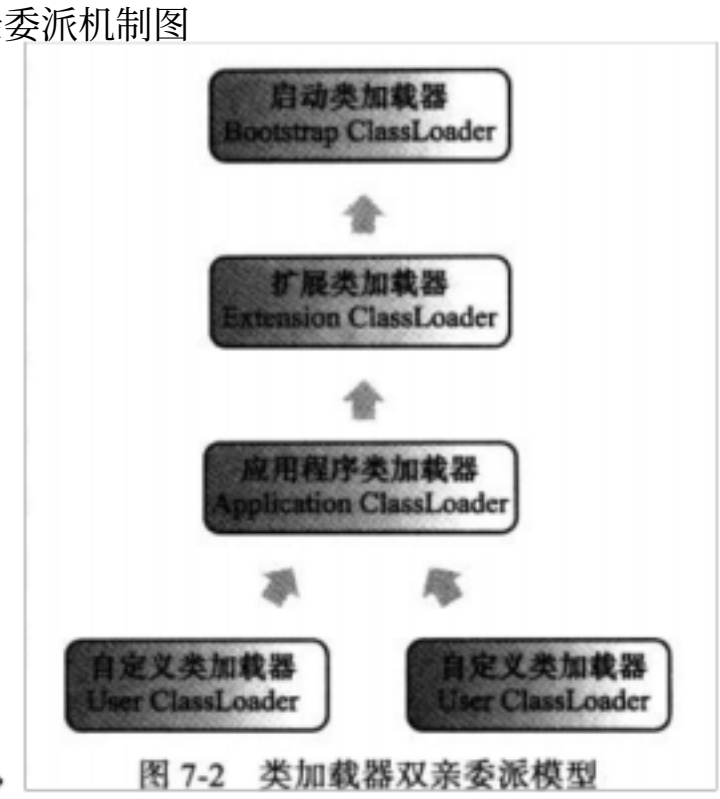


图 7-2 类加载器双亲委派模型

- 双亲委派概念
    - 如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的加载器都是如此，因此所有的类加载请求都会传给顶层的启动类加载器，只有当父加载器反馈自己无法完成该加载请求（该加载器的搜索范围中没有找到对应的类）时，子加载器才会尝试自己去加载。
- 加载器
    - 启动（ Bootstrap ）类加载器：是用本地代码实现的类装入器，它负责将 <Java_Runtime_Home>/lib 下面的类库加载到内存中（比如 rt.jar ）。由于引导类加载器涉及到虚拟机本地实现细节，开发者无法直接获取到启动类加载器的引用，所以不允许直接通过引用进行操作。
    - 标准扩展（ Extension ）类加载器：是由 Sun 的 ExtClassLoader （sun.misc.Launcher$ExtClassLoader ）实现的。它负责将 < Java_Runtime_Home >/lib/ext 或者由系统变量 java.ext.dir 指定位置中的类库加载到内存中。开发者可以直接使用标准扩展类加载器。、
    - 系统（ System ）类加载器：由 Sun 的 AppClassLoader （sun.misc.Launcher$AppClassLoader ）实现的。它负责将系统类路径（ CLASSPATH ）中指定的类库加载到内存中。开发者可以直接使用系统类加载器。除了以上列举的三种类加载器，还有一种比较特殊的类型 —线程上下文类加载器。

- 如果加载同一个类，该使用哪一个类？
  - 父类的

49. JVM 线程死锁，你该如何判断是因为什么？如果用 VisualVM ， dump 线程信息出来，会有哪些信息
    - 常常需要在隔两分钟后再次收集一次 thread dump ，如果得到的输出相同，仍然是大量 thread 都在等待给同一个地址上锁，那么肯定是死锁了。
50. java 是如何进行对象实例化的
51. Student s = new Student()? 在内存中做了哪些事情 ?
    - 加载 Student.class 文件进内存
    - 在栈内存为 s开辟空间
    - 在堆内存为学生对象开辟空间
    - 对学生对象的成员变量进行默认初始化
    - 对学生对象的成员变量进行显示初始化
    - 通过构造方法对学生对象的成员变量赋值
    - 学生对象初始化完毕，把对象地址赋值给 s变量

52. 用什么工具调试程序？ JConsole ，用过吗？
    JConsole 中，您将能够监视 JVM 内存的使用情况、线程堆栈跟踪、已装入的类和 VM 信息以及 CE MBean 。
53. 了解过 JVM 调优没，基本思路是什么

（JSP&Servlet ）

54. Servlet 的生存周期
    - Servlet 接口定义了 5个方法，其中前三个方法与 Servlet 生命周期相关：
      - -void init(ServletConfig config) throws ServletException
      - void service(ServletRequest req, ServletResponse resp) throws ServletException, java.io.IOException
      - void destory()
      - java.lang.String getServletInfo()
      - ServletConfig getServletConfig()
    - Web 容器加载 Servlet 并将其实例化后， Servlet 生命周期开始，容器运行其 init() 方法进行 Servlet 的初始化；请求到达时调用 Servlet 的 service() 方法， service() 方法会根据需要调用与请求对应的 doGet 或doPost 等方法；当服务器关闭或项目被卸载时服务器会将 Servlet 实例销毁，此时会调用 Servlet 的destroy() 方法。

55. Jsp和 Servlet 的区别
    - Servlet 是一个特殊的 Java 程序，它运行于服务器的 JVM 中，能够依靠服务器的支持向浏览器提供显示内容。 JSP 本质上是 Servlet 的一种简易形式， JSP 会被服务器处理成一个类似于 Servlet 的Java 程序，可以简化页面内容的生成。 Servlet 和JSP 最主要的不同点在于， Servlet 的应用逻辑是在 Java 文件中，并且完全从表示层中的 HTML 分离开来。而 JSP 的情况是 Java 和 HTML 可以组合成一个扩展名为 .jsp的文件。有人说， Servlet 就是在 Java 中写 HTML ，而 JSP 就是在 HTML 中写 Java 代码，当然这个说法是很片面且不够准确的。 JSP侧重于视图， Servlet 更侧重于控制逻辑，在 MVC 架构模式中， JSP 适合充当视图（view ）而 Servlet 适合充当控制器
56. 保存会话状态，有哪些方式、区别如何
    - 由于 HTTP 协议本身是无状态的，服务器为了区分不同的用户，就需要对用户会话进行跟踪，简单的说就是为用户进行登记，为用户分配唯一的 ID，下一次用户在请求中包含此 ID，服务器据此判断到底是哪一个用户。
      - ①URL 重写：在 URL 中添加用户会话的信息作为请求的参数，或者将唯一的会话 ID添加到 URL 结尾以标识一个会话。
      - ②设置表单隐藏域：将和会话跟踪相关的字段添加到隐式表单域中，这些信息不会在浏览器中显示但是提交表单时会提交给服务器。

      这两种方式很难处理跨越多个页面的信息传递，因为如果每次都要修改 URL 或在页面中添加隐式表单域来存储用户会话相关信息，事情将变得非常麻烦。
      - ③**补充： **HTML5 中可以使用 Web Storage 技术通过 JavaScript 来保存数据，例如可以使用 localStorage 和 sessionStorage 来保存用户会话的信息，也能够实现会话跟踪。

57. cookie 和 session 的区别
    - session 在服务器端， cookie 在客户端（浏览器）
    - session 的运行依赖 session id ，而 session id 是存在 cookie 中的，也就是说，如果浏览器禁用了 cookie ，同时 session 也会失效（但是可以通过其它方式实现，比如在 url 中传递 session_id ）
    - session 可以放在 文件、数据库、或内存中都可以。
    - 用户验证这种场合一般会用 session
    - cookie 不是很安全，别人可以分析存放在本地的 COOKIE 并进行 COOKIE 欺骗 考虑到安全应当使用 session 。
    - session 会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能考虑到减轻服务器性能方面，应当使用 COOKIE 。
    - 单个 cookie 保存的数据不能超过 4K，很多浏览器都限制一个站点最多保存 20 个cookie 。

（Spring&Hibernate ）

58. Spring IOC 、 AOP 的理解以及实现的原理
    - Spring IOC
      - IoC 叫控制反转，是 Inversion of Control 的缩写， DI （Dependency Injection ）叫依赖注入，是对 IoC 更简单的诠释。控制反转是把传统上由程序代码直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理。所谓的 "控制反转 "就是对组件对象控制权的转移，从程序代码本身转移到了外部容器，由容器来创建对象并管理对象之间的依赖关系。 DI 是对 IoC 更准确的描述，即组件之间的依赖关系由容器在运行期决定，形象的来说，即由容器动态的将某种依赖关系注入到组件之中。

        举个例子：一个类 A需要用到接口 B中的方法，那么就需要为类 A和接口 B建立关联或依赖关系，最原始的方法是在类 A中创建一个接口 B的实现类 C的实例，但这种方法需要开发人员自行维护二者的依赖关系，也就是说当依赖关系发生变动的时候需要修改代码并重新构建整个系统。如果通过一个容器来管理这些对象以及对象的依赖关系，则只需要在类 A中定义好用于关联接口 B的方法（构造器或 setter 方法），将类 A和接口 B的实现类 C放入容器中，通过对容器的配置来

实现二者的关联。

- Spring IOC 实现原理
  - 通过反射创建实例；
  - 获取需要注入的接口实现类并将其赋值给该接口。
- Spring AOP
  - AOP （Aspect-Oriented Programming ）指一种程序设计范型，该范型以一种称为切面（ aspect ）的语言构造为基础，切面是一种新的模块化机制，用来描述分散在对象、类或方法中的横切关注点（ crosscutting concern ）。
  - "横切关注 "是会影响到整个应用程序的关注功能，它跟正常的业务逻辑是正交的，没有必然的联系，但是几乎所有的业务逻辑都会涉及到这些关注功能。通常，事务、日志、安全性等关注就是应用中的横切关注功能。
- Spring AOP 实现原理
  - 动态代理（利用反射和动态编译将代理模式变成动态的）
  - JDK的动态代理
    - JDKProxy 返回动态代理类，是目标类所实现接口的另一个实现版本，它实现了对目标类的代理（如同UserDAOProxy 与UserDAOImp 的关系）
  - cglib 动态代理
    - CGLibProxy 返回的动态代理类，则是目标代理类的一个子类（代理类扩展了 UserDaoImpl 类）

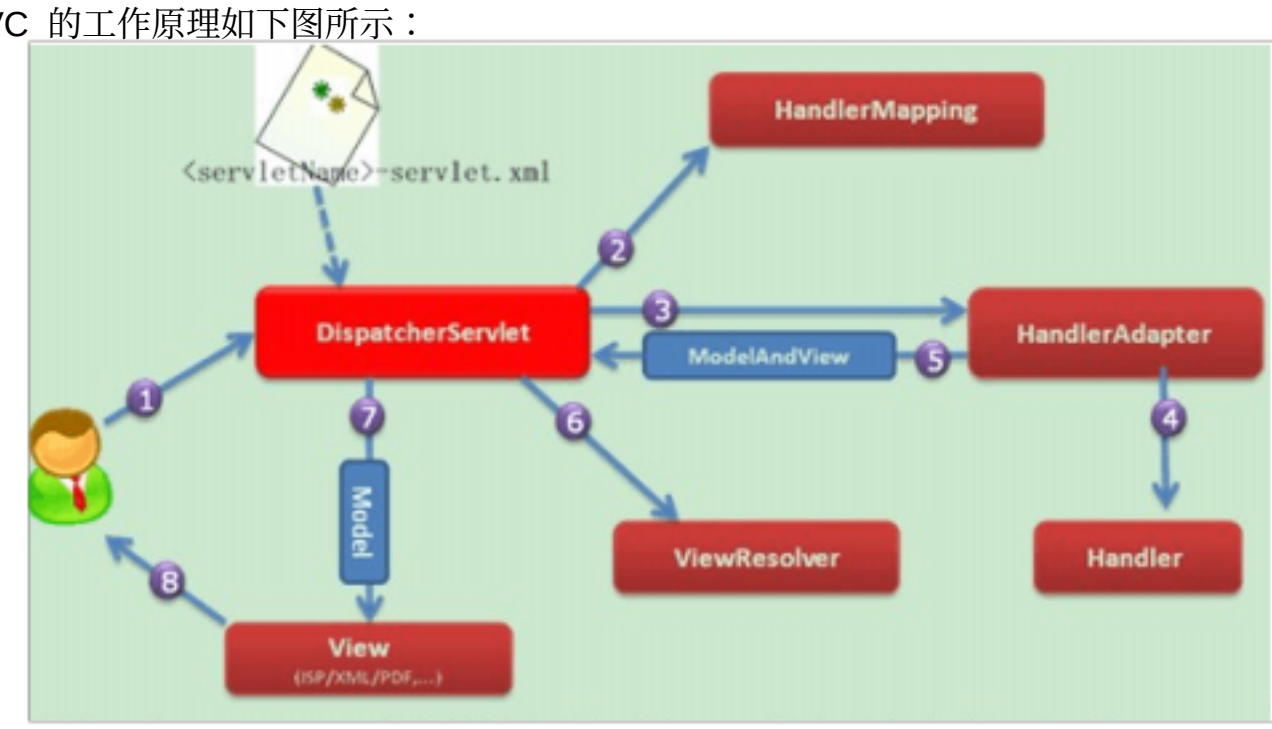
59. Ioc容器的加载过程
    - 创建 IOC 配置文件的抽象资源
    - 创建一个 BeanFactory
    - 把读取配置信息的 BeanDefinitionReader， 这里是 XmlBeanDefinitionReader 配置给 BeanFactory
    - 从定义好的资源位置读入配置信息，具体的解析过程由 XmlBeanDefinitionReader 来完成，这样完成整个载入 bean 定义的过程。

60. 动态代理与 cglib 实现的区别
    - JDK 动态代理只能对实现了接口的类生成代理，而不能针对类 .
    - CGLIB 是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法因为是继承，所以该类或方法最好不要声明成 final。
    - JDK 代理是不需要以来第三方的库，只要 JDK 环境就可以进行代理
    - CGLib 必须依赖于 CGLib 的类库，但是它需要类来实现任何接口代理的是指定的类生成一个子类，覆盖其中的方法，是一种继承

61. 代理的实现原理呗

62. HIbernate 一级缓存与二级缓存之间的区别
    - Hibernate 的 Session 提供了一级缓存的功能，默认总是有效的，当应用程序保存持久化实体、修改持久化实体时， Session 并不会立即把这种改变提交到数据库，而是缓存在当前的 Session 中，除非显示调用了 Session 的 flush() 方法或通过 close() 方法关闭 Session 。通过一级缓存，可以减少程序与数据库的交互，从而提高数据库访问性能。
    - SessionFactory 级别的二级缓存是全局性的，所有的 Session 可以共享这个二级缓存。不过二级缓存默认是关闭的，需要显示开启并指定需要使用哪种二级缓存实现类（可以使用第三方提供的实现）。一旦开启了二级缓存并设置了需要使用二级缓存的实体类， SessionFactory 就会缓存访问过的该实体类的每个对象，除非缓存的数据超出了指定的缓存空间。
    - 一级缓存和二级缓存都是对整个实体进行缓存，不会缓存普通属性，如果希望对普通属性进行缓存，可以使用查询缓存。查询缓存是将 HQL 或SQL 语句以及它们的查询结果作为键值对进行缓存，对于同样的查询可以直接从缓存中获取数据。查询缓存默认也是关闭的，需要显示开启。

63. Spring MVC 的原理
    Spring MVC 的工作原理如下图所示：



- ① 客户端的所有请求都交给前端控制器 DispatcherServlet 来处理，它会负责调用系统的其他模块来真正处理用户的请求。
- ② DispatcherServlet 收到请求后，将根据请求的信息（包括 URL、HTTP 协议方法、请求头、请求参数、 Cookie 等）以及 HandlerMapping 的配置找到处理该请求的 Handler （任何一个对象都可以作为请求的 Handler ）。
- ③在这个地方 Spring 会通过 HandlerAdapter 对该处理进行封装。
- ④ HandlerAdapter 是一个适配器，它用统一的接口对各种 Handler 中的方法进行调用。
- ⑤ Handler 完成对用户请求的处理后，会返回一个 ModelAndView 对象给 DispatcherServlet ， ModelAndView 顾名思义，包含了数据模型以及相应的视图的信息。
- ⑥ ModelAndView 的视图是逻辑视图， DispatcherServlet 还要借助 ViewResolver 完成从逻辑视图到真实视图对象的解析工作。
- ⑦ 当得到真正的视图对象后， DispatcherServlet 会利用视图对象对模型数据进行渲染。
- ⑧ 客户端得到响应，可能是一个普通的 HTML 页面，也可以是 XML 或 JSON 字符串，还可以是一张图片或者一个 PDF文件。

64. 简述Hibernate 常见优化策略。
    - 制定合理的缓存策略（二级缓存、查询缓存）。
    - 采用合理的 Session 管理机制。
    - 尽量使用延迟加载特性。
    - 设定合理的批处理参数。
    - 如果可以，选用 UUID 作为主键生成器。
    - 如果可以，选用基于版本号的乐观锁替代悲观锁。
    - 在开发过程中 ，开启 hibernate.show_sql 选项查看生成的 SQL，从而了解底层的状况；开发完成后关闭此选项。
    - 考虑数据库本身的优化，合理的索引、恰当的数据分区策略等都会对持久层的性能带来可观的提升，但这些需要专业的 DBA （数据库管理员）提供支持。

（操作系统）

65. 操作系统什么情况下会死锁？
    - 产生死锁的必要条件
        - 互斥条件。即某个资源在一段时间内只能由一个进程占有，不能同时被两个或两个以上的进程占有。这种独占资源如 CD-ROM 驱动器，打印机等等，必须在占有该资源的进程主动释放它之后，其它进程才能占有该资源。这是由资源本身的属性所决定的。如独木桥就是一种独占资源，两方的人不能同时过桥。
        - 不可抢占条件。进程所获得的资源在未使用完毕之前，资源申请者不能强行地从资源占有者手中夺取资源，而只能由该资源的占有者进程自行释放。如过独木桥的人不能强迫对方后退，也不能非法地将对方推下桥，必须是桥上的人自己过桥后空出桥面（即主动释放占有资源），对方的人才能过桥。
        - 占有且申请条件。进程至少已经占有一个资源，但又申请新的资源；由于该资源已被另外进程占有，此时该进程阻塞；但是，它在等待新资源之时，仍继续占用已占有的资源。还以过独木桥为例，甲乙两人在桥上相遇。甲走过一段桥面（即占有了一些资源），还需要走其余的桥面（申请新的资源），但那部分桥面被乙占有（乙走过一段桥面）。甲过不去，前进不能，又不后退；乙也处于同样的状况。
        - 循环等待条件。存在一个进程等待序列 {P1，P2，...，Pn}，其中 P1等待 P2所占有的某一资源， P2等待 P3所占有的某一源， .....，而 Pn等待 P1所占有的的某一资源，形成一个进程循环等待环。就像前面的过独木桥问题，甲等待乙占有的桥面，而乙又等待甲占有的桥面，从而彼此循环等待。
    - 死锁预防
        - 打破互斥条件。即允许进程同时访问某些资源。但是，有的资源是不允许被同时访问的，像打印机等等，这是由资源本身的属性所决定的。所以，这种办法并无实用价值。
        - 打破不可抢占条件。即允许进程强行从占有者那里夺取某些资源。就是说，当一个进程已占有了某些资源，它又申请新的资源，但不能立即被满足时，它必须释放所占有的全部资源，以后再重新申请。它所释放的资源可以分配给其它进程。这就相当于该进程占有的资源被隐蔽地强占了。这种预防死锁的方法实现起来困难，会降低系统性能。
        - 打破占有且申请条件。可以实行资源预先分配策略。即进程在运行前一次性地向系统申请它所需要的全部资源。如果某个进程所需的全部资源得不到满足，则不分配任何资源，此进程暂不运行。只有当系统能够满足当前进程的全部资源需求时，才一次性地将所申请的资源全部分配给该进程。由于运行的进程已占有了它所需的全部资源，所以不会发生占有资源又申请资源的现象，因此不会发生死锁。但是，这种策略也有如下缺点：
            - 在许多情况下，一个进程在执行之前不可能知道它所需要的全部资源。这是由于进程在执行时是动态的，不可预测的；
            - 资源利用率低。无论所分资源何时用到，一个进程只有在占有所需的全部资源后才能执行。即使有些资源最后才被该进程用到一次，但该进程在生存期间却一直占有它们，造成长期占着不用的状况。这显然是一种极大的资源浪费；
            - 降低了进程的并发性。因为资源有限，又加上存在浪费，能分配到所需全部资源的进程个数就必然少了。
        - 打破循环等待条件，实行资源有序分配策略。采用这种策略，即把资源事先分类编号，按号分配，使进程在申请，占用资源时不会形成环路。所有进程对资源的请求必须严格按资源序号递增的顺序提出。进程占用了小号资源，才能申请大号资源，就不会产生环路，从而预防了死锁。这种策略与前面的策略相比，资源的利用率和系统吞吐量都有很大提高，但是也存在以下缺点：
            - 限制了进程对资源的请求，同时给系统中所有资源合理编号也是件困难事，并增加了系统开销；
            - 为了遵循按编号申请的次序，暂不使用的资源也需要提前申请，从而增加了进程对资源的占用时间。
        - 死锁避免
            - 安全序列
            - 银行家算法

66. 如何理解分布式锁？
    - 分布式锁，是控制分布式系统之间同步访问共享资源的一种方式。在分布式系统中，常常需要协调他们的动作。如果不同的系统或是同一个系统的不同主机之间共享了一个或一组资源，那么访问这些资源的时候，往往需要互斥来防止彼此干扰来保证一致性，在这种情况下，便需要使用到分布式锁。

67. 进程间通信有哪几种方式？
    - 管道 (PIPE) ：管道可用于具有亲缘关系进程间的通信，允许一个进程和另一个与它有共同祖先的进程之间进行通信。
    - 命名管道 (FIFO) ：命名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。命名管道在文件系统中有对应的文件名。命名管道通过命令 mkfifo 或系统调用 mkfifo 来创建。
    - 信号 (Signal) ：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身。
    - 消息队列 (MessageQueue) ：消息队列是消息的链接表，包括 Posix 消息队列 system V 消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺限。
    - 共享内存 (SharedMemory) ：使得多个进程可以访问同一块内存空间，是最快的可用 IPC 形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。
    - 内存映射（ mapped memory ）：内存映射允许任何多个进程间通信，每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现它。
    - 信号量（ semaphore ）：主要作为进程间以及同一进程不同线程之间的同步手段。
    - 套接口（ Socket ）：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由 Unix 系统的 BSD 分支开发出来的，但现在一般可以移植到其它类 Unix 系统上： Linux 和System V 的变种都支持套接字。

68. 线程同步与阻塞的关系？同步一定阻塞吗？阻塞一定同步吗？
    - 线程同步与阻塞的关系
        - 线程同步与阻塞没有一点关系
        - 同步和异步关注的是消息通信机制（ synchronous communication/ asynchronous communication ）。所谓同步，就是在发出一个 *调用 *时，在没有得到结果之前，该 *调用 *就不返回。但是一旦调用返回，就得到返回值了。换句话说，就是由 *调用者 *主动等待这个 *调用 *的结果。而异步则是相反， *调用 *在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在 *调用 *发出后， *被调用者 *通过状态、通知来通知调用者 ，或通过回调函数处理这个调用。
        - 阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态。阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程

69. 操作系统如何进行分页调度？

（Linux）

70. Linux 下如何进行进程调度的？
71. Linux 下你常用的命令有哪些？

（其他）

72. 常用的 hash 算法有哪些？
    - 加法 Hash ；把输入元素一个一个的加起来构成最后的结果
    - 位运算 Hash ；这类型 Hash 函数通过利用各种位运算（常见的是移位和异或）来充分的混合输入元素
    - 乘法 Hash ；这种类型的 Hash 函数利用了乘法的不相关性（乘法的这种性质，最有名的莫过于平方取头尾的随机数生成算法，虽然这种算法效果并不好）； jdk5.0 里面的 String 类的 hashCode() 方法也使用乘法 Hash ；32 位 FNV 算法
    - 除法 Hash ；除法和乘法一样，同样具有表面上看起来的不相关性。不过，因为除法太慢，这种方式几乎找不到真正的应用
    - 查表 Hash ；查表 Hash 最有名的例子莫过于 CRC 系列算法。虽然 CRC 系列算法本身并不是查表，但是，查表是它的一种最快的实现方式。查表 Hash 中有名的例子有： Universal Hashing 和 Zobrist Hashing 。他们的表格都是随机生成的。
    - 混合 Hash ；混合 Hash 算法利用了以上各种方式。各种常见的 Hash 算法，比如 MD5 、Tiger 都属于这个范围。它们一般很少在面向查找的 Hash 函数里面使用

73. 如何设计存储海量数据的存储系统
74. 缓存的实现原理，设计缓存要注意什么
75. 什么是一致性哈希？用来解决什么问题？
    在设计分布式 cache 系统的时候，我们需要让 key 的分布均衡，并且在增加 cache server 后， cache 的迁移做到最少。
76. 现在有一个进程挂起了，如何用工具查出原因？
    - 通过 Javacore 了解线程运行状况
      javacore ，也可以称为 “ threaddump 或是“ javadump，”它是 Java 提供的一种诊断特性，能够提供一份可读的当前运行的 JVM 中线程使用情况的快照。即在某个特定时刻， JVM 中有哪些线程在运行，每个线程执行到哪一个类，哪一个方法。
      应用程序如果出现不可恢复的错误或是内存泄露，就会自动触发 Javacore 的生成。而为了性能问题诊断的需要，我们也会主动触发生成 Javacore 。在 AIX 、Linux、 Solaris 环境中，我们通常使用 kill -3 <PID> 产生该进程的 Javacore 。IBM Java6 中产生 Javacore 的详细方法可以参考文章 [1]。

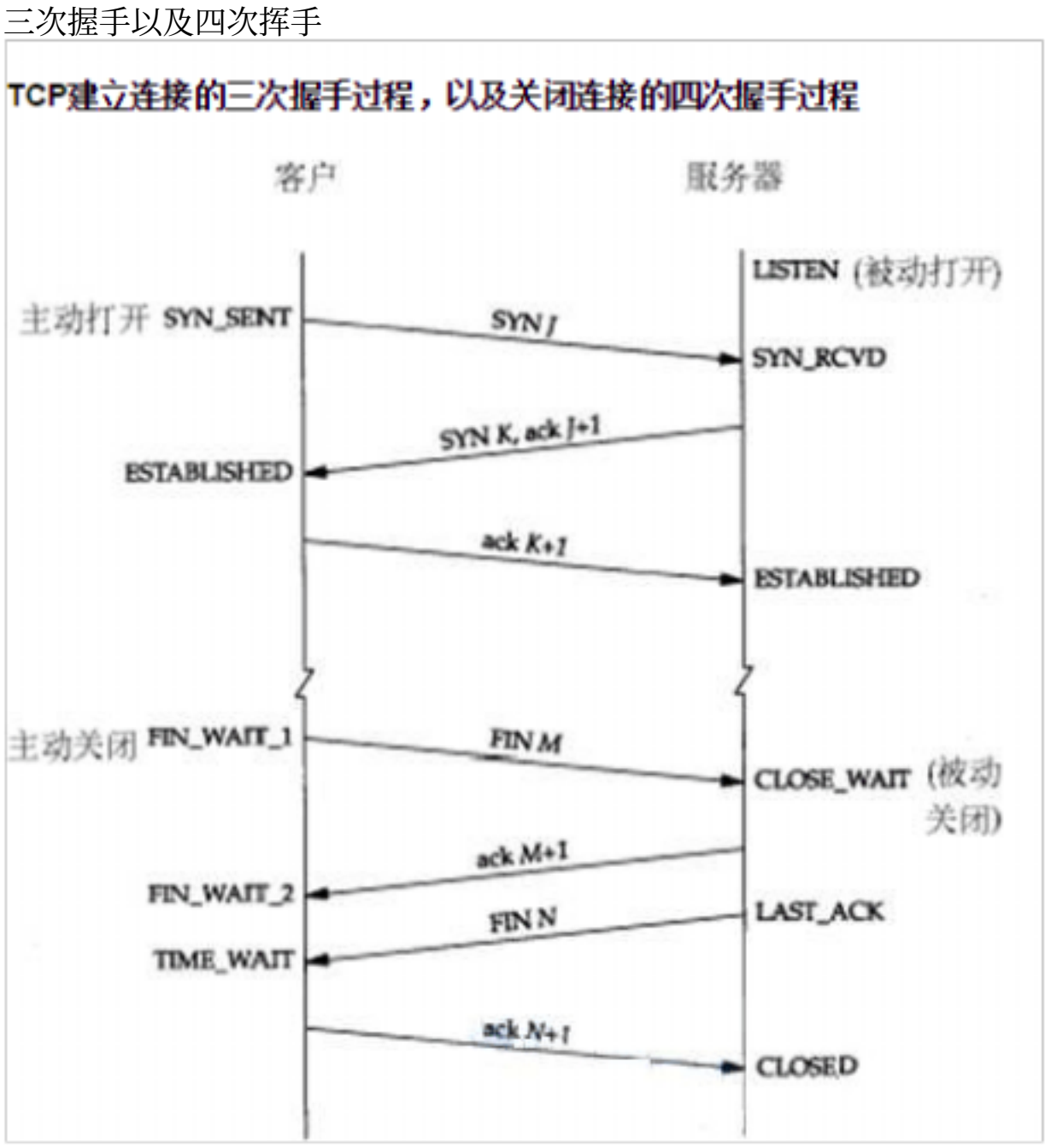
77. 你知道的开源协议有哪些？
    - Mozilla Public License ： MPL License ，允许免费重发布、免费修改，但要求修改后的代码版权归软件的发起者。这种授权维护了商业软件的利益，它要求基于这种软件得修改无偿贡献版权给该软件。这样，围绕该软件得所有代码得版权都集中在发起开发人得手中。但 MPL 是允许修改，无偿使用得。 MPL 软件对链接没有要求。
    - BSD 开源协议：给于使用者很大自由的协议。可以自由的使用，修改源代码，也可以将修改后的代码作为开源或者专有软件再发布。
    - Apache Licence 2.0 ：Apache Licence 是著名的非盈利开源组织 Apache 采用的协议。该协议和 BSD 类似，同样鼓励代码共享和尊重原作者的著作权，同样允许代码修改，再发布 (作为开源或商业软件)。
    - GPL ：GPL 许可证是自由软件的应用最广泛的软件许可证，人们可以修改程式的一个或几个副本或程式的任何部分，以此形成基於这些程式的衍生作品。必须在修改过的档案中附有明显的说明：您修改了此一档案及任何修改的日期。 您必须让您发布或出版的作品，包括本程式的全部或一部分，或内含本程式的全部或部分所衍生的作品，允许第三方在此许可证条款下使用，并且不得因为此项授权行为而收费。
    - LGPL ：LGPL 是GPL 的一个为主要为类库使用设计的开源协议。和 GPL 要求任何使用 /修改 /衍生之 GPL 类库的的软件必须采用 GPL 协议不同。 LGPL 允许商业软件通过类库引用 (link) 方式使用 LGPL 类库而不需要开源商业软件的代码。这使得采用 LGPL 协议的开源代码可以被商业软件作为类库引用并发布和销售。
    - Public Domain ：公共域授权。将软件授权为公共域，这些软件包没有授权协议，任何人都可以随意使用它
78. 你知道的开源软件有哪些？
    - JDK
    - Eclipse
    - Tomcat
    - Spring
    - Hibernate
    - MySQL

（计算机网络）

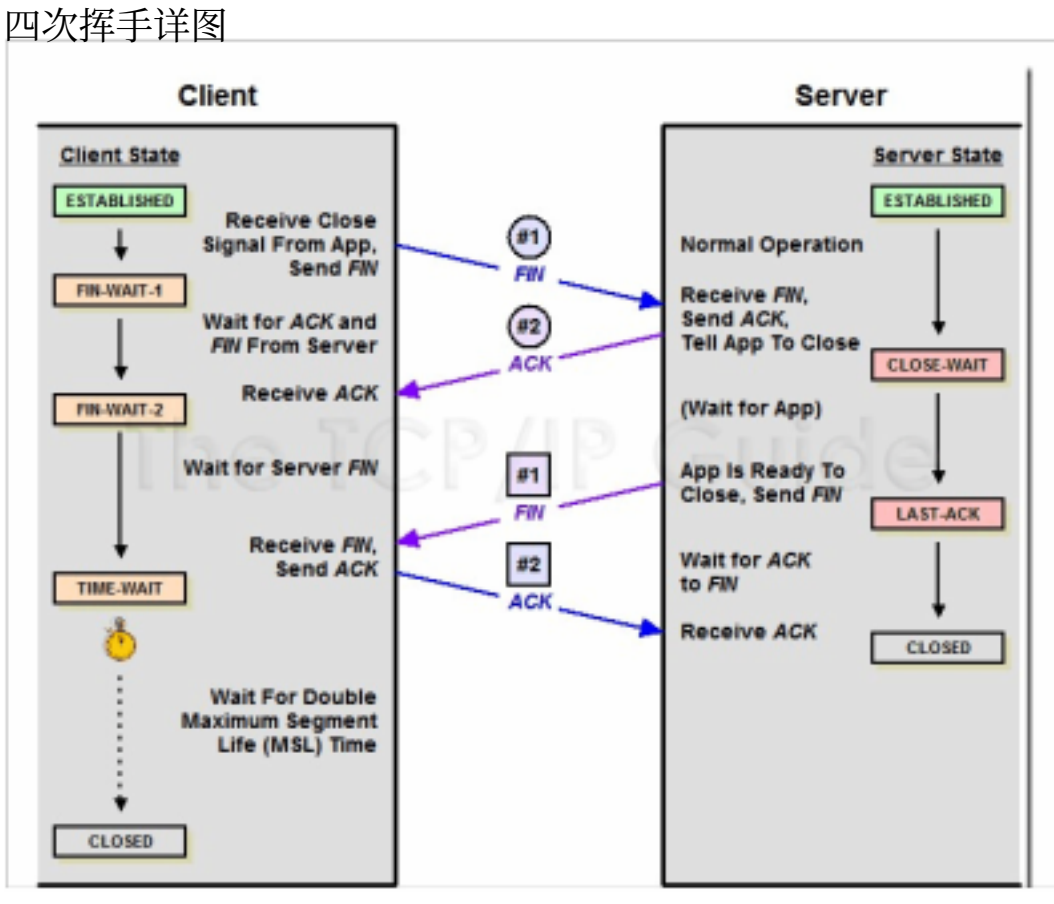
79. Http 和 https 的区别

- http 是HTTP 协议运行在 TCP 之上。所有传输的内容都是明文，客户端和服务器端都无法验证对方的身份。
- https 是 HTTP 运行在 SSL/TLS 之上， SSL/TLS 运行在 TCP 之上。所有传输的内容都经过加密，加密采用对称加密，但对称加密的密钥用服务器方的证书进行了非对称加密。此外客户端可以验证服务器端的身份，如果配置了客户端验证，服务器方也可以验证客户端的身份。
- https 协议需要到 ca申请证书，一般免费证书很少，需要交费。
- http 是超文本传输协议，信息是明文传输， https 则是具有安全性的 ssl加密传输协议
- http 和https 使用的是完全不同的连接方式用的端口也不一样 ,前者是 80, 后者是 443 。
- http 的连接很简单 ,是无状态的
- HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议 要比 http 协议安全

80. TCP 如何保证可靠传输？三次握手过程？
    - TCP 如何保证可靠传输
        - 数据包校验
        - 超时重传机制
        - 应答机制
        - 对失序数据包重排序
        - TCP 还能提供流量控制
    - 三次握手以及四次挥手



    - 四次挥手详图



81. 为什么 TCP 连接需要三次握手，两次不可以吗，为什么
    - 为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误。
    - 例子
        - 已失效的连接请求报文段 的产生在这样一种情况下： client 发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达 server 。本来这是一个早已失效的报文段。但 server 收到此失效的连接请求报文段后，就误认为是 client 再次发出的一个新的连接请求。于

是就向 client 发出确认报文段，同意建立连接。假设不采用 “三次握手”，那么只要 server 发出确认，新的连接就建立了。由于现在 client 并没有发出建立连接的请求，因此不会理睬 server 的确认，也不会向 server 发送数据。但 server 却以为新的运输连接已经建立，并一直等待 client 发来数据。这样，server 的很多资源就白白浪费掉了。采用 “三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client 不会向 server 的确认发出确认。server 由于收不到确认，就知道 client 并没有要求建立连接。”

82. 如果客户端不断的发送请求连接会怎样？
    - 服务器端回为每个请求创建一个链接，然后向 client 端发送创建链接时的回复，然后进行等待客户端发送第三次握手数据包，这样会白白浪费资源
    - DDos 攻击
      简单的说就是想服务器发送链接请求，首先进行
      第一步：客户端向服务器端发送连接请求数据包（1）
      第二步：服务器向客户端回复连接请求数据包（2），然后服务器等待客户端发送 tcp/ip 链接的第三步数据包 (3)
      第三步：如果客户端不向服务器端发送最后一个数据包 (3)，则服务器须等待 30s 到2min 中才能将此链接进行关闭。当大量的请求只进行到第二步，而不进行第三步，服务器又大量的资源等待第三个数据包。则造成 DDos 攻击。
    - DDos 预防 (没有根治的办法，除非不用 TCP/IP 链接)、
        - 确保服务器的系统文件是最新版本，并及时更新系统补丁
        - 关闭不必要的服务
        - 限制同时打开 SYN 的半连接数目
        - 缩短 SYN 半连接的 time out 时间
        - 正确设置防火墙
        - 禁止对主机的非开放服务的访问
        - 限制特定 IP短地址的访问
        - 启用防火墙的防 DDos 的属性
        - 严格限制对外开放的服务器的向外访问
        - 运行端口映射程序祸端口扫描程序，要认真检查特权端口和非特权端口。
        - 认真检查网络设备和主机 /服务器系统的日志。只要日志出现漏洞或是时间变更，那这台机器就可能遭到了攻击。
        - 限制在防火墙外与网络文件共享。这样会给黑客截取系统文件的机会，主机的信息暴露给黑客，无疑是给了对方入侵的机会。

83. 问：那怎么知道连接是恶意的呢？可能是正常连接？
84. GET 和POST 的区别？
    - GET 被强制服务器支持
    - 浏览器对 URL 的长度有限制，所以 GET 请求不能代替 POST 请求发送大量数据
    - GET 请求发送数据更小
    - GET 请求是不安全的
    - GET 请求是幂等的
    - POST 请求不能被缓存
    - POST 请求相对 GET 请求是「安全」的

    - 在以下情况中，请使用 POST 请求：
      1. 无法使用缓存文件（更新服务器上的文件或数据库）
      2. 向服务器发送大量数据（ POST 没有数据量限制）
      3. 发送包含未知字符的用户输入时，POST 比 GET 更稳定也更可靠
      4.post 比Get 安全性更高

85. TCP 和UDP 区别？如何改进 TCP
    - TCP 和UDP 区别
        - UDP 是无连接的，即发送数据之前不需要建立连接。
        - UDP 使用尽最大努力交付，即不保证可靠交付，同时也不使用拥塞控制。
        - UDP 是面向报文的。 UDP 没有拥塞控制，很适合多媒体通信的要求。
        - UDP 支持一对一、一对多、多对一和多对多的交互通信。
        - UDP 的首部开销小，只有 8 个字节。
        - TCP 是面向连接的运输层协议。
        - 每一条 TCP 连接只能有两个端点 (endpoint) ，每一条 TCP 连接只能是点对点的（一对一）。
        - TCP 提供可靠交付的服务。
        - TCP 提供全双工通信。
        - TCP 是面向字节流。
        - 首部最低 20个字节。
    - TCP 加快传输效率的方法
        - 采取一块确认的机制
86. 滑动窗口算法？
    http://coolshell.cn/articles/11609.html
87. TCP 的拥塞处理 – Congestion Handling
    1）慢启动， 2）拥塞避免， 3）拥塞发生， 4）快速恢复。
88. 从输入网址到获得页面的过程
    - 查询 DNS，获取域名对应的 IP 地址
        - 浏览器搜索自身的 DNS 缓存

    - 搜索操作系统的 DNS 缓存
    - 读取本地的 HOST 文件
    - 发起一个 DNS 的系统调用
        - 宽带运营服务器查看本身缓存
        - 运营商服务器发起一个迭代 DNS 解析请求
- 浏览器获得域名对应的 IP 地址后，发起 HTTP 三次握手
- TCP/IP 连接建立起来后，浏览器就可以向服务器发送 HTTP 请求了
- 服务器接受到这个请求，根据路径参数，经过后端的一些处理生成 HTML 页面代码返回给浏览器
- 浏览器拿到完整的 HTML 页面代码开始解析和渲染，如果遇到引用的外部 JS， CSS, 图片等静态资源，它们同样也是一个个的 HTTP 请求，都需要经过上面的步骤
- 浏览器根据拿到的资源对页面进行渲染，最终把一个完整的页面呈现给用户

（算法）

89. 如何判断一个单链表是否有环？
90. 快速排序，过程，复杂度？什么情况下适用，什么情况下不适用？
91. 什么是二叉平衡树，如何插入节点，删除节点
92. 二分搜索的过程
93. 归并排序的过程？时间复杂度？空间复杂度？
94. 给你一万个数，如何找出里面所有重复的数？用所有你能想到的方法，时间复杂度和空间复杂度分别是多少
95. 给你一个数组，如何里面找到和为 K的两个数
96. 100000 个数找出最小或最大的 10个？
97. 一堆数字里面继续去重，要怎么处理？阅读 RFC2616 文档，即 HTTP/1.1 规范，输入某个网址，利用 Java 的 Socket 发送 HTTP 请求，特别要求能够解码 chunked 编码，观察文档中的伪代码实现，自己用 Java 代码实现，将解析后的整个html 文档输出到控制台上，不要求关注太多细节。 (就是不允许用 httpclient 的jar包，自行实现这个 jar包类似的功能)

（智力题）

98. 给你 50个红球和 50 个黑球，有两个一模一样的桶，往桶里放球，让朋友去随机抽，采用什么策略可以让朋友抽到红球的概率更高？
99. 称重的方法（从 100 个硬币中找出最轻的那个假币）
100. 两个鸡蛋
101. 一筐鸡蛋 取 剩2 1， 3 0， 4 1， 5 4， 6 3，

102. 在项目中遇到的最难的问题是什么？你是怎么解决的
103. 你认为自己有那些方面不足
104. 平常如何学习的