V2版本说明

为了方便大家阅读，我这里做一个说明，就是 V2 版和 V1 版有什么区别，以后每个版本都会与前一个版本进行对比说明。

此次 V2 的更新，有如下两点

1. 新增了 **62 道 Redis** 面试题，V1 版本只有 12 道，现在改成 62 道，那么关于 Redis 这块就基本全了。
2. 操作系统从 **20** 道 新增到 **32** 道，后续还会继续更新。

另外，也建议大家来网站读，因为如果有错误，我会及时在网站更改，PDF 很难及时更新。

在线网站：https://www.iamshuaidi.com

前言（必读）

对于Java的学习，很多人可能学了之后，不知道自己处于哪个阶段，也不到究竟要学到哪个程度，帅地觉得，验证自己学得如何最好的面试，就是尝试去面试，而面试无非就是问你一些面试题，所以呢，帅地整理了这些 Java 面试题，从 Java 基础，集合，并发到虚拟机，并且附带了详细的答案，无论是想面试还是想看看自己学得如何，那么这份面试题，都值得你去学习。

当然，如果单单只会 Java，是很难进大公司的，所以计算机基础之类的也得学，为此，帅地整理了这份近 20 万字的面试题，而且还在持续完善中。

你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**Java面试题**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**Java面试题**」，即可获取最新版 PDF。

另外，也建议大家来网站读，因为如果有错误，我会及时在网站更改，PDF 很难及时更新。

在线网站：https://www.iamshuaidi.com

Java基础

1、解释下什么是面向对象？面向对象和面向过程的区别?

面向对象是一种基于面向过程的编程思想，是向现实世界模型的自然延伸，这是一种“万物皆对象”的编程思想。由执行者变为指挥者，在现实生活中的任何物体都可以归为一类事物，而每一个个体都是一类事物的实例。面向对象的编程是以对象为中心，以消息为驱动。

区别:

（1）编程思路不同：面向过程以实现功能的函数开发为主，而面向对象要首先抽象出类、属性及其方法，然后通过实例化类、执行方法来完成功能。

（2）封装性：都具有封装性，但是面向过程是封装的是功能，而面向对象封装的是数据和功能。

（3）面向对象具有继承性和多态性，而面向过程没有继承性和多态性，所以面向对象优势很明显

2、面向对象的三大特性？分别解释下?

（1）封装：通常认为封装是把数据和操作数据的方法封装起来，对数据的访问只能通过已定义的接口。 （2）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），得到继承信息的被称为子类（派生类）。 （3）多态：分为编译时多态（方法重载）和运行时多态（方法重写）。要实现多态需要做两件事：一是子类继承父类并重写父类中的方法，二是用父类型引用子类型对象，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。

几点补充

1）子类拥有父类对象所有的属性和方法（包括私有属性和私有方法），但是父类中的私有属性和方法子类是无法访问，只是拥有。因为在一个子类被创建的时候，首先会在内存中创建一个父类对象，然后在父类对象外部放上子类独有的属性，两者合起来形成一个子类的对象； （2）子类可以拥有自己属性和方法； （3）子类可以用自己的方式实现父类的方法。（重写）

3、JDK、JRE、JVM 三者之间的关系?

JDK（Java Development Kit）： 是 Java 开发工具包，是整个 Java 的核心，包括了 Java 运行环境 JRE、Java 工具和 Java 基础类库。

JRE（ Java Runtime Environment）： 是 Java 的运行环境，包含 JVM 标准实现及 Java 核心类库。

JVM（Java Virtual Machine）： 是 Java 虚拟机，是整个 Java 实现跨平台的最核心的部分，能够运行以 Java 语言写作的软件程序。所有的 Java 程序会首先被编译为 .class 的类文件，这种类文件可以在虚拟机上执行。

4、重载和重写的区别?

（1）重载：编译时多态、同一个类中同名的方法具有不同的参数列表、不能根据返回类型进行区分【因为：函数调用时不能指定类型信息，编译器不知道你要调哪个函数】；

（2）重写（又名覆盖）：运行时多态、子类与父类之间、子类重写父类的方法具有相同的返回类型、更好的访问权限。

5、Java 中是否可以重写一个 private 或者 static 方法?

Java 中 static 方法不能被覆盖，因为方法覆盖是基于运行时动态绑定的，而 static 方法是编译时静态绑定的。static 方法跟类的任何实例都不相关，所以概念上不适用。

Java 中也不可以覆盖 private 的方法，因为 private 修饰的变量和方法只能在当前类中使用， 如果是其他的类继承当前类是不能访问到 private 变量或方法的，当然也不能覆盖。

> 静态方法补充：静态的方法可以被继承，但是不能重写。如果父类和子类中存在同样名称和参数的静态方法，那么该子类的方法会把原来继承过来的父类的方法隐藏，而不是重写。通俗的讲就是父类的方法和子类的方法是两个没有关系的方法，具体调用哪一个方法是看是哪个对象的引用；这种父子类方法也不在存在多态的性质。

6、构造器是否可以被重写?

在讲继承的时候我们就知道父类的私有属性和构造方法并不能被继承，所以 Constructor 也就不能被 Override（重写），但是可以 Overload（重载），所以你可以看到一个类中有多个构造函数的情况。

7、构造方法有哪些特性?

（1）名字与类名相同；

（2）没有返回值，但不能用 void 声明构造函数；

（3）成类的对象时自动执行，无需调用。

8、在 Java 中定义一个不做事且没有参数的构造方法有什么作用?

Java 程序在执行子类的构造方法之前，如果没有用 super() 来调用父类特定的构造方法，则会调用父类中“没有参数的构造方法”。

因此，如果父类中只定义了有参数的构造方法，而在子类的构造方法中又没有用 super() 来调用父类中特定的构造方法，则编译时将发生错误，因为 Java 程序在父类中找不到没有参数的构造方法可供执行。解决办法是：在父类里加上一个不做事且没有参数的构造方法。

9、Java 中创建对象的几种方式?

1、使用 new 关键字；

2、使用 Class 类的 newInstance 方法，该方法调用无参的构造器创建对象（反射）：Class.forName.newInstance()；

3、使用 clone() 方法；

4、反序列化，比如调用 ObjectInputStream 类的 readObject() 方法。

10、抽象类和接口有什么区别?

1）抽象类中可以定义构造函数，接口不能定义构造函数；

（2）抽象类中可以有抽象方法和具体方法，而接口中只能有抽象方法（public abstract）；

（3）抽象类中的成员权限可以是 public、默认、protected（抽象类中抽象方法就是为了重写，所以不能被 private 修饰），而接口中的成员只可以是 public（方法默认：public abstrat、成员变量默认：public static final）；

（4）抽象类中可以包含静态方法，而接口中不可以包含静态方法；

JDK 8 中的改变：

1、在 JDK1.8中，允许在接口中包含带有具体实现的方法，使用 default 修饰，这类方法就是默认方法。

2、抽象类中可以包含静态方法，在 JDK1.8 之前接口中不能包含静态方法，JDK1.8 以后可以包含。之前不能包含是因为，接口不可以实现方法，只可以定义方法，所以不能使用静态方法（因为静态方法必须实现）。现在可以包含了，只能直接用接口调用静态方法。JDK1.8 仍然不可以包含静态代码块。

11、静态变量和实例变量的区别?

静态变量：是被 static 修饰的变量，也称为类变量，它属于类，因此不管创建多少个对象，静态变量在内存中有且仅有一个拷贝；静态变量可以实现让多个对象共享内存。

实例变量：属于某一实例，需要先创建对象，然后通过对象才能访问到它。

12、short s1 = 1；s1 = s1 + 1；有什么错？那么 short s1 = 1; s1 += 1；呢？有没有错误?

对于 short s1 = 1; s1 = s1 + 1; 来说，在 s1 + 1 运算时会自动提升表达式的类型为 int ，那么将 int 型值赋值给 short 型变量，s1 会出现类型转换错误。

对于 short s1 = 1; s1 += 1; 来说，+= 是 Java 语言规定的运算符，Java 编译器会对它进行特殊处理，因此可以正确编译。

13、Integer 和 int 的区别?

（1）int 是 Java 的八种基本数据类型之一，而 Integer 是 Java 为 int 类型提供的封装类；

（2）int 型变量的默认值是 0，Integer 变量的默认值是 null，这一点说明 Integer 可以区分出未赋值和值为 0 的区分；

（3）Integer 变量必须实例化后才可以使用，而 int 不需要。

Integer 和 int 的比较延伸：

1、由于 Integer 变量实际上是对一个 Integer 对象的引用，所以两个通过 new 生成的 Integer 变量永远是不相等的，因为其内存地址是不同的；

2、Integer 变量和 int 变量比较时，只要两个变量的值是相等的，则结果为 true。因为包装类 Integer 和基本数据类型 int 类型进行比较时，Java 会自动拆包装类为 int，然后进行比较，实际上就是两个 int 型变量在进行比较；

3、非 new 生成的 Integer 变量和 new Integer() 生成的变量进行比较时，结果为 false。因为非 new 生成的 Integer 变量指向的是 Java 常量池中的对象，而 new Integer() 生成的变量指向堆中新建的对象，两者在内存中的地址不同；

4、对于两个非 new 生成的 Integer 对象进行比较时，如果两个变量的值在区间 [-128, 127] 之间，则比较结果为 true，否则为 false。Java 在编译 Integer i = 100 时，会编译成 Integer i = Integer.valueOf(100)，而 Integer 类型的 valueOf 的源码如下所示：

```
public static Integer valueOf(int var0) {
  return var0 >= -128 && var0 <= Integer.IntegerCache.high ?
Integer.IntegerCache.cache[var0 + 128] : new Integer(var0);
}
```

从上面的代码中可以看出：Java 对于 [-128, 127] 之间的数会进行缓存，比如：Integer i = 127，会将 127 进行缓存，下次再写 Integer j = 127 的时候，就会直接从缓存中取出，而对于这个区间之外的数就需要 new 了。

- **包装类的缓存：**

Boolean：全部缓存

Byte：全部缓存

Character：<= 127 缓存

Short：-128 — 127 缓存

Long：-128 — 127 缓存

Integer：-128 — 127 缓存

Float：没有缓存

Doulbe：没有缓存

14、装箱和拆箱

自动装箱是 Java 编译器在基本数据类型和对应得包装类之间做的一个转化。比如：把 int 转化成 Integer，double 转化成 Double 等等。反之就是自动拆箱。

原始类型：boolean、char、byte、short、int、long、float、double

封装类型：Boolean、Character、Byte、Short、Integer、Long、Float、Double

15、switch 语句能否作用在 byte 上，能否作用在 long 上，能否作用在 String 上?

在 switch(expr 1) 中，expr1 只能是一个整数表达式或者枚举常量。而整数表达式可以是 int 基本数据类型或者 Integer 包装类型。由于，byte、short、char 都可以隐式转换为 int，所以，这些类型以及这些类型的包装类型也都是可以的。而 long 和 String 类型都不符合 switch 的语法规定，并且不能被隐式的转换为 int 类型，所以，它们不能作用于 switch 语句中。不过，需要注意的是在 JDK1.7 版本之后 switch 就可以作用在 String 上了。

16、Object 的常用方法有哪些?

clone 方法：用于创建并返回当前对象的一份拷贝；

getClass 方法：用于返回当前运行时对象的 Class；

toString 方法：返回对象的字符串表示形式；

finalize 方法：实例被垃圾回收器回收时触发的方法；

equals 方法：用于比较两个对象的内存地址是否相等，一般需要重写；

hashCode 方法：用于返回对象的哈希值；

notify 方法：唤醒一个在此对象监视器上等待的线程。如果有多个线程在等待只会唤醒一个。

notifyAll 方法：作用跟 notify() 一样，只不过会唤醒在此对象监视器上等待的所有线程，而不是一个线程。

wait 方法：让当前对象等待；

.....

16、final、finally、finalize 的区别?

final：用于声明属性、方法和类，分别表示属性不可变、方法不可覆盖、被其修饰的类不可继承；

finally：异常处理语句结构的一部分，表示总是执行；

finallize：Object类的一个方法，在垃圾回收时会调用被回收对象的finalize

17、== 和 equals 的区别?

==：如果比较的对象是基本数据类型，则比较的是数值是否相等；如果比较的是引用数据类型，则比较的是对象的地址值是否相等。

equals 方法：用来比较两个对象的内容是否相等。注意：equals 方法不能用于比较基本数据类型的变量。如果没有对 equals 方法进行重写，则比较的是引用类型的变量所指向的对象的地址（很多类重新了 equals 方法，比如 String、Integer 等把它变成了值比较，所以一般情况下 equals 比较的是值是否相等）。

18、两个对象的 hashCode() 相同，则 equals() 也一定为 true 吗?

两个对象的 hashCode() 相同，equals() 不一定为 true。因为在散列表中，hashCode() 相等即两个键值对的哈希值相等，然而哈希值相等，并不一定能得出键值对相等【散列冲突】。

19、为什么重写 equals() 就一定要重写 hashCode() 方法?

这个问题应该是有个前提，就是你需要用到 HashMap、HashSet 等 Java 集合，用不到哈希表的话，其实仅仅重写 equals() 方法也可以。而工作中的场景是常常用到 Java 集合，所以 Java 官方建议重写 equals() 就一定要重写 hashCode() 方法。

对于对象集合的判重，如果一个集合含有 10000 个对象实例，仅仅使用 equals() 方法的话，那么对于一个对象判重就需要比较 10000 次，随着集合规模的增大，时间开销是很大的。但是同时使用哈希表的话，就能快速定位到对象的大概存储位置，并且在定位到大概存储位置后，后续比较过程中，如果两个对象的 hashCode 不相同，也不再需要调用 equals() 方法，从而大大减少了 equals() 比较次数。

所以从程序实现原理上来讲的话，既需要 equals() 方法，也需要 hashCode() 方法。那么既然重写了 equals()，那么也要重写 hashCode() 方法，以保证两者之间的配合关系。

hashCode（）与equals（）的相关规定：

1、如果两个对象相等，则 hashCode 一定也是相同的；

2、两个对象相等，对两个对象分别调用 equals 方法都返回 true；

3、两个对象有相同的 hashCode 值，它们也不一定是相等的；

4、因此，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖；

5、hashCode() 的默认行为是对堆上的对象产生独特值。如果没有重写 hashCode()，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）。

20、& 和 && 的区别?

Java 中 && 和 & 都是表示与的逻辑运算符，都表示逻辑运输符 and，当两边的表达式都为 true 的时候，整个运算结果才为 true，否则为 false。

&&：有短路功能，当第一个表达式的值为 false 的时候，则不再计算第二个表达式；

&：不管第一个表达式结果是否为 true，第二个都会执行。除此之外，& 还可以用作位运算符：当 & 两边的表达式不是 Boolean 类型的时候，& 表示按位操作。

21、Java 中的参数传递时传值呢？还是传引用?

Java 的参数是以值传递的形式传入方法中，而不是引用传递。

当传递方法参数类型为基本数据类型（数字以及布尔值）时，一个方法是不可能修改一个基本数据类型的参数。

当传递方法参数类型为引用数据类型时，一个方法将修改一个引用数据类型的参数所指向对象的值。即使 Java 函数在传递引用数据类型时，也只是拷贝了引用的值罢了，之所以能修改引用数据是因为它们同时指向了一个对象，但这仍然是按值调用而不是引用调用。

22、Java 中的 Math.round(-1.5) 等于多少?

等于 -1，因为在数轴上取值时，中间值（0.5）向右取整，所以正 0.5 是往上取整，负 0.5 是直接舍弃。

23、两个二进制数的异或结果是什么?

两个二进制数异或结果是这两个二进制数差的绝对值。表达式如下：a^b = |a-b|。

两个二进制 a 与 b 异或，即 a 和 b 两个数按位进行运算。如果对应的位相同，则为 0（相当于对应的算术相减），如果不同即为 1（相当于对应的算术相加）。由于二进制每个位只有两种状态，要么是 0，要么是 1，则按位异或操作可表达为按位相减取值相对值，再按位累加。

26、如何实现对象的克隆?

（1）实现 Cloneable 接口并重写 Object 类中的 clone() 方法；

（2）实现 Serializable 接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深克隆。

27、深克隆和浅克隆的区别?

（1）浅克隆：拷贝对象和原始对象的引用类型引用同一个对象。浅克隆只是复制了对象的引用地址，两个对象指向同一个内存地址，所以修改其中任意的值，另一个值都会随之变化，这就是浅克隆。

（2）深克隆：拷贝对象和原始对象的引用类型引用不同对象。深拷贝是将对象及值复制过来，两个对象修改其中任意的值另一个值不会改变，这就是深拷贝（例：JSON.parse() 和 JSON.stringify()，但是此方法无法复制函数类型）。

补充：

深克隆的实现就是在引用类型所在的类实现 Cloneable 接口，并使用 public 访问修饰符重写 clone 方法。

Java 中定义的 clone 没有深浅之分，都是统一的调用 Object 的 clone 方法。为什么会有深克隆的概念？是由于我们在实现的过程中刻意的嵌套了 clone 方法的调用。也就是说深克隆就是在需要克隆的对象类型的类中重新实现克隆方法 clone()。

28、什么是 Java 的序列化，如何实现 Java 的序列化?

对象序列化是一个用于将对象状态转换为字节流的过程，可以将其保存到磁盘文件中或通过网络发送到任何其他程序。从字节流创建对象的相反的过程称为反序列化。而创建的字节流是与平台无关的，在一个平台上序列化的对象可以在不同的平台上反序列化。序列化是为了解决在对象流进行读写操作时所引发的问题。

序列化的实现：将需要被序列化的类实现 Serializable 接口，该接口没有需要实现的方法，只是用于标注该对象是可被序列化的，然后使用一个输出流（如：FileOutputStream）来构造一个 ObjectOutputStream 对象，接着使用 ObjectOutputStream 对象的 writeObject(Object obj) 方法可以将参数为 obj 的对象写出，要恢复的话则使用输入流。

29、什么情况下需要序列化?

（1）当你想把的内存中的对象状态保存到一个文件中或者数据库中时候；

（2）当你想用套接字在网络上传送对象的时候；

（3）当你想通过 RMI 传输对象的时候。

Java泛型与反射

1、Java 的泛型是如何工作的？什么是类型擦除？

泛型是通过类型擦除来实现的，编译器在编译时擦除了所有类型相关的信息，所以在运行时不存在任何类型相关的信息。例如：List<String> 在运行时仅用一个 List 来表示。这样做的目的，是确保能和 Java 5 之前的版本开发二进制类库进行兼容。

类型擦除：泛型信息只存在于代码编译阶段，在进入 JVM 之前，与泛型相关的信息会被擦除掉，专业术语叫做类型擦除。在泛型类被类型擦除的时候，之前泛型类中的类型参数部分如果没有指定上限，如 < T > 则会被转译成普通的 Object 类型，如果指定了上限如 < T extends String > 则类型参数就被替换成类型上限。

补充

```
List<String> list = new ArrayList<String>();
```

1、两个 String 其实只有第一个起作用，后面一个没什么卵用，只不过 JDK7 才开始支持 List<String>list = new ArrayList<> 这种写法。

2、第一个 String 就是告诉编译器，List 中存储的是 String 对象，也就是起类型检查的作用，之后编译器会擦除泛型占位符，以保证兼容以前的代码。

2、什么是泛型中的限定通配符和非限定通配符？

限定通配符对类型进行了限制。有两种限定通配符，一种是< ? extends T > 它通过确保类型必须是 T 的子类来设定类型的上界，另一种是< ? super T >它通过确保类型必须是 T 的父类来设定类型的下界。泛型类型必须用限定内的类型来进行初始化，否则会导致编译错误。另一方面 < ? > 表示了非限定通配符，因为 < ? > 可以用任意类型来替代。

3、List<? extends T> 和 List <? super T> 之间有什么区别？

这两个 List 的声明都是限定通配符的例子，List< ? extends T > 可以接受任何继承自 T 的类型的 List，而List < ? super T > 可以接受任何 T 的父类构成的 List。例如 List< ? extends Number > 可以接受 List< Integer > 或 List< Float > 。

> Array 不支持泛型，要用 List 代替 Array，因为 List 可以提供编译器的类型安全保证，而 Array却不能。

4、Java 中的反射是什么意思？有哪些应用场景?

每个类都有一个 Class 对象，包含了与类有关的信息。当编译一个新类时，会产生一个同名的 .class 文件，该文件内容保存着 Class 对象。类加载相当于 Class 对象的加载，类在第一次使用时才动态加载到 JVM 中。也可以使用 Class.forName("com.mysql.jdbc.Driver") 这种方式来控制类的加载，该方法会返回一个 Class 对象。

反射可以提供运行时的类信息，并且这个类可以在运行时才加载进来，甚至在编译时期该类的 .class 不存在也可以加载进来。Class 和 java.lang.reflect 一起对反射提供了支持，java.lang.reflect 类库主要包含了以下三个类：

（1）Field ： 可以使用 get() 和 set() 方法读取和修改 Field 对象关联的字段；

（2）Method ： 可以使用 invoke() 方法调用与 Method 对象关联的方法；

（3）Constructor ： 可以用 Constructor 创建新的对象。

应用举例：工厂模式，使用反射机制，根据全限定类名获得某个类的 Class 实例。

5、反射的优缺点?

优点：

运行期类型的判断，class.forName() 动态加载类，提高代码的灵活度；

缺点：

尽管反射非常强大，但也不能滥用。如果一个功能可以不用反射完成，那么最好就不用。在我们使用反射技术时，下面几条内容应该牢记于心。

（1）性能开销：反射涉及了动态类型的解析，所以 JVM 无法对这些代码进行优化。因此，反射操作的效率要比那些非反射操作低得多。我们应该避免在经常被执行的代码或对性能要求很高的程序中使用反射。

（2）安全限制：使用反射技术要求程序必须在一个没有安全限制的环境中运行。如果一个程序必须在有安全限制的环境中运行，如 Applet，那么这就是个问题了。

（3）内部暴露：由于反射允许代码执行一些在正常情况下不被允许的操作（比如：访问私有的属性和方法），所以使用反射可能会导致意料之外的副作用，这可能导致代码功能失调并破坏可移植性。反射代码破坏了抽象性，因此当平台发生改变的时候，代码的行为就有可能也随着变化。

32、Java 中的动态代理是什么？有哪些应用?

动态代理：当想要给实现了某个接口的类中的方法，加一些额外的处理。比如说加日志，加事务等。可以给这个类创建一个代理，故名思议就是创建一个新的类，这个类不仅包含原来类方法的功能，而且还在原来的基础上添加了额外处理的新功能。这个代理类并不是定义好的，是动态生成的。具有解耦意义，灵活，扩展性强。

动态代理的应用：Spring 的 AOP 、加事务、加权限、加日志。

6、怎么实现动态代理?

首先必须定义一个接口，还要有一个 InvocationHandler（将实现接口的类的对象传递给它）处理类。再有一个工具类 Proxy（习惯性将其称为代理类，因为调用它的 newInstance() 可以产生代理对象，其实它只是一个产生代理对象的工具类）。利用到 InvocationHandler，拼接代理类源码，将其编译生成代理类的二进制码，利用加载器加载，并将其实例化产生代理对象，最后返回。

每一个动态代理类都必须要实现 InvocationHandler 这个接口，并且每个代理类的实例都关联到了一个 handler，当我们通过代理对象调用一个方法的时候，这个方法的调用就会被转发为由 InvocationHandler 这个接口的 invoke 方法来进行调用。我们来看看 InvocationHandler 这个接口的唯一一个方法 invoke 方法：

```
Object invoke(Object proxy, Method method, Object[] args) throws Throwable
```

proxy: 指代我们所代理的那个真实对象

method: 指代的是我们所要调用真实对象的某个方法的 Method 对象

args: 指代的是调用真实对象某个方法时接受的参数

Proxy 类的作用是动态创建一个代理对象的类。它提供了许多的方法，但是我们用的最多的就是 newProxyInstance 这个方法：

```
public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces,
 InvocationHandler handler)  throws IllegalArgumentException
```

loader：一个 ClassLoader 对象，定义了由哪个 ClassLoader 对象来对生成的代理对象进行加载；

interfaces：一个 Interface 对象的数组，表示的是我将要给我需要代理的对象提供一组什么接口，如果我提供了一组接口给它，那么这个代理对象就宣称实现了该接口(多态)，这样我就能调用这组接口中的方法了

handler：一个 InvocationHandler 对象，表示的是当我这个动态代理对象在调用方法的时候，会关联到哪一个 InvocationHandler 对象上。

通过 Proxy.newProxyInstance 创建的代理对象是在 Jvm 运行时动态生成的一个对象，它并不是我们的 InvocationHandler 类型，也不是我们定义的那组接口的类型，而是在运行是动态生成的一个对象。

Java字符串

1、字节和字符的区别?

字节是存储容量的基本单位；

字符是数字、字母、汉字以及其他语言的各种符号；

1 字节 = 8 个二进制单位，一个字符由一个字节或多个字节的二进制单位组成。

2、String 为什么要设计为不可变类?

在 Java 中将 String 设计成不可变的是综合考虑到各种因素的结果。主要的原因主要有以下三点：

（1）字符串常量池的需要：字符串常量池是 Java 堆内存中一个特殊的存储区域, 当创建一个 String 对象时，假如此字符串值已经存在于常量池中，则不会创建一个新的对象，而是引用已经存在的对象；

（2）允许 String 对象缓存 HashCode：Java 中 String 对象的哈希码被频繁地使用, 比如在 HashMap 等容器中。字符串不变性保证了 hash 码的唯一性，因此可以放心地进行缓存。这也是一种性能优化手段，意味着不必每次都去计算新的哈希码；

（3）String 被许多的 Java 类(库)用来当做参数，例如：网络连接地址 URL、文件路径 path、还有反射机制所需要的 String 参数等, 假若 String 不是固定不变的，将会引起各种安全隐患。

3、String、StringBuilder、StringBuffer 的区别?

String：用于字符串操作，属于不可变类；【补充：String 不是基本数据类型，是引用类型，底层用 char 数组实现的】

StringBuilder：与 StringBuffer 类似，都是字符串缓冲区，但线程不安全；

StringBuffer：也用于字符串操作，不同之处是 StringBuffer 属于可变类，对方法加了同步锁，线程安全

StringBuffer的补充

说明：StringBuffer 中并不是所有方法都使用了 Synchronized 修饰来实现同步：

```
@Overridepublic StringBuffer insert(int dstOffset, CharSequence s) {
  // Note, synchronization achieved via invocations of other StringBuffer methods
  // after narrowing of s to specific type
  // Ditto for toStringCache clearing
  super.insert(dstOffset, s);
  return this;
}
```

执行效率：StringBuilder > StringBuffer > String

4、String 字符串修改实现的原理?

当用 String 类型来对字符串进行修改时，其实现方法是首先创建一个 StringBuffer，其次调用 StringBuffer 的 append() 方法，最后调用 StringBuffer 的 toString() 方法把结果返回。

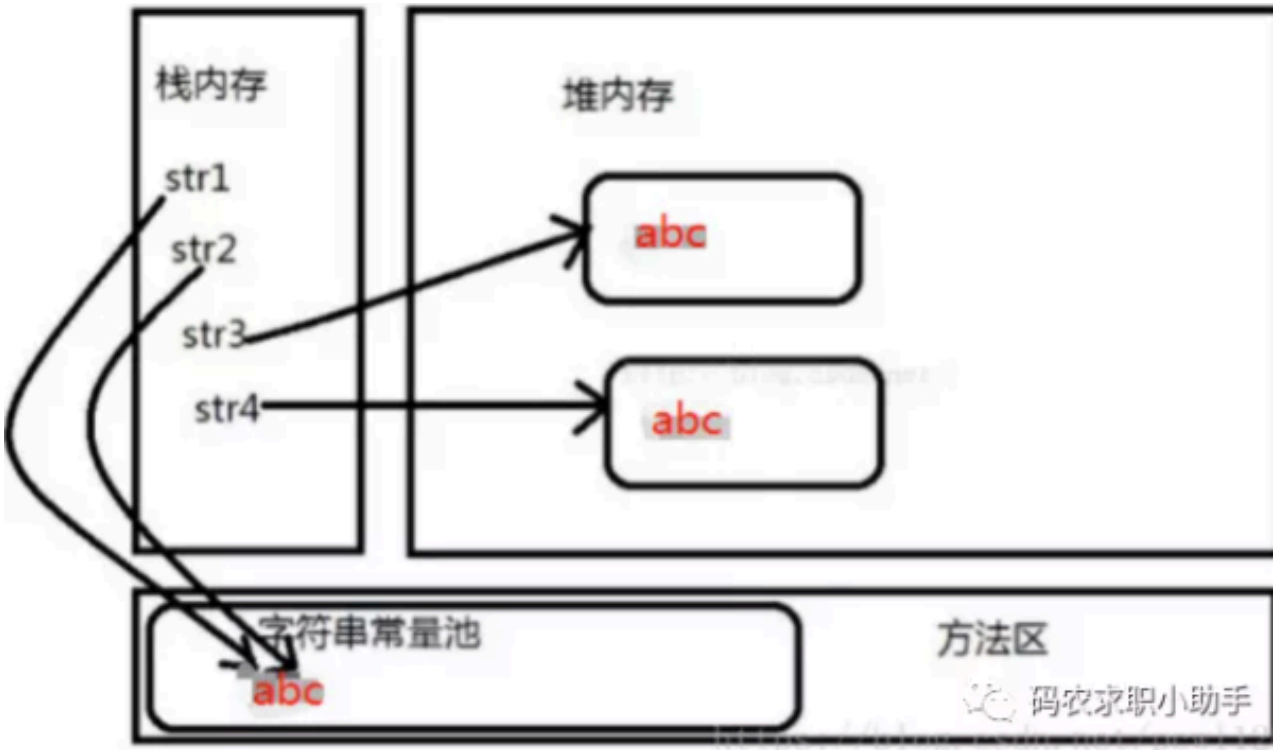
5、String str = "i" 与 String str = new String("i") 一样吗?

不一样，因为内存的分配方式不一样。String str = "i" 的方式，Java 虚拟机会将其分配到常量池中；而 String str = new String("i") 则会被分到堆内存中。

```
public class StringTest {
  public static void main(String[] args) {
    String str1 = "abc";
    String str2 = "abc";
    String str3 = new String("abc");
    String str4 = new String("abc");
    System.out.println(str1 == str2);      // true
    System.out.println(str1 == str3);      // false
    System.out.println(str3 == str4);      // false
    System.out.println(str3.equals(str4)); // true
   }
}
```

在执行 String str1 = "abc" 的时候，JVM 会首先检查字符串常量池中是否已经存在该字符串对象，如果已经存在，那么就不会再创建了，直接返回该字符串在字符串常量池中的内存地址；如果该字符串还不存在字符串常量池中，那么就会在字符串常量池中创建该字符串对象，然后再返回。所以在执行 String str2 = "abc" 的时候，因为字符串常量池中已经存在“abc”字符串对象了，就不会在字符串常量池中再次创建了，所以栈内存中 str1 和 str2 的内存地址都是指向 "abc" 在字符串常量池中的位置，所以 str1 = str2 的运行结果为 true。

而在执行 String str3 = new String("abc") 的时候，JVM 会首先检查字符串常量池中是否已经存在“abc”字符串，如果已经存在，则不会在字符串常量池中再创建了；如果不存在，则就会在字符串常量池中创建 "abc" 字符串对象，然后再到堆内存中再创建一份字符串对象，把字符串常量池中的 "abc" 字符串内容拷贝到内存中的字符串对象中，然后返回堆内存中该字符串的内存地址，即栈内存中存储的地址是堆内存中对象的内存地址。String str4 = new String("abc") 是在堆内存中又创建了一个对象，所以 str 3 == str4 运行的结果是 false。str1、str2、str3、str4 在内存中的存储状况如下图所示：



6、String 类的常用方法都有那些?

indexOf()：返回指定字符的索引。

charAt()：返回指定索引处的字符。

replace()：字符串替换。

trim()：去除字符串两端空白。

split()：分割字符串，返回一个分割后的字符串数组。

getBytes()：返回字符串的 byte 类型数组。

length()：返回字符串长度。

toLowerCase()：将字符串转成小写字母。

toUpperCase()：将字符串转成大写字符。

substring()：截取字符串。

equals()：字符串比较。

7、final 修饰 StringBuffer 后还可以 append 吗?

可以。final 修饰的是一个引用变量，那么这个引用始终只能指向这个对象，但是这个对象内部的属性是可以变化的。

> 官方文档解释：once a final variable has been assigned, it always contains the same value. If a final variable holds a reference to an object, then the state of the object may be changed by operations on the object, but the variable will always refer to the same object.

Java异常

1、finally 块中的代码什么时候被执行?

在 Java 语言的异常处理中，finally 块的作用就是为了保证无论出现什么情况，finally 块里的代码一定会被执行。由于程序执行 return 就意味着结束对当前函数的调用并跳出这个函数体，因此任何语句要执行都只能在 return 前执行（除非碰到 exit 函数），因此 finally 块里的代码也是在 return 之前执行的。

此外，如果 try-finally 或者 catch-finally 中都有 return，那么 finally 块中的 return 将会覆盖别处的 return 语句，最终返回到调用者那里的是 finally 中 return 的值。

2、finally 是不是一定会被执行到?

不一定。下面列举两种执行不到的情况：

（1）当程序进入 try 块之前就出现异常时，会直接结束，不会执行 finally 块中的代码；

（2）当程序在 try 块中强制退出时也不会去执行 finally 块中的代码，比如在 try 块中执行 exit 方法。

3、try-catch-finally 中，如果 catch 中 return 了，finally 还会执行吗?

会。程序在执行到 return 时会首先将返回值存储在一个指定的位置，其次去执行 finally 块，最后再返回。因此，对基本数据类型，在 finally 块中改变 return 的值没有任何影响，直接覆盖掉；而对引用类型是有影响的，返回的是在 finally 对 前面 return 语句返回对象的修改值。

4、try-catch-finally 中那个部分可以省略?

catch 可以省略。try 只适合处理运行时异常，try+catch 适合处理运行时异常+普通异常。也就是说，如果你只用 try 去处理普通异常却不加以 catch 处理，编译是通不过的，因为编译器硬性规定，普通异常如果选择捕获，则必须用 catch 显示声明以便进一步处理。而运行时异常在编译时没有如此规定，所以 catch 可以省略，你加上 catch 编译器也觉得无可厚非。

5、Error 和 Exception 的区别?

Error 类和 Exception 类的父类都是 Throwable 类。主要区别如下：

Error 类： 一般是指与虚拟机相关的问题，如：系统崩溃、虚拟机错误、内存空间不足、方法调用栈溢出等。这类错误将会导致应用程序中断，仅靠程序本身无法恢复和预防；

Exception 类：分为运行时异常和受检查的异常。

6、运行时异常与受检异常有何异同?

运行时异常：如：空指针异常、指定的类找不到、数组越界、方法传递参数错误、数据类型转换错误。可以编译通过，但是一运行就停止了，程序不会自己处理；

受检查异常：要么用 try ... catch... 捕获，要么用 throws 声明抛出，交给父类处理。

7、throw 和 throws 的区别?

（1）throw：在方法体内部，表示抛出异常，由方法体内部的语句处理；throw 是具体向外抛出异常的动作，所以它抛出的是一个异常实例；

（2）throws：在方法声明后面，表示如果抛出异常，由该方法的调用者来进行异常的处理；表示出现异常的可能性，并不一定会发生这种异常。

8、常见的异常类有哪些?

NullPointerException：当应用程序试图访问空对象时，则抛出该异常。

SQLException：提供关于数据库访问错误或其他错误信息的异常。

IndexOutOfBoundsException：指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。

FileNotFoundException：当试图打开指定路径名表示的文件失败时，抛出此异常。

IOException：当发生某种 I/O 异常时，抛出此异常。此类是失败或中断的 I/O 操作生成的异常的通用类。

ClassCastException：当试图将对象强制转换为不是实例的子类时，抛出该异常。

IllegalArgumentException：抛出的异常表明向方法传递了一个不合法或不正确的参数。

9、主线程可以捕获到子线程的异常吗?

线程设计的理念：“线程的问题应该线程自己本身来解决，而不要委托到外部”。

正常情况下，如果不做特殊的处理，在主线程中是不能够捕获到子线程中的异常的。如果想要在主线程中捕获子线程的异常，我们可以用如下的方式进行处理，使用 Thread 的静态方法

```
Thread.setDefaultUncaughtExceptionHandler(new MyUncaughtExceptionHandle());
```

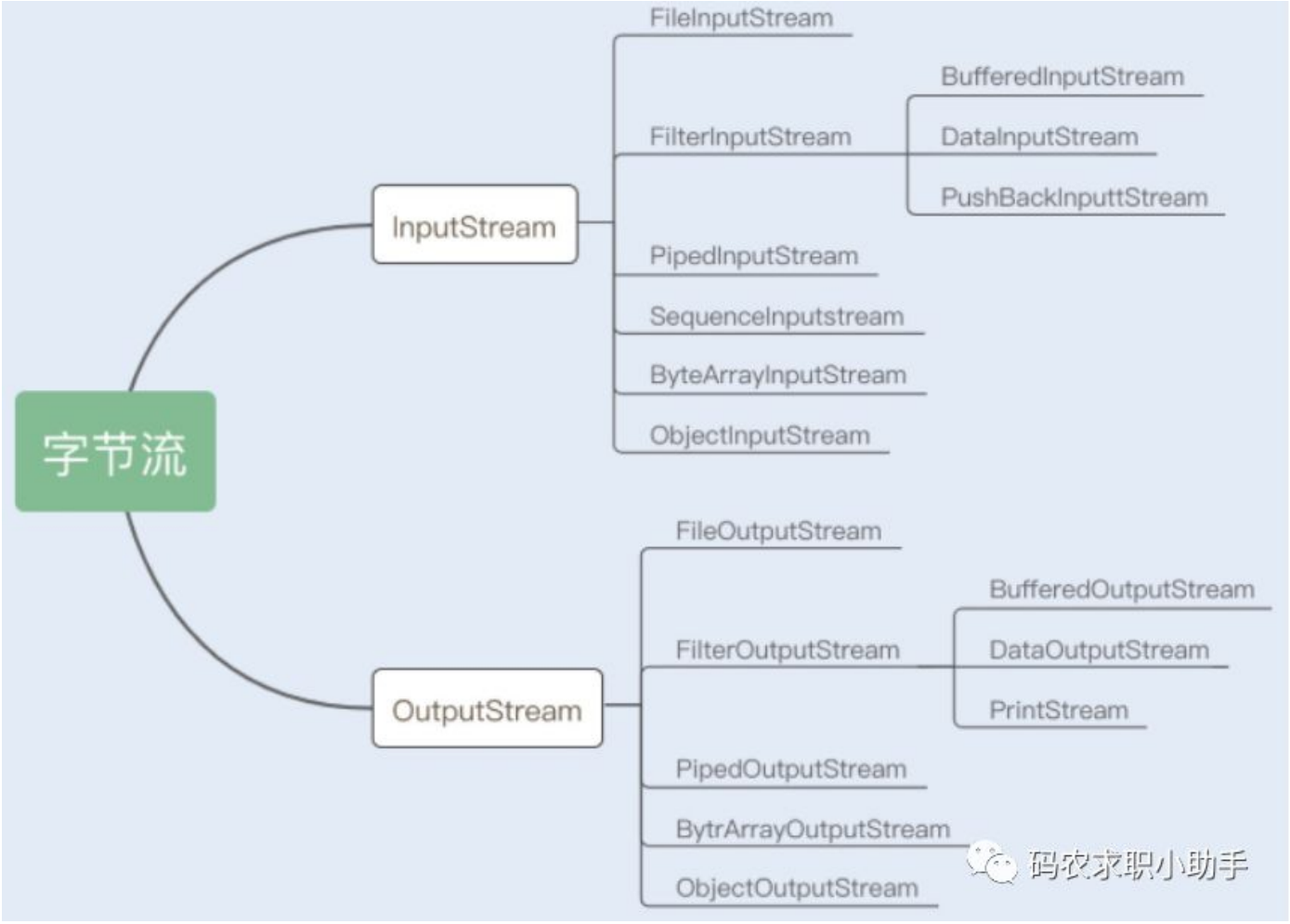
Java IO

1、Java 中的 IO 流的分类？说出几个你熟悉的实现类?

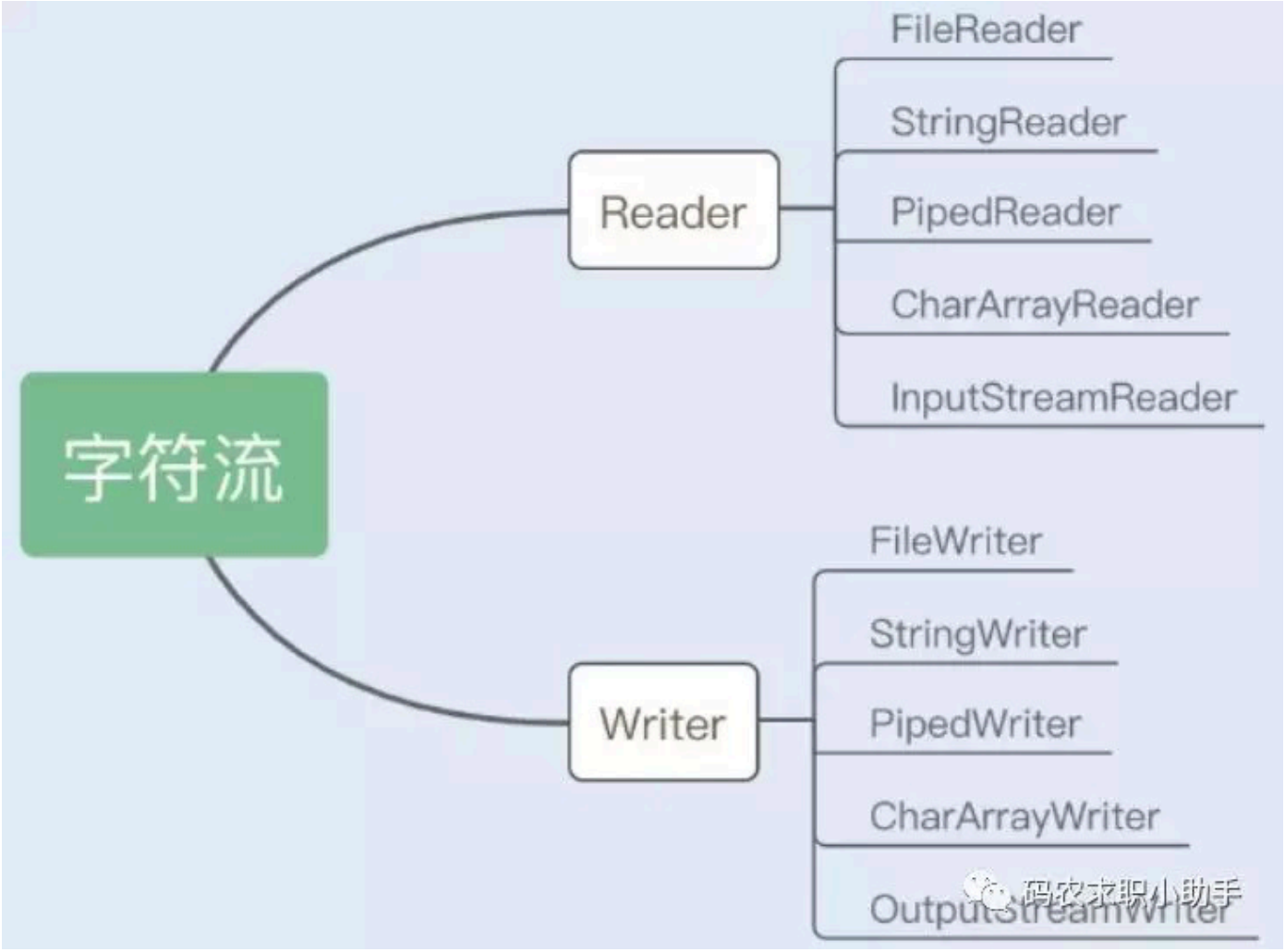
按功能来分：输入流（input）、输出流（output）。

按类型来分：字节流 和 字符流。

字节流：InputStream/OutputStream 是字节流的抽象类，这两个抽象类又派生了若干子类，不同的子类分别处理不同的操作类型。具体子类如下所示：



字符流：Reader/Writer 是字符的抽象类，这两个抽象类也派生了若干子类，不同的子类分别处理不同的操作类型。



2、字节流和字符流有什么区别?

字节流按 8 位传输，以字节为单位输入输出数据，字符流按 16 位传输，以字符为单位输入输出数据。

但是不管文件读写还是网络发送接收，信息的最小存储单元都是字节。

3、BIO、NIO、AIO 有什么区别?

BIO：Block IO 同步阻塞式 IO，就是我们平常使用的传统 IO，它的特点是模式简单使用方便，并发处理能力低。同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。

NIO：New IO 同步非阻塞 IO，是传统 IO 的升级，客户端和服务器端通过 Channel（通道）通讯，实现了多路复用。NIO 是一种同步非阻塞的 I/O 模型，在 Java1.4 中引入了 NIO 框架，对应 java.nio 包，提供了 Channel , Selector，Buffer 等抽象。NIO 中的 N 可以理解为 Non-blocking，不单纯是 New。它支持面向缓冲的，基于通道的 I/O 操作方法。NIO 提供了与传统BIO模型中的 Socket 和 ServerSocket 相对应的 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现，两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞 I/O 来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发。

AIO：Asynchronous IO 是 NIO 的升级，也叫 NIO2，实现了异步非堵塞 IO ，异步 IO 的操作基于事件和回调机制。也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步 IO 的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO操作本身是同步的。

Java集合

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**Java面试题**」，即可获取最新版的 PDF 哦，扫码直达

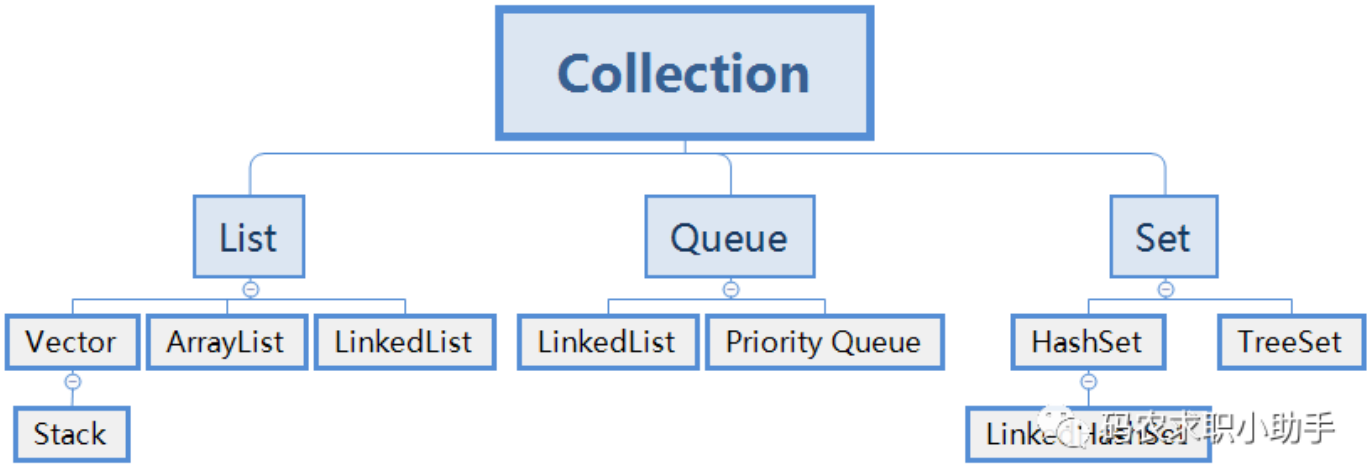
关注后，回复「**Java面试题**」，即可获取最新版 PDF。

另外，也建议大家来网站读，因为如果有错误，我会及时在网站更改，PDF 很难及时更新。

在线网站：https://www.iamshuaidi.com

1、Java 中常用的容器有哪些?

常见容器主要包括 Collection 和 Map 两种，Collection 存储着对象的集合，而 Map 存储着键值对（两个对象）的映射表



Collection

- **Set**

1. SetTreeSet：基于红黑树实现，支持有序性操作，例如：根据一个范围查找元素的操作。但是查找效率不如 HashSet，HashSet 查找的时间复杂度为 O(1)，TreeSet 则为 O(logN)。
2. HashSet：基于哈希表实现，支持快速查找，但不支持有序性操作。并且失去了元素的插入顺序信息，也就是说使用 Iterator 遍历 HashSet 得到的结果是不确定的。
3. LinkedHashSet：具有 HashSet 的查找效率，且内部使用双向链表维护元素的插入顺序。

- **List**

1. ArrayList：基于动态数组实现，支持随机访问。
2. Vector：和 ArrayList 类似，但它是线程安全的。
3. LinkedList：基于双向链表实现，只能顺序访问，但是可以快速地在链表中间插入和删除元素。不仅如此，LinkedList 还可以用作栈、队列和双向队列。

- **Queue**

1. LinkedList：可以用它来实现双向队列。
2. PriorityQueue：基于堆结构实现，可以用它来实现优先队列。

Map

1. TreeMap：基于红黑树实现。
2. HashMap：基于哈希表实现。
3. HashTable：和 HashMap 类似，但它是线程安全的，这意味着同一时刻多个线程可以同时写入 HashTable 并且不会导致数据不一致。它是遗留类，不应该去使用它。现在可以使用 ConcurrentHashMap 来支持线程安全，并且 ConcurrentHashMap 的效率会更高，因为 ConcurrentHashMap 引入了分段锁。
4. LinkedHashMap：使用双向链表来维护元素的顺序，顺序为插入顺序或者最近最少使用（LRU）顺序。

2、ArrayList 和 LinkedList 的区别?

ArrayList：底层是基于数组实现的，查找快，增删较慢；

LinkedList：底层是基于链表实现的。确切的说是循环双向链表（JDK1.6 之前是双向循环链表、JDK1.7 之后取消了循环），查找慢、增删快。LinkedList 链表由一系列表项连接而成，一个表项包含 3 个部分：元素内容、前驱表和后驱表。链表内部有一个 header 表项，既是链表的开始也是链表的结尾。header 的后继表项是链表中的第一个元素，header 的前驱表项是链表中的最后一个元素。

ArrayList 的增删未必就是比 LinkedList 要慢：

1. 如果增删都是在末尾来操作【每次调用的都是 remove() 和 add()】，此时 ArrayList 就不需要移动和复制数组来进行操作了。如果数据量有百万级的时，速度是会比 LinkedList 要快的。
2. 如果删除操作的位置是在中间。由于 LinkedList 的消耗主要是在遍历上，ArrayList 的消耗主要是在移动和复制上（底层调用的是 arrayCopy() 方法，是 native 方法）。LinkedList 的遍历速度是要慢于 ArrayList 的复制移动速度的如果数据量有百万级的时，还是 ArrayList 要快。

3、ArrayList 实现 RandomAccess 接口有何作用？为何 LinkedList 却没实现这个接口?

1. RandomAccess 接口只是一个标志接口，只要 List 集合实现这个接口，就能支持快速随机访问。通过查看 Collections 类中的 binarySearch() 方法，可以看出，判断 List 是否实现 RandomAccess 接口来实行 indexedBinarySerach(list, key) 或 iteratorBinarySerach(list, key)方法。再通过查看这两个方法的源码发现：实现 RandomAccess 接口的 List 集合采用一般的 for 循环遍历，而未实现这接口则采用迭代器，即 ArrayList 一般采用 for 循环遍历，而 LinkedList 一般采用迭代器遍历；
2. ArrayList 用 for 循环遍历比 iterator 迭代器遍历快，LinkedList 用 iterator 迭代器遍历比 for 循环遍历快。所以说，当我们在做项目时，应该考虑到 List 集合的不同子类采用不同的遍历方式，能够提高性能。

4、ArrayList 的扩容机制?

推荐阅读：

https://juejin.im/post/5d42ab5e5188255d691bc8d6

1. 当使用 add 方法的时候首先调用 ensureCapacityInternal 方法，传入 size+1 进去，检查是否需要扩充 elementData 数组的大小；
2. newCapacity = 扩充数组为原来的 1.5 倍(不能自定义)，如果还不够，就使用它指定要扩充的大小 minCapacity ，然后判断 minCapacity 是否大于 MAX_ARRAY_SIZE(Integer.MAX_VALUE - 8) ，如果大于，就取 Integer.MAX_VALUE；
3. 扩容的主要方法：grow；
4. ArrayList 中 copy 数组的核心就是 System.arraycopy 方法，将 original 数组的所有数据复制到 copy 数组中，这是一个本地方法。

5、Array 和 ArrayList 有何区别？什么时候更适合用 Array?

1. Array 可以容纳基本类型和对象，而 ArrayList 只能容纳对象；
2. Array 是指定大小的，而 ArrayList 大小是固定的。

什么时候更适合使用 Array：

1. 如果列表的大小已经指定，大部分情况下是存储和遍历它们；
2. 对于遍历基本数据类型，尽管 Collections 使用自动装箱来减轻编码任务，在指定大小的基本类型的列表上工作也会变得很慢；
3. 如果你要使用多维数组，使用 [ ][ ] 比 List> 更容易。

6、HashMap 的实现原理/底层数据结构？JDK1.7 和 JDK1.8

JDK1.7：Entry数组 + 链表

JDK1.8：Node 数组 + 链表/红黑树，当链表上的元素个数超过 8 个并且数组长度 >= 64 时自动转化成红黑树，节点变成树节点，以提高搜索效率和插入效率到 O(logN)。Entry 和 Node 都包含 key、value、hash、next 属性。

7、HashMap 的 put 方法的执行过程?

当我们想往一个 HashMap 中添加一对 key-value 时，系统首先会计算 key 的 hash 值，然后根据 hash 值确认在 table 中存储的位置。若该位置没有元素，则直接插入。否则迭代该处元素链表并依次比较其 key 的 hash 值。如果两个 hash 值相等且 key 值相等(e.hash == hash && ((k = e.key) == key || key.equals(k)))，则用新的 Entry 的 value 覆盖原来节点的 value。如果两个 hash 值相等但 key 值不等 ，则将该节点插入该链表的链头。

8、HashMap 的 get 方法的执行过程?

通过 key 的 hash 值找到在 table 数组中的索引处的 Entry，然后返回该 key 对应的 value 即可。

在这里能够根据 key 快速的取到 value 除了和 HashMap 的数据结构密不可分外，还和 Entry 有莫大的关系。HashMap 在存储过程中并没有将 key，value 分开来存储，而是当做一个整体 key-value 来处理的，这个整体就是 Entry 对象。同时 value 也只相当于 key 的附属而已。在存储的过程中，系统根据 key 的 HashCode 来决定 Entry 在 table 数组中的存储位置，在取的过程中同样根据 key 的 HashCode 取出相对应的 Entry 对象（value 就包含在里面）。

9、HashMap 的 resize 方法的执行过程?

有两种情况会调用 resize 方法：

1. 第一次调用 HashMap 的 put 方法时，会调用 resize 方法对 table 数组进行初始化，如果不传入指定值，默认大小为 16。
2. 扩容时会调用 resize，即 size > threshold 时，table 数组大小翻倍。

每次扩容之后容量都是**翻倍**。扩容后要将原数组中的所有元素找到在新数组中合适的位置。

当我们把 table[i] 位置的所有 Node 迁移到 newtab 中去的时候：这里面的 node 要么在 newtab 的 i 位置（不变），要么在 newtab 的 i + n 位置。也就是我们可以这样处理：把 table[i] 这个桶中的 node 拆分为两个链表 l1 和 l2：如果 hash & n == 0，那么当前这个 node 被连接到 l1 链表；否则连接到 l2 链表。这样下来，当遍历完 table[i] 处的所有 node 的时候，我们得到两个链表 l1 和 l2，这时我们令 newtab[i] = l1，newtab[i + n] = l2，这就完成了 table[i] 位置所有 node 的迁移（rehash），这也是 HashMap 中容量一定的是 2 的整数次幂带来的方便之处。

10、HashMap 的 size 为什么必须是 2 的整数次方?

1. 这样做总是能够保证 HashMap 的底层数组长度为 2 的 n 次方。当 length 为 2 的 n 次方时，h & (length - 1) 就相当于对 length 取模，而且速度比直接取模快得多，这是 HashMap 在速度上的一个优化。而且每次扩容时都是翻倍。
2. 如果 length 为 2 的次幂，则 length - 1 转化为二进制必定是 11111……的形式，在与 h 的二进制进行与操作时效率会非常的快，而且空间不浪费。但是，如果 length 不是 2 的次幂，比如：length 为 15，则 length - 1 为 14，对应的二进制为 1110，在于 h 与操作，最后一位都为 0 ，而 0001，0011，0101，1001，1011，0111，1101 这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率，这样就会造成空间的浪费。

11、HashMap 多线程死循环问题?

详细请阅读：

https://blog.csdn.net/xuefeng0707/article/details/40797085

https://blog.csdn.net/dgutliangxuan/article/details/78779448

主要是多线程同时 put 时，如果同时触发了 rehash 操作，会导致 HashMap 中的链表中出现循环节点，进而使得后面 get 的时候，会死循环。

12、HashMap 的 get 方法能否判断某个元素是否在 map 中?

HashMap 的 get 函数的返回值不能判断一个 key 是否包含在 map 中，因为 get 返回 null 有可能是不包含该 key，也有可能该 key 对应的 value 为 null。因为 HashMap 中允许 key 为 null，也允许 value 为 null。

13、HashMap 与 HashTable 的区别是什么?

1. HashTable 基于 Dictionary 类，而 HashMap 是基于 AbstractMap。Dictionary 是任何可将键映射到相应值的类的抽象父类，而 AbstractMap 是基于 Map 接口的实现，它以最大限度地减少实现此接口所需的工作。
2. HashMap 的 key 和 value 都允许为 null，而 Hashtable 的 key 和 value 都不允许为 null。HashMap 遇到 key 为 null 的时候，调用 putForNullKey 方法进行处理，而对 value 没有处理；Hashtable 遇到 null，直接返回 NullPointerException。
3. Hashtable 是线程安全的，而 HashMap 不是线程安全的，但是我们也可以通过 Collections.synchronizedMap(hashMap)，使其实现同步。

HashTable的补充：

HashTable 和 HashMap 的实现原理几乎一样，差别无非是

1. HashTable 不允许 key 和 value 为 null；
2. HashTable 是线程安全的。但是 HashTable 线程安全的策略实现代价却太大了，简单粗暴，get/put 所有相关操作都是 synchronized 的，这相当于给整个哈希表加了一把大锁，多线程访问时候，只要有一个线程访问或操作该对象，那其他线程只能阻塞，相当于将所有的操作串行化，在竞争激烈的并发场景中性能就会非常差。

14、HashMap 与 ConcurrentHashMap 的区别是什么?

HashMap 不是线程安全的，而 ConcurrentHashMap 是线程安全的。

ConcurrentHashMap 采用锁分段技术，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，这样做明显减小了锁的粒度。

15、HashTable 和 ConcurrentHashMap 的区别?

HashTable 和 ConcurrentHashMap 相比，效率低。 Hashtable 之所以效率低主要是使用了 synchronized 关键字对 put 等操作进行加锁，而 synchronized 关键字加锁是对整张 Hash 表的，即每次锁住整张表让线程独占，致使效率低下，而 ConcurrentHashMap 在对象中保存了一个 Segment 数组，即将整个 Hash 表划分为多个分段；而每个Segment元素，即每个分段则类似于一个Hashtable；这样，在执行 put 操作时首先根据 hash 算法定位到

元素属于哪个 Segment，然后对该 Segment 加锁即可，因此， ConcurrentHashMap 在多线程并发编程中可是实现多线程 put操作。

16、ConcurrentHashMap 的实现原理是什么?

数据结构

JDK 7：中 ConcurrentHashMap 采用了数组 + Segment + 分段锁的方式实现。

JDK 8：中 ConcurrentHashMap 参考了 JDK 8 HashMap 的实现，采用了数组 + 链表 + 红黑树的实现方式来设计，内部大量采用 CAS 操作。

ConcurrentHashMap 采用了非常精妙的"分段锁"策略，ConcurrentHashMap 的主干是个 Segment 数组。

```
final  Segment<K,V>[]  segments;
```

Segment 继承了 ReentrantLock，所以它就是一种可重入锁（ReentrantLock)。在 ConcurrentHashMap，一个 Segment 就是一个子哈希表，Segment 里维护了一个 HashEntry 数组，并发环境下，对于不同 Segment 的数据进行操作是不用考虑锁竞争的。就按默认的 ConcurrentLevel 为 16 来讲，理论上就允许 16 个线程并发执行。

所以，对于同一个 Segment 的操作才需考虑线程同步，不同的 Segment 则无需考虑。Segment 类似于 HashMap，一个 Segment 维护着一个HashEntry 数组：

```
transient  volatile  HashEntry<K,V>[]  table;
```

HashEntry 是目前我们提到的最小的逻辑处理单元了。一个 ConcurrentHashMap 维护一个 Segment 数组，一个 Segment 维护一个 HashEntry 数组。因此，ConcurrentHashMap 定位一个元素的过程需要进行两次 Hash 操作。第一次 Hash 定位到 Segment，第二次 Hash 定位到元素所在的链表的头部。

17、HashSet 的实现原理?

HashSet 的实现是依赖于 HashMap 的，HashSet 的值都是存储在 HashMap 中的。在 HashSet 的构造法中会初始化一个 HashMap 对象，HashSet 不允许值重复。因此，HashSet 的值是作为 HashMap 的 key 存储在 HashMap 中的，当存储的值已经存在时返回 false。

18、HashSet 怎么保证元素不重复的?

```
public boolean add(E e) {
  return map.put(e, PRESENT)==null;
 }
```

元素值作为的是 map 的 key，map 的 value 则是 PRESENT 变量，这个变量只作为放入 map 时的一个占位符而存在，所以没什么实际用处。其实，这时候答案已经出来了：HashMap 的 key 是不能重复的，而这里HashSet 的元素又是作为了 map 的 key，当然也不能重复了。

19、LinkedHashMap 的实现原理?

LinkedHashMap 也是基于 HashMap 实现的，不同的是它定义了一个 Entry header，这个 header 不是放在 Table 里，它是额外独立出来的。LinkedHashMap 通过继承 hashMap 中的 Entry，并添加两个属性 Entry before，after 和 header 结合起来组成一个双向链表，来实现按插入顺序或访问顺序排序。

LinkedHashMap 定义了排序模式 accessOrder，该属性为 boolean 型变量，对于访问顺序，为 true；对于插入顺序，则为 false。一般情况下，不必指定排序模式，其迭代顺序即为默认为插入顺序。

20、Iterator 怎么使用？有什么特点?

迭代器是一种设计模式，它是一个对象，它可以遍历并选择序列中的对象，而开发人员不需要了解该序列的底层结构。迭代器通常被称为“轻量级”对象，因为创建它的代价小。Java 中的 Iterator 功能比较简单，并且只能单向移动：

1. 使用方法 iterator() 要求容器返回一个 Iterator。第一次调用 Iterator 的 next() 方法时，它返回序列的第一个元素。注意：iterator() 方法是 java.lang.Iterable 接口，被 Collection 继承。
2. 使用 next() 获得序列中的下一个元素。
3. 使用 hasNext() 检查序列中是否还有元素。
4. 使用 remove() 将迭代器新返回的元素删除。

21、 Iterator 和 ListIterator 有什么区别?

Iterator 可用来遍历 Set 和 List 集合，但是 ListIterator 只能用来遍历 List。Iterator 对集合只能是前向遍历，ListIterator 既可以前向也可以后向。ListIterator 实现了 Iterator 接口，并包含其他的功能，比如：增加元素，替换元素，获取前一个和后一个元素的索引等等。

22、Iterator 和 Enumeration 接口的区别?

与 Enumeration 相比，Iterator 更加安全，因为当一个集合正在被遍历的时候，它会阻止其它线程去修改集合。否则会抛出 ConcurrentModificationException 异常。这其实就是 fail-fast 机制。具体区别有三点：

1. Iterator 的方法名比 Enumeration 更科学；
2. Iterator 有 fail-fast 机制，比 Enumeration 更安全；
3. Iterator 能够删除元素，Enumeration 并不能删除元素。

23、fail-fast 与 fail-safe 有什么区别?

Iterator 的 fail-fast 属性与当前的集合共同起作用，因此它不会受到集合中任何改动的影响。Java.util 包中的所有集合类都被设计为 fail-fast 的，而 java.util.concurrent 中的集合类都为 fail-safe 的。当检测到正在遍历的集合的结构被改变时，fail-fast 迭代器抛出ConcurrentModificationException，而 fail-safe 迭代器从不抛出 ConcurrentModificationException。

24、Collection 和 Collections 有什么区别?

Collection：是最基本的集合接口，一个 Collection 代表一组 Object，即 Collection 的元素。它的直接继承接口有 List，Set 和 Queue。

Collections：是不属于 Java 的集合框架的，它是集合类的一个工具类/帮助类。此类不能被实例化， 服务于 Java 的 Collection 框架。它包含有关集合操作的静态多态方法，实现对各种集合的搜索、排序、线程安全等操作。

Java并发

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**Java面试题**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**Java面试题**」，即可获取最新版 PDF。

另外，也建议大家来网站读，因为如果有错误，我会及时在网站更改，PDF 很难及时更新。

在线网站：https://www.iamshuaidi.com

1、并行和并发有什么区别?

1. 并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔发生；
2. 并行是在不同实体上的多个事件，并发是在同一实体上的多个事件；
3. 在一台处理器上“同时”处理多个任务，在多台处理器上同时处理多个任务。如 Hadoop 分布式集群。所以并发编程的目标是充分的利用处理器的每一个核，以达到最高的处理性能。

2、线程和进程的区别?

进程：是程序运行和资源分配的基本单位，一个程序至少有一个进程，一个进程至少有一个线程。进程在执行过程中拥有独立的内存单元，而多个线程共享内存资源，减少切换次数，从而效率更高。

线程：是进程的一个实体，是 cpu 调度和分派的基本单位，是比程序更小的能独立运行的基本单位。同一进程中的多个线程之间可以并发执行。

3、守护线程是什么?

守护线程（即 Daemon thread），是个服务线程，准确地来说就是服务其他的线程

4、创建线程的几种方式?

1. 继承 Thread 类创建线程；
2. 实现 Runnable 接口创建线程；
3. 通过 Callable 和 Future 创建线程；
4. 通过线程池创建线程。

5、Runnable 和 Callable 有什么区别?

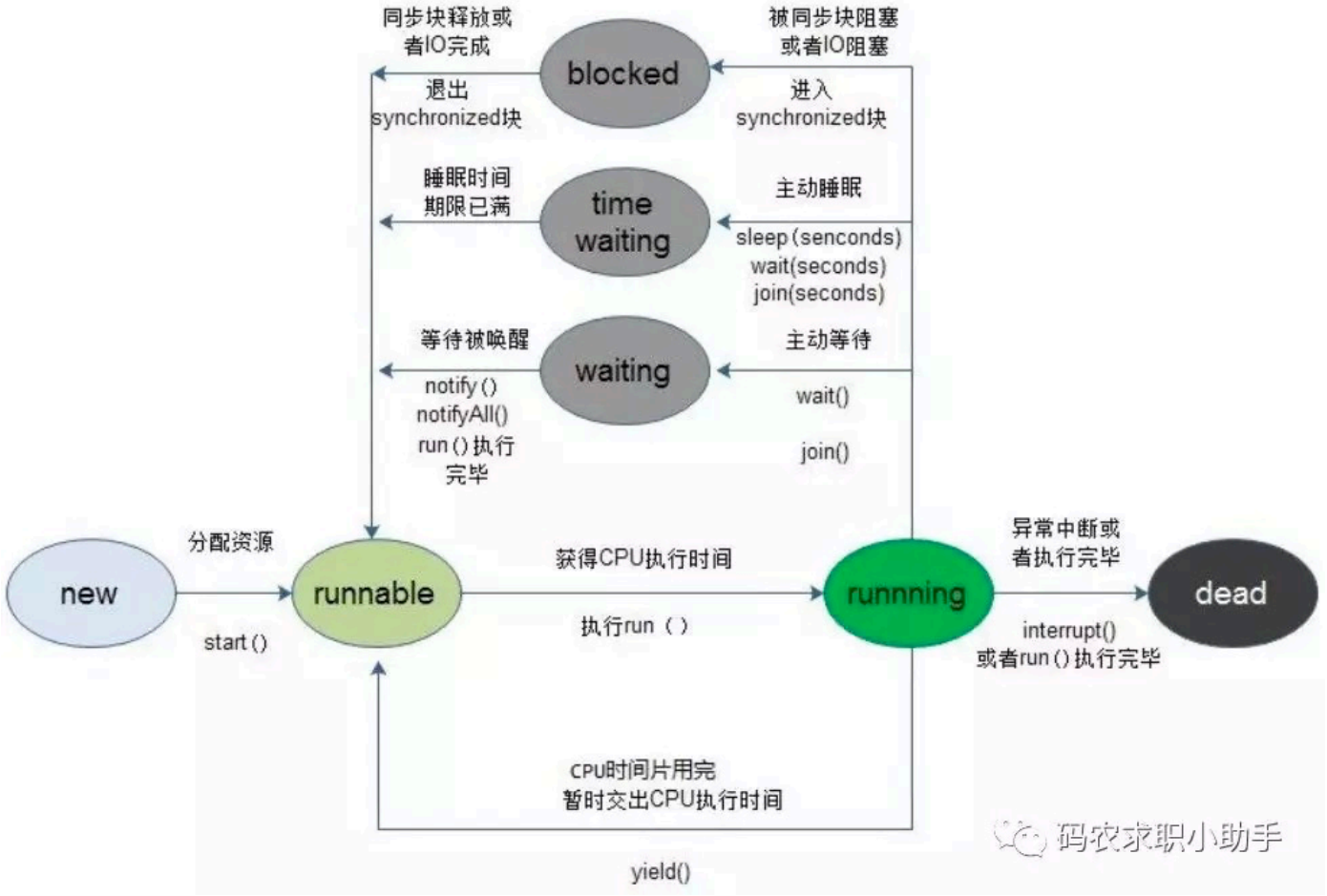
1. Runnable 接口中的 run() 方法的返回值是 void，它做的事情只是纯粹地去执行 run() 方法中的代码而已；
2. Callable 接口中的 call() 方法是有返回值的，是一个泛型，和 Future、FutureTask 配合可以用来获取异步执行的结果。

6、线程状态及转换?

Thread 的源码中定义了6种状态：new（新建）、runnnable（可运行）、blocked（阻塞）、waiting（等待）、time waiting （定时等待）和 terminated（终止）。

状态名称	说明
NEW	初始状态，线程被构建，但是还没有调用start()方法
RUNNABLE	运行状态，Java线程将操作系统中的就绪和运行两种状态笼统地称为"运行中"
BLOCKED	阻塞状态，表示线程阻塞与锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕

线程状态转换如下图所示：



7、sleep() 和 wait() 的区别?

1、sleep() 方法正在执行的线程主动让出 cpu（然后 cpu 就可以去执行其他任务），在 sleep 指定时间后 cpu 再回到该线程继续往下执行（注意：sleep 方法只让出了 cpu，而并不会释放同步资源锁）；而 wait() 方法则是指当前线程让自己暂时退让出同步资源锁，以便其他正在等待该资源的线程得到该资源进而运行，只有调用了 notify() 方法，之前调用 wait() 的线程才会解除 wait 状态，可以去参与竞争同步资源锁，进而得到执行。（注意：notify 的作用相当于叫醒睡着的人，而并不会给他分配任务，就是说 notify 只是让之前调用 wait 的线程有权利重新参与线程的调度）；

2、 sleep() 方法可以在任何地方使用，而 wait() 方法则只能在同步方法或同步块中使用；

3、 sleep() 是线程类（Thread）的方法，调用会暂停此线程指定的时间，但监控依然保持，不会释放对象锁，到时间自动恢复；wait() 是 Object 的方法，调用会放弃对象锁，进入等待队列，待调用 notify()/notifyAll() 唤醒指定的线程或者所有线程，才会进入锁池，不再次获得对象锁才会进入运行状态。

8、线程的 run() 和 start() 有什么区别?

1、每个线程都是通过某个特定 Thread 对象所对应的方法 run() 来完成其操作的，方法 run() 称为线程体。通过调用 Thread 类的 start() 方法来启动一个线程；

2、start() 方法来启动一个线程，真正实现了多线程运行。这时无需等待 run() 方法体代码执行完毕，可以直接继续执行下面的代码；这时此线程是处于就绪状态，并没有运行。然后通过此 Thread 类调用方法 run() 来完成其运行状态，这里方法 run() 称为线程体，它包含了要执行的这个线程的内容，run() 方法运行结束，此线程终止。然后 cpu 再调度其它线程；

3、 run() 方法是在本线程里的，只是线程里的一个函数，而不是多线程的。如果直接调用 run()，其实就相当于是调用了一个普通函数而已，直接待用 run() 方法必须等待 run() 方法执行完毕才能执行下面的代码，所以执行路径还是只有一条，根本就没有线程的特征，所以在多线程执行时要使用 start() 方法而不是 run() 方法。

9、在 Java 程序中怎么保证多线程的运行安全?

线程安全在三个方面体现：

原子性：提供互斥访问，同一时刻只能有一个线程对数据进行操作，（atomic，synchronized）；

可见性：一个线程对主内存的修改可以及时地被其他线程看到，（synchronized、volatile）；

有序性：一个线程观察其他线程中的指令执行顺序，由于指令重排序，该观察结果一般杂乱无序，（happens-before 原则）。

10、Java 线程同步的几种方法?

1. 使用 Synchronized 关键字；
2. wait 和 notify；
3. 使用特殊域变量 volatile 实现线程同步；
4. 使用可重入锁实现线程同步；
5. 使用阻塞队列实现线程同步；
6. 使用信号量 Semaphore。

11、Thread.interrupt() 方法的工作原理是什么?

在 Java 中，线程的中断 interrupt 只是改变了线程的中断状态，至于这个中断状态改变后带来的结果，那是无法确定的，有时它更是让停止中的线程继续执行的唯一手段。不但不是让线程停止运行，反而是继续执行线程的手段。

在一个线程对象上调用 interrupt() 方法，真正有影响的是 wait、join、sleep 方法，当然这 3 个方法包括它们的重载方法。请注意：上面这三个方法都会抛出 InterruptedException。

1、对于 wait 中的等待 notify、notifyAll 唤醒的线程，其实这个线程已经“暂停”执行，因为它正在某一对象的休息室中，这时如果它的中断状态被改变，那么它就会抛出异常。这个 InterruptedException 异常不是线程抛出的，而是 wait 方法，也就是对象的 wait 方法内部会不断检查在此对象上休息的线程的状态，如果发现哪个线程的状态被置为已中断，则会抛出 InterruptedException，意思就是这个线程不能再等待了，其意义就等同于唤醒它了，然后执行 catch 中的代码。

2、 对于 sleep 中的线程，如果你调用了 Thread.sleep(一年)；现在你后悔了，想让它早些醒过来，调用 interrupt() 方法就是唯一手段，只有改变它的中断状态，让它从 sleep 中将控制权转到处理异常的 catch 语句中，然后再由 catch 中的处理转换到正常的逻辑。同样，对于 join 中的线程你也可以这样处理。

12、谈谈对 ThreadLocal 的理解?

1、Java 的 Web 项目大部分都是基于 Tomcat。每次访问都是一个新的线程，每一个线程都独享一个 ThreadLocal，我们可以在接收请求的时候 set 特定内容，在需要的时候 get 这个值。

2、 ThreadLocal 提供 get 和 set 方法，为每一个使用这个变量的线程都保存有一份独立的副本。

```
public T get() {...}
public void set(T value) {...}
public void remove() {...}
protected T initialValue() {...}
```

1、get() 方法是用来获取 ThreadLocal 在当前线程中保存的变量副本；

2、set() 用来设置当前线程中变量的副本；

3、 remove() 用来移除当前线程中变量的副本；

4、 initialValue() 是一个 protected 方法，一般是用来在使用时进行重写的，如果在没有 set 的时候就调用 get，会调用 initialValue 方法初始化内容。

13、在哪些场景下会使用到 ThreadLocal?

在调用 API 接口的时候传递了一些公共参数，这些公共参数携带了一些设备信息（是安卓还是 ios），服务端接口根据不同的信息组装不同的格式数据返回给客户端。假定服务器端需要通过设备类型（device）来下发下载地址，当然接口也有同样的其他逻辑，我们只要在返回数据的时候判断好是什么类型的客户端就好了。上面这种场景就可以将传进来的参数 device 设置到 ThreadLocal 中。用的时候取出来就行。避免了参数的层层传递。

14、说一说自己对于 synchronized 关键字的了解?

synchronized关键字解决的是多个线程之间访问资源的同步性，synchronized 关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

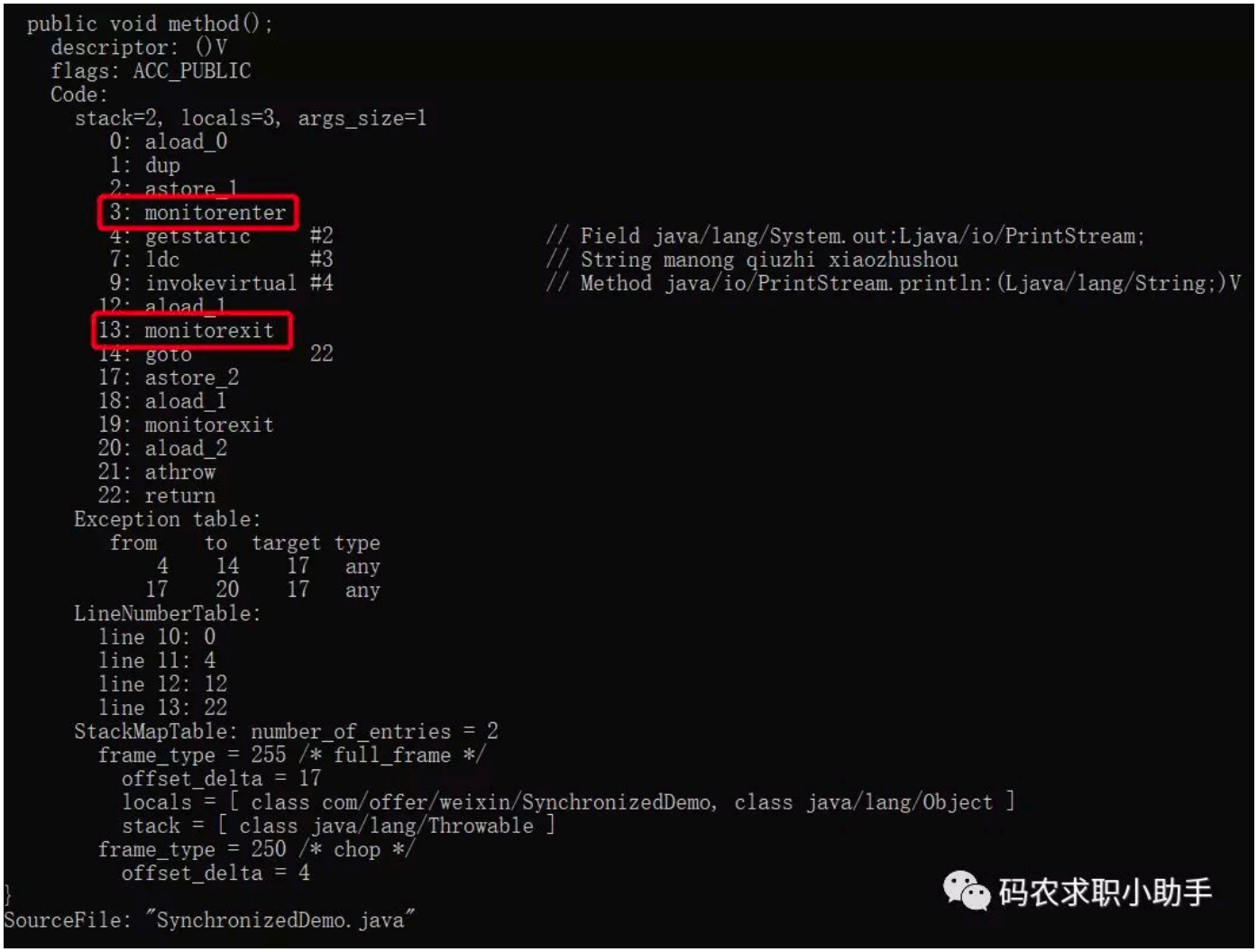
另外，在 Java 早期版本中，synchronized 属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的 synchronized 效率低的原因。庆幸的是在 JDK6 之后 Java 官方对从 JVM 层面对synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK6 对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

synchronized 关键字底层原理属于 JVM 层面。

synchronized 同步语句块的情况

```
public class SynchronizedDemo {
  public void method(){
    synchronized(this){
      System.out.println("manong qiuzhi xiaozhushou");
      }
    }
  }
```

通过 JDK 自带的 javap 命令查看 SynchronizedDemo 类的相关字节码信息：首先切换到类的对应目录执行 javac SynchronizedDemo.java 命令生成编译后的 .class 文件，然后执行 javap -c -s -v -l SynchronizedDemo.class



从上面我们可以看出：synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置。

当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor的持有权。monitor 对象存在于每个 Java 对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么 Java 中任意对象可以作为锁的原因。当计数器为 0 则可以成功获取，获取后将锁计数器设为 1 也就是加 1。相应的在执行 monitorexit 指令后，将锁计数器设为 0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

synchronized 修饰方法的的情况

```
public class SynchronizedDemo2 {
    public synchronized void method() {
        System.out.println("manong qiuzhi xiaozhushou");
    }
}
```

synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的确实是 ACC_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法，JVM 通过该 ACC_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用

16、如何在项目中使用 synchronized 的?

synchronized 关键字最主要的三种使用方式：

1、 修饰实例方法：作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁；

2、修饰静态方法：作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁 。也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（static 表明这是该类的一个静态资源，不管 new了多少个对象，只有一份，所以对该类的所有对象都加了锁）。所以如果一个线程 A 调用一个实例对象的非静态 synchronized 方法，而线程 B 需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁；

3、 修饰代码块：指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。和 synchronized 方法一样，synchronized(this) 代码块也是锁定当前对象的。synchronized 关键字加到 static 静态方法和 synchronized(class) 代码块上都是是给 Class 类上锁。这里再提一下：synchronized 关键字加到非 static 静态方法上是给对象实例上锁。另外需要注意的是：尽量不要使用 synchronized(String a) 因为 JVM 中，字符串常量池具有缓冲功能。

补充：双重校验锁实现单例模式

问到 synchronized 的使用，很有可能让你用 synchronized 实现个单例模式。这里补充下使用 synchronized 双重校验锁的方法实现单例模式：

```
public class Singleton {

    private volatile static Singleton uniqueInstance;
    private Singleton() {}

    public static Singleton getUniqueInstance() {
       // 先判断对象是否已经实例过，没有实例化过才进入加锁代码
        if (uniqueInstance == null) {
            // 类对象加锁
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

另外，需要注意 uniqueInstance 采用 volatile 关键字修饰也是很有必要。采用 volatile 关键字修饰也是很有必要的， uniqueInstance = new Singleton(); 这段代码其实是分为三步执行：

1. 为 uniqueInstance 分配内存空间
2. 初始化 uniqueInstance
3. 将 uniqueInstance 指向分配的内存地址

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1 -> 3 -> 2。指令重排在单线程环境下不会出现问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 getUniqueInstance() 后发现 uniqueInstance 不为空，因此返回 uniqueInstance，但此时 uniqueInstance 还未被初始化。

使用 volatile 可以禁止 JVM 的指令重排，保证在多线程环境下也能正常运行。

17、说说 JDK1.6 之后的 synchronized 关键字底层做了哪些优化，可以详细介绍一下这些优化吗?

说明：这道题答案有点长，但是回答的详细面试会很加分。

JDK1.6 对锁的实现引入了大量的优化，如偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化等技术来减少锁操作的开销。

锁主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，它们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

- **偏向锁**

引入偏向锁的目的和引入轻量级锁的目的很像，它们都是为了没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。但是不同是：轻量级锁在无竞争的情况下使用 CAS 操作去代替使用互斥量。而偏向锁在无竞争的情况下会把整个同步都消除掉。

偏向锁的“偏”就是偏心的偏，它的意思是会偏向于第一个获得它的线程，如果在接下来的执行中，该锁没有被其他线程获取，那么持有偏向锁的线程就不需要进行同步。

但是对于锁竞争比较激烈的场合，偏向锁就失效了，因为这样场合极有可能每次申请锁的线程都是不相同的，因此这种场合下不应该使用偏向锁，否则会得不偿失，需要注意的是，偏向锁失败后，并不会立即膨胀为重量级锁，而是先升级为轻量级锁。

- **轻量级锁**

倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段(JDK1.6 之后加入的)。轻量级锁不是为了代替重量级锁，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗，因为使用轻量级锁时，不需要申请互斥量。另外，轻量级锁的加锁和解锁都用到了 CAS 操作。

轻量级锁能够提升程序同步性能的依据是“对于绝大部分锁，在整个同步周期内都是不存在竞争的”，这是一个经验数据。如果没有竞争，轻量级锁使用 CAS 操作避免了使用互斥操作的开销。但如果存在锁竞争，除了互斥量开销外，还会额外发生 CAS 操作，因此在有锁竞争的情况下，轻量级锁比传统的重量级锁更慢！如果锁竞争激烈，那么轻量级将很快膨胀为重量级锁！

- **自旋锁和自适应自旋**

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。

互斥同步对性能最大的影响就是阻塞的实现，因为挂起线程/恢复线程的操作都需要转入内核态中完成（用户态转换到内核态会耗费时间）。

一般线程持有锁的时间都不是太长，所以仅仅为了这一点时间去挂起线程/恢复线程是得不偿失的。所以，虚拟机的开发团队就这样去考虑：“我们能不能让后面来的请求获取锁的线程等待一会而不被挂起呢？看看持有锁的线程是否很快就会释放锁”。为了让一个线程等待，我们只需要让线程执行一个忙循环（自旋），这项技术就叫做自旋

> **百度百科对自旋锁的解释：**何谓自旋锁？它是为实现保护共享资源而提出一种锁机制。其实，自旋锁与互斥锁比较类似，它们都是为了解决对某项资源的互斥使用。无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名

自旋锁在 JDK1.6 之前其实就已经引入了，不过是默认关闭的，需要通过 --XX:+UseSpinning 参数来开启。JDK1.6 及 1.6 之后，就改为默认开启的了。需要注意的是：自旋等待不能完全替代阻塞，因为它还是要占用处理器时间。如果锁被占用的时间短，那么效果当然就很好了。反之，自旋等待的时间必须要有限度。如果自旋超过了限定次数任然没有获得锁，就应该挂起线程。自旋次数的默认值是 10 次，用户可以修改 --XX:PreBlockSpin 来更改。

另外，在 JDK1.6 中引入了自适应的自旋锁。自适应的自旋锁带来的改进就是：自旋的时间不在固定了，而是和前一次同一个锁上的自旋时间以及锁的拥有者的状态来决定，虚拟机变得越来越“聪明”了。

- **锁消除**

锁消除理解起来很简单，它指的就是虚拟机即使编译器在运行时，如果检测到那些共享数据不可能存在竞争，那么就执行锁消除。锁消除可以节省毫无意义的请求锁的时间。

- **锁粗化**

原则上，我们在编写代码的时候，总是推荐将同步块的作用范围限制得尽量小。只在共享数据的实际作用域才进行同步，这样是为了使得需要同步的操作数量尽可能变小，如果存在锁竞争，那等待线程也能尽快拿到锁。

大部分情况下，上面的原则都是没有问题的，但是如果一系列的连续操作都对同一个对象反复加锁和解锁，那么会带来很多不必要的性能消耗

18、谈谈 synchronized 和 ReenTrantLock 的区别?

1、synchronized 是和 for、while 一样的关键字，ReentrantLock 是类，这是二者的本质区别。既然 ReentrantLock 是类，那么它就提供了比 synchronized 更多更灵活的特性：等待可中断、可实现公平锁、可实现选择性通知（锁可以绑定多个条件）、性能已不是选择标准。

2、 synchronized 依赖于 JVM 而 ReenTrantLock 依赖于 API。synchronized 是依赖于 JVM 实现的，JDK1.6 为 synchronized 关键字进行了很多优化，但是这些优化都是在虚拟机层面实现的，并没有直接暴露给我们。ReenTrantLock 是 JDK 层面实现的（也就是 API 层面，需要 lock() 和 unlock 方法配合 try/finally 语句块来完成），所以我们可以通过查看它的源代码，来看它是如何实现的。

19、synchronized 和 volatile 的区别是什么?

1. volatile 本质是在告诉 JVM当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取；synchronized 则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。
2. volatile 仅能使用在变量级别；synchronized 则可以使用在变量、方法、和类级别的。
3. volatile 仅能实现变量的修改可见性，不能保证原子性；而 synchronized 则可以保证变量的修改可见性和原子性。
4. volatile 不会造成线程的阻塞；synchronized 可能会造成线程的阻塞。
5. volatile 标记的变量不会被编译器优化；synchronized 标记的变量可以被编译器优化。

20、谈一下你对 volatile 关键字的理解?

volatile 关键字是用来保证有序性和可见性的。这跟 Java 内存模型有关。我们所写的代码，不一定是按照我们自己书写的顺序来执行的，编译器会做重排序，CPU 也会做重排序的，这样做是为了减少流水线阻塞，提高 CPU 的执行效率。这就需要有一定的顺序和规则来保证，不然程序员自己写的代码都不知道对不对了，所以有 happens-before 规则，其中有条就是 volatile 变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作、有序性实现的是通过插入内存屏障来保证的。

被 volatile 修饰的共享变量，就具有了以下两点特性：

1 . 保证了不同线程对该变量操作的内存可见性;

2 . 禁止指令重排序。

备注：这个题如果扩展了答，可以从 Java 的内存模型入手，下一篇 Java 虚拟机高频面试题中会讲到，这里不做过多赘述。

21、说下对 ReentrantReadWriteLock 的理解?

ReentrantReadWriteLock 允许多个读线程同时访问，但是不允许写线程和读线程、写线程和写线程同时访问。读写锁内部维护了两个锁：一个是用于读操作的 ReadLock，一个是用于写操作的 WriteLock。读写锁 ReentrantReadWriteLock 可以保证多个线程可以同时读，所以在读操作远大于写操作的时候，读写锁就非常有用了。

ReentrantReadWriteLock 基于 AQS 实现，它的自定义同步器（继承 AQS）需要在同步状态 state 上维护多个读线程和一个写线程，该状态的设计成为实现读写锁的关键。ReentrantReadWriteLock 很好的利用了高低位。来实现一个整型控制两种状态的功能，读写锁将变量切分成了两个部分，高 16 位表示读，低 16 位表示写。

- **ReentrantReadWriteLock 的特点：**

1、写锁可以降级为读锁，但是读锁不能升级为写锁；

2、 不管是 ReadLock 还是 WriteLock 都支持 Interrupt，语义与 ReentrantLock 一致；

3、WriteLock 支持 Condition 并且与 ReentrantLock 语义一致，而 ReadLock 则不能使用 Condition，否则抛出 UnsupportedOperationException 异常；

4、 默认构造方法为非公平模式 ，开发者也可以通过指定 fair 为 true 设置为公平模式 。

- **升降级**

1、 读锁里面加写锁，会导致死锁；

\2. 写锁里面是可以加读锁的，这就是锁的降级。

22、说下对悲观锁和乐观锁的理解?

- **悲观锁**

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。传统的关系型数据库里边就用到了很多这种锁机制，比如：行锁、表锁、读锁、写锁等，都是在做操作之前先上锁。Java 中 synchronized 和 ReentrantLock 等独占锁就是悲观锁思想的实现。

- **乐观锁**

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和 CAS 算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 write_condition 机制，其实都是提供的乐观锁。在 Java 中 java.util.concurrent.atomic 包下面的原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。

- **两种锁的使用场景**

从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行 retry，这样反倒是降低了性能，所以一般多写的场景下用悲观锁就比较合适。

23、乐观锁常见的两种实现方式是什么?

乐观锁一般会使用版本号机制或者 CAS 算法实现。

- **版本号机制**

  一般是在数据表中加上一个数据版本号 version 字段，表示数据被修改的次数，当数据被修改时，version 值会加 1。当线程 A 要更新数据值时，在读取数据的同时也会读取 version 值，在提交更新时，若刚才读取到的 version 值为当前数据库中的 version 值相等时才更新，否则重试更新操作，直到更新成功。

- **CAS 算法**

  即 compare and swap（比较与交换），是一种有名的无锁算法。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。CAS 算法涉及到三个操作数：

1、需要读写的内存值 V

2、进行比较的值 A

3、拟写入的新值 B

当且仅当 V 的值等于 A 时，CAS 通过原子方式用新值 B 来更新 V 的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个自旋操作，即不断的重试。

24、乐观锁的缺点有哪些?

- **1. ABA 问题**

如果一个变量 V 初次读取的时候是 A 值，并且在准备赋值的时候检查到它仍然是 A 值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回 A，那 CAS 操作就会误认为它从来没有被修改过。这个问题被称为 CAS 操作的 "ABA" 问题。

JDK 1.5 以后的AtomicStampedReference 类就提供了此种能力，其中的 compareAndSet 方法就是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

- **2. 循环时间长开销大**

自旋 CAS（也就是不成功就一直循环执行直到成功）如果长时间不成功，会给 CPU 带来非常大的执行开销。如果 JVM 能支持处理器提供的 pause 指令那么效率会有一定的提升，pause 指令有两个作用，第一：它可以延迟流水线执行指令（de-pipeline），使 CPU 不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二：它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起 CPU

流水线被清空（CPU pipeline flush），从而提高 CPU 的执行效率。

- **3. 只能保证一个共享变量的原子操作**

CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。 但是从 JDK 1.5 开始，提供了 AtomicReference 类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作。所以我们可以使用锁或者利用 AtomicReference 类把多个共享变量合并成一个共享变量来操作。

25、CAS 和 synchronized 的使用场景?

简单的来说 CAS 适用于写比较少的情况下（多读场景，冲突一般较少），synchronized 适用于写比较多的情况下（多写场景，冲突一般较多）。

1、对于资源竞争较少（线程冲突较轻）的情况，使用 synchronized 同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗 cpu 资源；而 CAS 基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。

2、对于资源竞争严重（线程冲突严重）的情况，CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 synchronized。

26、简单说下对 Java 中的原子类的理解?

这里 Atomic 是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰。所以，所谓原子类说简单点就是具有原子操作特征的类。

并发包 java.util.concurrent 的原子类都存放在 java.util.concurrent.atomic 下。根据操作的数据类型，可以将 JUC 包中的原子类分为 4 类：

- **1. 基本类型**

使用原子的方式更新基本类型：

AtomicInteger：整型原子类

AtomicLong：长整型原子类

AtomicBoolean ： 布尔型原子类

- **2. 数组类型**

使用原子的方式更新数组里的某个元素：

AtomicIntegerArray：整型数组原子类

AtomicLongArray：长整型数组原子类

AtomicReferenceArray ： 引用类型数组原子类

- **3. 引用类型**

AtomicReference：引用类型原子类

AtomicStampedReference：原子更新引用类型里的字段原子类

AtomicMarkableReference ： 原子更新带有标记位的引用类型

- **4. 对象的属性修改类型**

AtomicIntegerFieldUpdater：原子更新整型字段的更新器

AtomicLongFieldUpdater：原子更新长整型字段的更新器

AtomicStampedReference ：原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于解决原子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。

27、atomic 的原理是什么?

Atomic 包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。

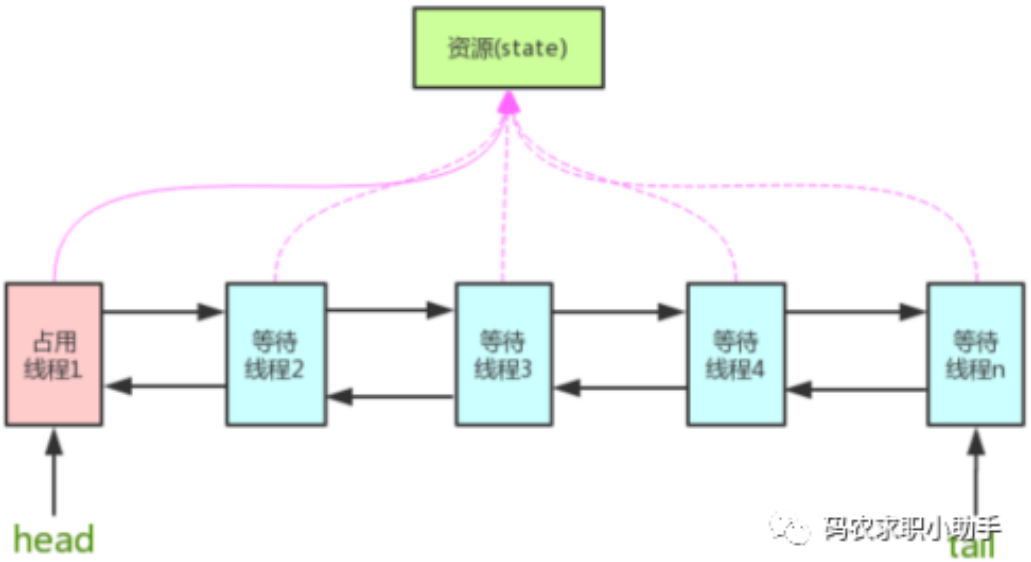
Atomic 系列的类中的核心方法都会调用 unsafe 类中的几个本地方法。我们需要先知道一个东西就是 Unsafe 类，全名为：sun.misc.Unsafe，这个类包含了大量的对 C 代码的操作，包括很多直接内存分配以及原子操作的调用，而它之所以标记为非安全的，是告诉你这个里面大量的方法调用都会存在安全隐患，需要小心使用，否则会导致严重的后果，例如在通过 unsafe 分配内存的时候，如果自己指定某些区域可能会导致一些类似 C++ 一样的指针越界到其他进程的问题。

28、说下对同步器 AQS 的理解?

AQS 的全称为：AbstractQueuedSynchronizer，这个类在 java.util.concurrent.locks 包下面。AQS 是一个用来构建锁和同步器的框架，使用 AQS 能简单且高效地构造出应用广泛的大量的同步器，比如：我们提到的 ReentrantLock，Semaphore，其他的诸如ReentrantReadWriteLock，SynchronousQueue，FutureTask 等等皆是基于 AQS 的。当然，我们自己也能利用 AQS 非常轻松容易地构造出符合我们自己需求的同步器。

29、AQS 的原理是什么?

AQS 核心思想是：如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制 AQS 是用 CLH 队列锁实现的，即将暂时获取不到锁的线程加入到队列中。



> CLH队列：CLH(Craig, Landin, and Hagersten)队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）。AQS 是将每条请求共享资源的线程封装成一个 CLH 锁队列的一个结点（Node）来实现锁的分配。

AQS 使用一个 int 成员变量 (state) 来表示同步状态，通过内置的 FIFO 队列来完成获取资源线程的排队工作。AQS 使用 CAS 对该同步状态进行原子操作实现对其值的修改。

```
// 共享变量，使用 volatile 修饰保证线程可见性
private volatile int state;
```

30、AQS 对资源的共享模式有哪些?

1. Exclusive（独占）：只有一个线程能执行，如：ReentrantLock，又可分为公平锁和非公平锁：
2. Share（共享）：多个线程可同时执行，如：CountDownLatch、Semaphore、CountDownLatch、CyclicBarrier、ReadWriteLock。

31、AQS 底层使用了模板方法模式，你能说出几个需要重写的方法吗?

使用者继承 AbstractQueuedSynchronizer 并重写指定的方法。将 AQS 组合在自定义同步组件的实现中，并调用其模板方法，而这些模板方法会调用使用者重写的方法。

1. isHeldExclusively()：该线程是否正在独占资源。只有用到 condition 才需要去实现它。
2. tryAcquire(int)：独占方式。尝试获取资源，成功则返回 true，失败则返回 false。
3. tryRelease(int)：独占方式。尝试释放资源，成功则返回 true，失败则返回 false。
4. tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0 表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
5. tryReleaseShared(int)：共享方式。尝试释放资源，成功则返回 true，失败则返回 false。

32、说下对信号量 Semaphore 的理解?

synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，Semaphore (信号量)可以指定多个线程同时访问某个资源。

执行 acquire 方法阻塞，直到有一个许可证可以获得然后拿走一个许可证；每个 release 方法增加一个许可证，这可能会释放一个阻塞的 acquire 方法。然而，其实并没有实际的许可证这个对象，Semaphore 只是维持了一个可获得许可证的数量。Semaphore 经常用于限制获取某种资源的线程数量。当然一次也可以一次拿取和释放多个许可证，不过一般没有必要这样做。除了 acquire方法（阻塞）之外，另一个比较常用的与之对应的方法是 tryAcquire 方法，该方法如果获取不到许可就立即返回 false。

33、CountDownLatch 和 CyclicBarrier 有什么区别?

CountDownLatch 是计数器，只能使用一次，而 CyclicBarrier 的计数器提供 reset 功能，可以多次使用。

对于 CountDownLatch 来说，重点是“一个线程（多个线程）等待”，而其他的 N 个线程在完成“某件事情”之后，可以终止，也可以等待。而对于 CyclicBarrier，重点是多个线程，在任意一个线程没有完成，所有的线程都必须等待。

CountDownLatch 是计数器，线程完成一个记录一个，只不过计数不是递增而是递减，而 CyclicBarrier 更像是一个阀门，需要所有线程都到达，阀门才能打开，然后继续执行。

CountDownLatch 应用场景：

1、某一线程在开始运行前等待 n 个线程执行完毕：启动一个服务时，主线程需要等待多个组件加载完毕，之后再继续执行。

2、实现多个线程开始执行任务的最大并行性。注意是并行性，不是并发，强调的是多个线程在某一时刻同时开始执行。类似于赛跑，将多个线程放到起点，等待发令枪响，然后同时开跑。

3、死锁检测：一个非常方便的使用场景是，你可以使用 n 个线程访问共享资源，在每次测试阶段的线程数目是不同的，并尝试产生死锁。

CyclicBarrier 应用场景：

CyclicBarrier 可以用于多线程计算数据，最后合并计算结果的应用场景。比如：我们用一个 Excel 保存了用户所有银行流水，每个 Sheet 保存一个帐户近一年的每笔银行流水，现在需要统计用户的日均银行流水，先用多线程处理每个 sheet 里的银行流水，都执行完之后，得到每个 sheet 的日均银行流水，最后，再用 barrierAction 用这些线程的计算结果，计算出整个 Excel 的日均银行流水。

34、说下对线程池的理解？为什么要使用线程池?

线程池提供了一种限制和管理资源（包括执行一个任务）的方式。每个线程池还维护一些基本统计信息，例如：已完成任务的数量。

- **使用线程池的好处**

1、降低资源消耗：通过重复利用已创建的线程降低线程创建和销毁造成的消耗;

2、 提高响应速度：当任务到达时，任务可以不需要的等到线程创建就能立即执行;

3、 提高线程的可管理性：线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

35、创建线程池的参数有哪些?

1、 corePoolSize（线程池的基本大小）：当提交一个任务到线程池时，如果当前 poolSize < corePoolSize 时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。如果调用了线程池的prestartAllCoreThreads() 方法，线程池会提前创建并启动所有基本线程。

2、 maximumPoolSize（线程池最大数量）：线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。值得注意的是，如果使用了无界的任务队列这个参数就没什么效果。

3、 keepAliveTime（线程活动保持时间）：线程池的工作线程空闲后，保持存活的时间。所以，如果任务很多，并且每个任务执行的时间比较短，可以调大时间，提高线程的利用率。

4、 TimeUnit（线程活动保持时间的单位）：可选的单位有天（DAYS）、小时（HOURS）、分钟（MINUTES）、毫秒（MILLISECONDS）、微秒（MICROSECONDS，千分之一毫秒）和纳秒（NANOSECONDS，千分之一微秒）。

\5. workQueue（任务队列）：用于保存等待执行的任务的阻塞队列。

可以选择以下几个阻塞队列：

1）、 ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。

2）、LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按 FIFO 排序元素，吞吐量通常要高于 ArrayBlockingQueue。静态工厂方法 Executors.newFixedThreadPool() 使用了这个队列。

3）、SynchronousQueue：一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于 LinkedBlockingQueue，静态工厂方法 Executors.newCachedThreadPool 使用了这个队列。

4）、 PriorityBlockingQueue：一个具有优先级的无限阻塞队列。

6、 threadFactory：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字。

\7. RejectExecutionHandler（饱和策略）：队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是 AbortPolicy，表示无法处理新任务时抛出异常。

饱和策略：

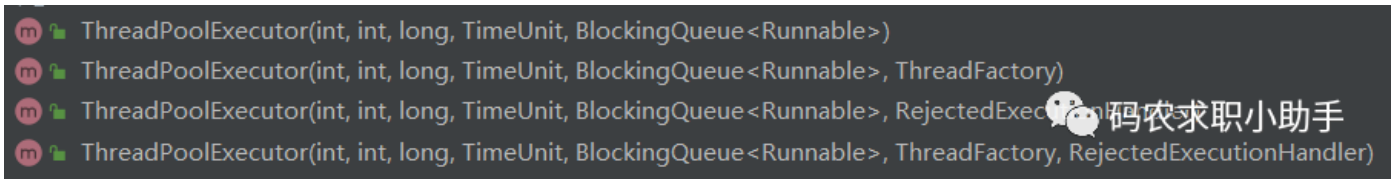
在 JDK1.5 中 Java 线程池框架提供了以下 4 种策略：

1. AbortPolicy：直接抛出异常。
2. CallerRunsPolicy：只用调用者所在线程来运行任务。
3. DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。
4. DiscardPolicy：不处理，丢弃掉。

当然，也可以根据应用场景需要来实现RejectedExecutionHandler 接口自定义策略。如记录日志或持久化存储不能处理的任务。

36、如何创建线程池?

方式一：通过 ThreadPoolExecutor 的构造方法实现：



方式二：通过 Executor 框架的工具类 Executors 来实现：

我们可以创建三种类型的 ThreadPoolExecutor：

1、FixedThreadPool：该方法返回一个固定线程数量的线程池。该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在一个任务队列中，待有线程空闲时，便处理在任务队列中的任务。

2、 SingleThreadExecutor：方法返回一个只有一个线程的线程池。若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先进先出的顺序执行队列中的任务。

3、CachedThreadPool：该方法返回一个可根据实际情况调整线程数量的线程池。线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕后，将返回线程池进行复用。

注意：

阿里巴巴Java开发手册》中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

Executors 创建线程池对象的弊端如下：

FixedThreadPool 和 SingleThreadExecutor： 允许请求的队列长度为 Integer.MAX_VALUE，可能堆积大量的请求，从而导致 OOM。CachedThreadPool 和 ScheduledThreadPool： 允许创建的线程数量为 Integer.MAX_VALUE ，可能会创建大量线程，从而导致 OOM。

37、线程池中的的线程数一般怎么设置？需要考虑哪些问题?

主要考虑下面几个方面：

- **1. 线程池中线程执行任务的性质：**

计算密集型的任务比较占 cpu，所以一般线程数设置的大小 等于或者略微大于 cpu 的核数；但 IO 型任务主要时间消耗在 IO 等待上，cpu 压力并不大，所以线程数一般设置较大。

- **2. cpu 使用率：**

当线程数设置较大时，会有如下几个问题：第一，线程的初始化，切换，销毁等操作会消耗不小的 cpu 资源，使得 cpu 利用率一直维持在较高水平。第二，线程数较大时，任务会短时间迅速执行，任务的集中执行也会给 cpu 造成较大的压力。第三， 任务的集中支持，会让 cpu 的使用率呈现锯齿状，即短时间内 cpu 飙高，然后迅速下降至闲置状态，cpu 使用的不合理，应该减小线程数，让任务在队列等待，使得 cpu 的使用率应该持续稳定在一个合理，平均的数值范围。所以 cpu 在够用时，不宜过大，不是越大越好。可以通过上线后，观察机器的 cpu 使用率和 cpu 负载两个参数来判断线程数是否合理。

- **3. 内存使用率：**

线程数过多和队列的大小都会影响此项数据，队列的大小应该通过前期计算线程池任务的条数，来合理的设置队列的大小，不宜过小，让其不会溢出，因为溢出会走拒绝策略，多少会影响性能，也会增加复杂度。

- **4. 下游系统抗并发能力：**

多线程给下游系统造成的并发等于你设置的线程数，例如：如果是多线程访问数据库，你就考虑数据库的连接池大小设置，数据库并发太多影响其 QPS，会把数据库打挂等问题。如果访问的是下游系统的接口，你就得考虑下游系统是否能抗的住这么多并发量，不能把下游系统打挂了。

38、执行 execute() 方法和 submit() 方法的区别是什么呢?

1、 execute() 方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功与否；

2、 submit() 方法用于提交需要返回值的任务。线程池会返回一个 Future 类型的对象，通过这个 Future 对象可以判断任务是否执行成功，并且可以通过 Future 的 get() 方法来获取返回值，get() 方法会阻塞当前线程直到任务完成，而使用 get(long timeout，TimeUnit unit) 方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

39、说下对 Fork/Join 并行计算框架的理解?

Fork/Join 并行计算框架主要解决的是分治任务。分治的核心思想是“分而治之”：将一个大的任务拆分成小的子任务的结果聚合起来从而得到最终结果。

Fork/Join 并行计算框架的核心组件是 ForkJoinPool。ForkJoinPool 支持任务窃取机制，能够让所有的线程的工作量基本均衡，不会出现有的线程很忙，而有的线程很闲的情况，所以性能很好。

ForkJoinPool 中的任务队列采用的是双端队列，工作线程正常获取任务和“窃取任务”分别是从任务队列不同的端消费，这样能避免很多不必要的数据竞争。

40、JDK 中提供了哪些并发容器?

JDK 提供的这些容器大部分在 java.util.concurrent 包中。

1. ConcurrentHashMap：线程安全的 HashMap；
2. CopyOnWriteArrayList：线程安全的 List，在读多写少的场合性能非常好，远远好于 Vector；
3. ConcurrentLinkedQueue：高效的并发队列，使用链表实现。可以看做一个线程安全的 LinkedList，这是一个非阻塞队列；
4. BlockingQueue：这是一个接口，JDK 内部通过链表、数组等方式实现了这个接口。表示阻塞队列，非常适合用于作为数据共享的通道；
5. ConcurrentSkipListMap：跳表的实现。这是一个 Map，使用跳表的数据结构进行快速查找。

41、谈谈对 CopyOnWriteArrayList 的理解?

在很多应用场景中，读操作可能会远远大于写操作。由于读操作根本不会修改原有的数据，因此对于每次读取都进行加锁其实是一种资源浪费。我们应该允许多个线程同时访问 List 的内部数据，毕竟读取操作是安全的。

CopyOnWriteArrayList 类的所有可变操作（add，set 等等）都是通过创建底层数组的新副本来实现的。当 List 需要被修改的时候，我们并不需要修改原有内容，而是对原有数据进行一次复制，将修改的内容写入副本。写完之后，再将修改完的副本替换原来的数据，这样就可以保证写操作不会影响读操作了。

从 CopyOnWriteArrayList 的名字就能看出 CopyOnWriteArrayList 是满足 CopyOnWrite 的 ArrayList，所谓 CopyOnWrite 也就是说：在计算机，如果你想要对一块内存进行修改时，我们不在原有内存块中进行写操作，而是将内存拷贝一份，在新的内存中进行写操作，写完之后，就将指向原来内存指针指向新的内存，原来的内存就可以被回收掉了。

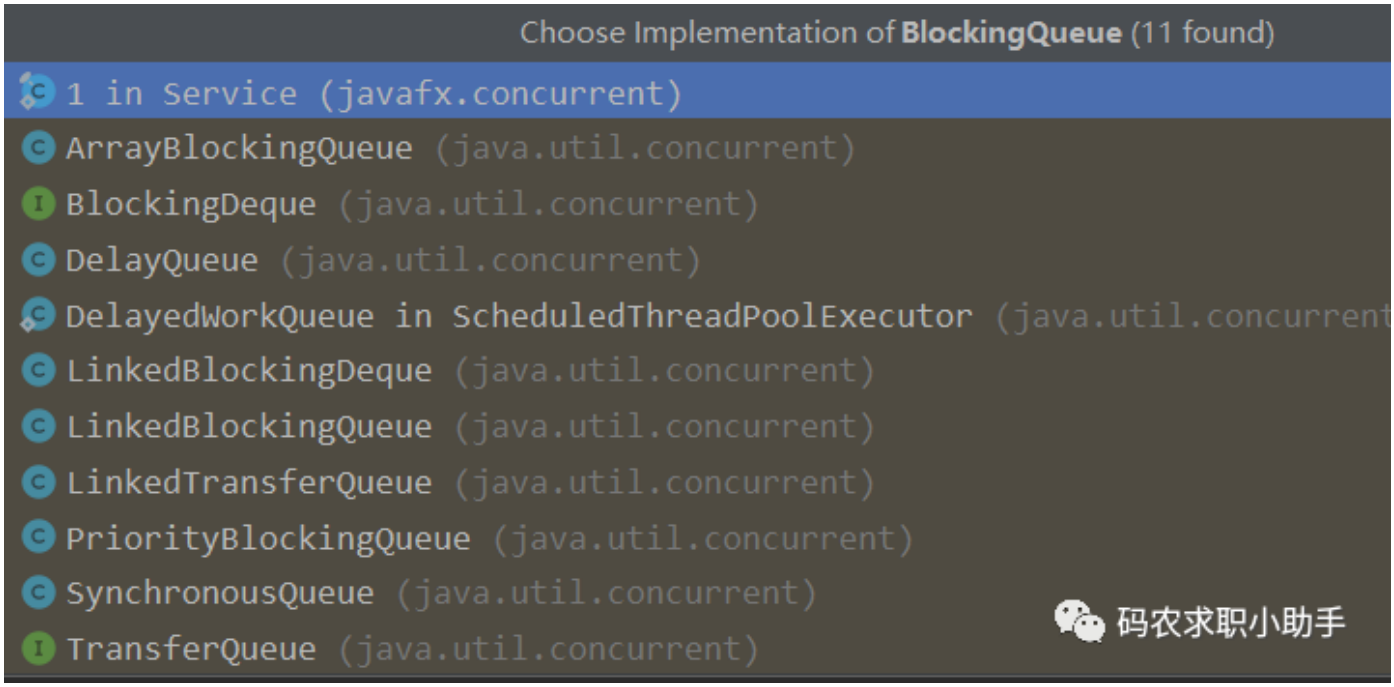
CopyOnWriteArrayList 读取操作没有任何同步控制和锁操作，理由就是内部数组 array 不会发生修改，只会被另外一个 array 替换，因此可以保证数据安全。

CopyOnWriteArrayList 写入操作 add() 方法在添加集合的时候加了锁，保证了同步，避免了多线程写的时候会 copy 出多个副本出来。

42、谈谈对 BlockingQueue 的理解？分别有哪些实现类?

阻塞队列（BlockingQueue）被广泛使用在“生产者-消费者”问题中，其原因是 BlockingQueue 提供了可阻塞的插入和移除的方法。当队列容器已满，生产者线程会被阻塞，直到队列未满；当队列容器为空时，消费者线程会被阻塞，直至队列非空时为止。

BlockingQueue 是一个接口，继承自 Queue，所以其实现类也可以作为 Queue 的实现来使用，而 Queue 又继承自 Collection 接口。下面是 BlockingQueue 的相关实现类：



43、谈谈对 ConcurrentSkipListMap 的理解?

对于一个单链表，即使链表是有序的，如果我们想要在其中查找某个数据，也只能从头到尾遍历链表，这样效率自然就会很低，跳表就不一样了。跳表是一种可以用来快速查找的数据结构，有点类似于平衡树。它们都可以对元素进行快速的查找。

但一个重要的区别是：对平衡树的插入和删除往往很可能导致平衡树进行一次全局的调整。而对跳表的插入和删除只需要对整个数据结构的局部进行操作即可。这样带来的好处是：在高并发的情况下，你会需要一个全局锁来保证整个平衡树的线程安全。而对于跳表，你只需要部分锁即可。这样，在高并发环境下，你就可以拥有更好的性能。而就查询的性能而言，跳表的时间复杂度也是 O(logn) 。跳表的本质是同时维护了多个链表，并且链表是分层的。

Java虚拟机

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**Java面试题**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**Java面试题**」，即可获取最新版 PDF。

另外，也建议大家来网站读，因为如果有错误，我会及时在网站更改，PDF 很难及时更新。

在线网站：https://www.iamshuaidi.com

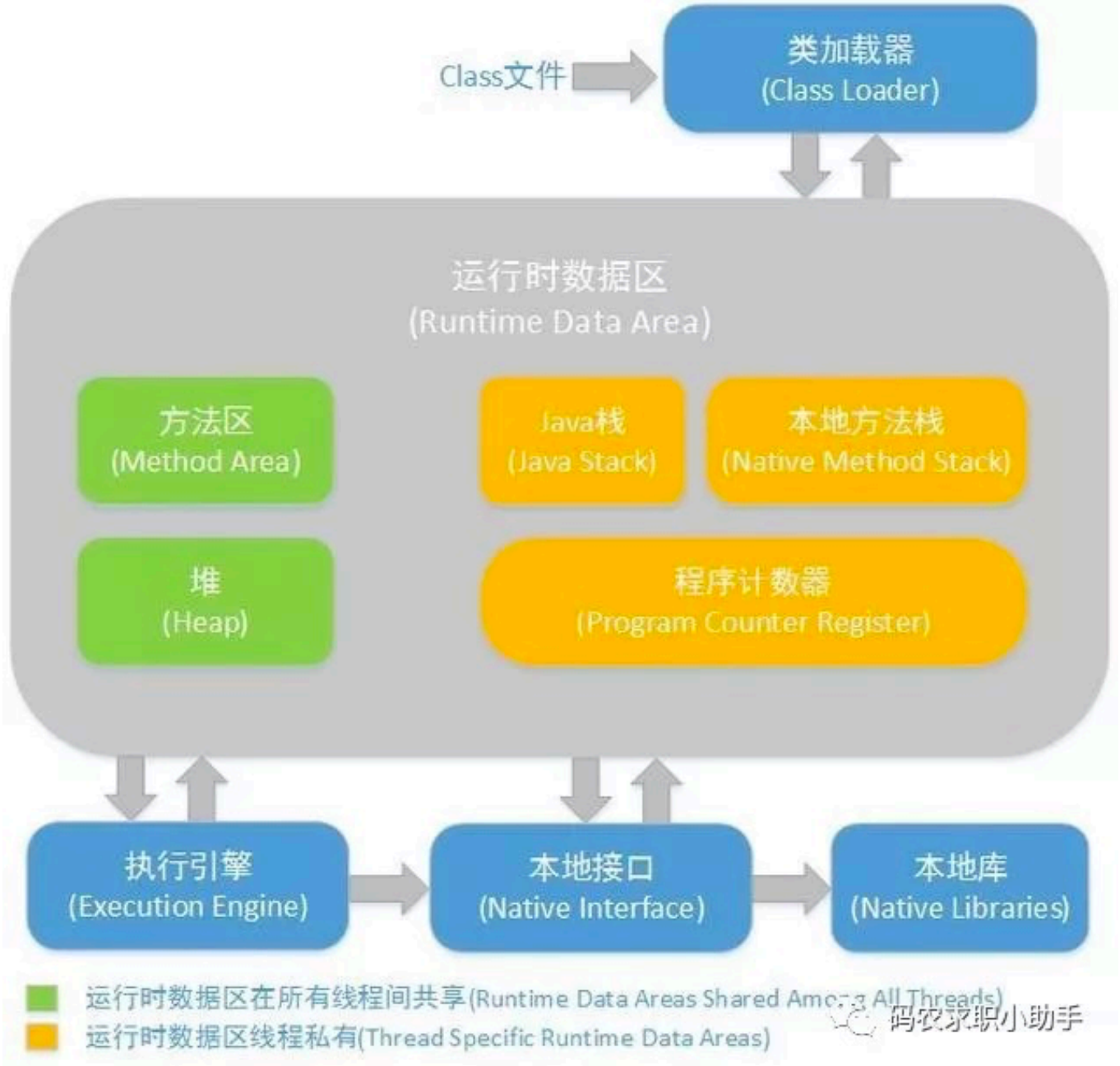
1、说一下 Jvm 的主要组成部分？及其作用?

1. 类加载器（ClassLoader）
2. 运行时数据区（Runtime Data Area）
3. 执行引擎（Execution Engine）
4. 本地库接口（Native Interface）

各组件的作用：首先通过类加载器（ClassLoader）会把 Java 代码转换成字节码，运行时数据区（Runtime Data Area）再把字节码加载到内存中，而字节码文件只是 JVM 的一套指令集规范，并不能直接交给底层操作系统去执行，因此需要特定的命令解析器执行引擎（Execution Engine），将字节码翻译成底层系统指令，再交由 CPU 去执行，而这个过程中需要调用其他语言的本地库接口（Native Interface）来实现整个程序的功能。

2、谈谈对运行时数据区的理解?

Tip：这道题是非常重要的题目，几乎问到 Java 虚拟机这块都是会被问到的。建议不要简单的只回答几个区域的名称，最好展开的讲解下，下面的答案是比较详细的，根据自己的理解回答其中某一段即可。



- **1. 程序计数器**

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

字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。程序的分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

由于 Java 虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器都只会执行一条线程中的命令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各线程之间的计数器互不影响，独立存储，我们程这块内存区域为“线程私有”的内存。

此区域是唯一 一个虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域。

- **2. Java 虚拟机栈**

Java 虚拟机栈（Java Virtual Machine Stacks）：描述的是Java方法执行的内存模型：每个方法在执行的同时都会创建一个帧栈（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。它的线程也是私有的，生命周期与线程相同。

局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用和 returnAddress 类型（指向了一条字节码指令的地址）。

Java 虚拟机栈的局部变量表的空间单位是槽（Slot），其中 64 位长度的 double 和 long 类型会占用两个 Slot。局部变量表所需内存空间在编译期完成分配，当进入一个方法时，该方法需要在帧中分配多大的局部变量是完全确定的，在方法运行期间不会改变局部变量表的大小。

Java虚拟机栈有两种异常状况：如果线程请求的栈的深度大于虚拟机所允许的深度，将抛出 StackOverflowError 异常；如果扩展时无法申请到足够的内存，就会抛出 OutOfMemoryError 异常。

- **3. 本地方法栈**

本地方法栈（Native Method Stack）：与虚拟机栈所发挥的作用是非常相似的，它们之间的区别只不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。

Java 虚拟机规范没有对本地方法栈中方法使用的语言、使用的方式和数据结构做出强制规定，因此具体的虚拟机可以自由地实现它。比如：Sun HotSpot 虚拟机直接把Java虚拟机栈和本地方法栈合二为一。

与Java虚拟机栈一样，本地方法栈也会抛出StackOverflowError和 OutOfMemoryError 异常。

- **4. Java 堆**

Java堆（Java Heap）：是被所有线程所共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是：存放对象实例，几乎所有的对象实例都在这里分配内存。

Java 堆是垃圾收集器管理的主要区域，因此很多时候也被称做“GC”堆（Garbage Collected Heap）。从内存回收的角度看，由于现在收集器基本都采用分代收集算法，所以 Java 堆中还可以细分为：新生代和老年代。从内存分配角度来看，线程共享的 Java 堆中可能划分出多个线程私有的分配缓冲区（Thread Local Allocation Buffer, TLAB）。不过无论如何划分，都与存放的内容无关，无论哪个区域，存储的都仍然是对象实例，进一步划分的目的是为了更好地回收内存，或者更快地分配内存。

Java 虚拟机规定，Java 堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可。在实现时，可以是固定大小的，也可以是可扩展的。如果在堆中没有完成实例分配。并且堆也无法扩展时，将会抛出 OutOfMemoryError 异常。

- **5. 方法区**

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

虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 Non-Heap（非堆），其目的应该就是与 Java 堆区分开来。

Java 虚拟机规范对方法区的限制非常宽松，除了和 Java 堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载。

根据Java虚拟机规范规定，当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常。

> **运行时常量池**:运行时常量池（Runtime Constant Pool）： 是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有一些信息是常量池，用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。
>
> Java 虚拟机对 Class 文件每一部分（自然也包括常量池）的格式都有严格的规定，每一个字节用于存储哪种数据都必须符合规范上的要求才会被虚拟机认可、装载和执行。

> **直接内存**:直接内存（Direct Memory）： 并不是虚拟机运行时数据区的一部分，也不是 Java 虚拟机规范中定义的内存区域。但是这部分内存也频繁地使用，而且也可能导致 OutOfMemoryError 异常。
>
> 本地直接内存的分配不会受到 Java 堆大小的限制，但是，既然是内存，肯定还是会受到本机总内存大小以及处理器寻址空间的限制。如果各个内存区域总和大于物理内存限制，从而导致动态扩展时出现 OutOfMemoryError 异常。

3、堆和栈的区别是什么?

堆和栈（虚拟机栈）是完全不同的两块内存区域，一个是线程独享的，一个是线程共享的。二者之间最大的区别就是存储的内容不同：堆中主要存放对象实例。栈（局部变量表）中主要存放各种基本数据类型、对象的引用。

从作用来说，栈是运行时的单位，而堆是存储的单位。栈解决程序的运行问题，即程序如何执行，或者说如何处理数据。堆解决的是数据存储的问题，即数据怎么放、放在哪儿。在 Java 中一个线程就会相应有一个线程栈与之对应，因为不同的线程执行逻辑有所不同，因此需要一个独立的线程栈。而堆则是所有线程共享的。栈因为是运行单位，因此里面存储的信息都是跟当前线程（或程序）相关信息的。包括局部变量、程序运行状态、方法返回值等等；而堆只负责存储对象信息。

4、堆中存什么？栈中存什么?

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

为什么不把基本类型放堆中呢?

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

5、 为什么要把堆和栈区分出来呢？栈中不是也可以存储数据吗?

1. 从软件设计的角度看，栈代表了处理逻辑，而堆代表了数据。这样分开，使得处理逻辑更为清晰。分而治之的思想。这种隔离、模块化的思想在软件设计的方方面面都有体现。
2. 堆与栈的分离，使得堆中的内容可以被多个栈共享（也可以理解为多个线程访问同一个对象）。这种共享的收益是很多的。一方面这种共享提供了一种有效的数据交互方式(如：共享内存)，另一方面，堆中的共享常量和缓存可以被所有栈访问，节省了空间。
3. 栈因为运行时的需要，比如：保存系统运行的上下文，需要进行地址段的划分。由于栈只能向上增长，因此就会限制住栈存储内容的能力。而堆不同，堆中的对象是可以根据需要动态增长的，因此栈和堆的拆分，使得动态增长成为可能，相应栈中只需记录堆中的一个地址即可。

6、Java 中的参数传递时传值呢？还是传引用?

要说明这个问题，先要明确两点：

1. 不要试图与 C 进行类比，Java 中没有指针的概念。
2. 程序运行永远都是在栈中进行的，因而参数传递时，只存在传递基本类型和对象引用的问题。不会直接传对象本身。

Java 在方法调用传递参数时，因为没有指针，所以它都是进行传值调用。但是传引用的错觉是如何造成的呢？在运行栈中，基本类型和引用的处理是一样的，都是传值。所以，如果是传引用的方法调用，也同时可以理解为“传引用值”的传值调用，即引用的处理跟基本类型是完全一样的。但是当进入被调用方法时，被传递的这个引用的值，被程序解释到堆中的对象，这个时候才对应到真正的对象。如果此时进行修改，修改的是引用对应的对象，而不是引用本身，即：修改的是堆中的数据。所以这个修改是可以保持的了。

对象，从某种意义上说，是由基本类型组成的。可以把一个对象看作为一棵树，对象的属性如果还是对象，则还是一颗树（即非叶子节点），基本类型则为树的叶子节点。程序参数传递时，被传递的值本身都是不能进行修改的，但是，如果这个值是一个非叶子节点（即一个对象引用），则可以修改这个节点下面的所有内容。

7、Java 对象的大小是怎么计算的?

基本数据的类型的大小是固定的。对于非基本类型的 Java 对象，其大小就值得商榷。在 Java 中，一个空 Object 对象的大小是 8 byte，这个大小只是保存堆中一个没有任何属性的对象的大小。看下面语句：

```
Object ob = new Object();
```

这样在程序中完成了一个 Java 对象的生命，但是它所占的空间为：4 byte + 8 byte。4 byte 是上面部分所说的 Java 栈中保存引用的所需要的空间。而那 8 byte 则是 Java 堆中对象的信息。因为所有的 Java 非基本类型的对象都需要默认继承 Object 对象，因此不论什么样的 Java 对象，其大小都必须是大于 8 byte。有了 Object 对象的大小，我们就可以计算其他对象的大小了。

```
Class MaNong {
    int count;
    boolean flag;
    Object obj;
}
```

MaNong 的大小为：空对象大小(8 byte) + int 大小(4 byte) + Boolean 大小(1 byte) + 空 Object 引用的大小（4 byte） = 17byte。但是因为 Java 在对对象内存分配时都是以 8 的整数倍来分，因此大于 17 byte 的最接近 8 的整数倍的是 24，因此此对象的大小为 24 byte。

这里需要注意一下基本类型的包装类型的大小。因为这种包装类型已经成为对象了，因此需要把它们作为对象来看待。包装类型的大小至少是12 byte（声明一个空 Object 至少需要的空间），而且 12 byte 没有包含任何有效信息，同时，因为 Java 对象大小是 8 的整数倍，因此一个基本类型包装类的大小至少是 16 byte。这个内存占用是很恐怖的，它是使用基本类型的 N 倍（N > 2），有些类型的内存占用更是夸张（随便想下就知道了）。因此，可能的话应尽量少使用包装类。在 JDK5 以后，因为加入了自动类型装换，因此，Java 虚拟机会在存储方面进行相应的优化。

8、对象的访问定位的两种方式?

Java 程序通过栈上的引用数据来操作堆上的具体对象。目前主流的对象访问方式有：句柄 和 直接指针。

- **1. 使用句柄**

如果使用句柄的话，那么 Java 堆中将会划分出一块内存来作为句柄池，引用中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。

- **2. 直接指针**

如果使用直接指针访问，那么 Java 堆对象的布局中就必须考虑如何防止访问类型数据的相关信息，reference 中存储的直接就是对象的地址。

- **3. 各自的优点**

1、 使用句柄来访问的最大好处是引用中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而引用本身不需要修改；

2、使用直接指针访问方式最大的好处就是速度快，它节省了一次指针定位的时间开销。

9、判断垃圾可以回收的方法有哪些?

垃圾收集器在对堆区和方法区进行回收前，首先要确定这些区域的对象哪些可以被回收，哪些暂时还不能回收，这就要用到判断对象是否存活的算法。

1、引用计数法

- 基本思想

引用计数是垃圾收集器中的早期策略。在这种方法中，堆中每个对象实例都有一个引用计数。当一个对象被创建时，就将该对象实例分配给一个变量，该变量计数设置为 1。当任何其它变量被赋值为这个对象的引用时，计数加 1（a = b，则 b 引用的对象实例的计数器加 1），但当一个对象实例的某个引用超过了生命周期或者被设置为一个新值时，对象实例的引用计数器减 1。任何引用计数器为 0 的对象实例可以被当作垃圾收集。当一个对象实例被垃圾收集时，它引用的任何对象实例的引用计数器减 1。

- 优缺点

优点：引用计数收集器可以很快的执行，交织在程序运行中。对程序需要不被长时间打断的实时环境比较有利。

缺点：无法检测出循环引用。如父对象有一个对子对象的引用，子对象反过来引用父对象。这样，他们的引用计数永远不可能为 0。

例如如下代码：

```
public class Demo{
    public static void main(String[]   args){
        MyObject  object1 = new  MyObject();
        MyObject  object2 = new  MyObject();

        object1.object = object2;
        object2.object = object1;
        object1 = null;
        object2 = null;
```

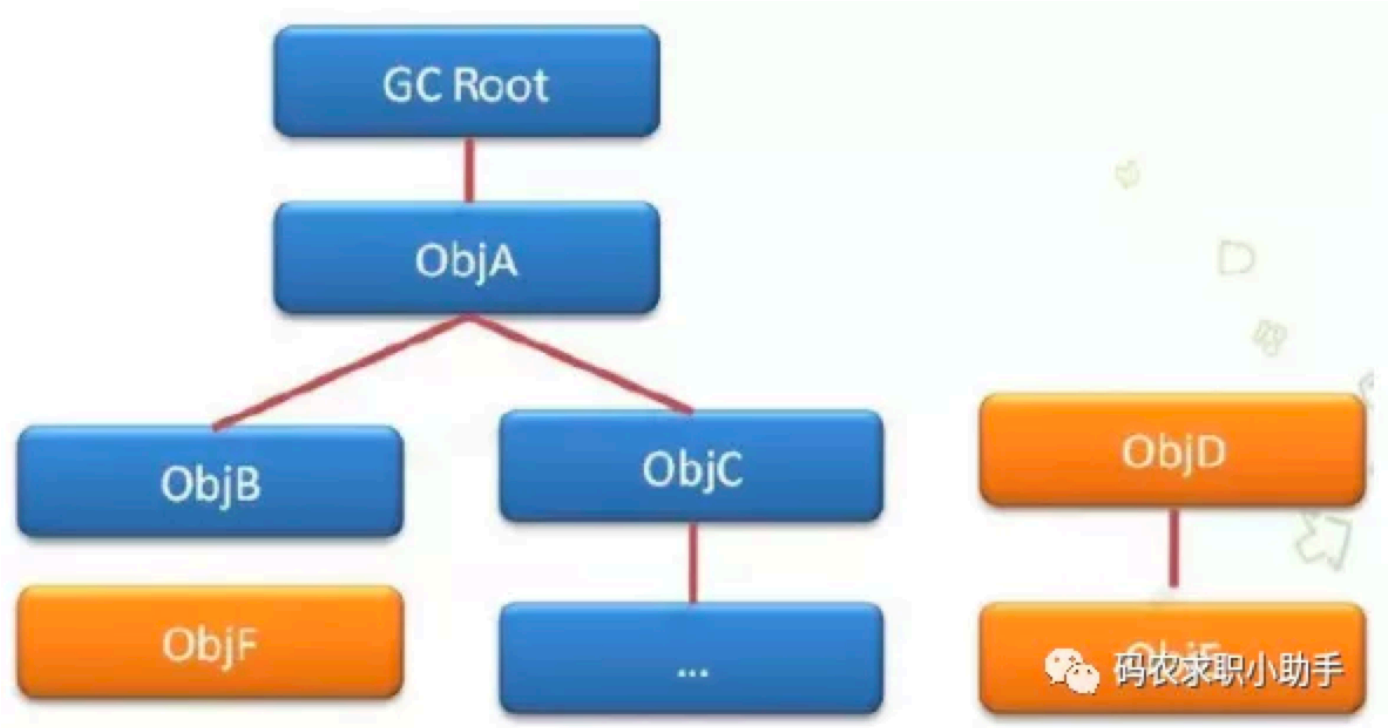
```
    }
}

class   MyObject{
   MyObject    object;
}
```

这段代码是用来验证引用计数算法不能检测出循环引用。最后面两句将 object1 和 object2 赋值为null，也就是说 object1 和 object2 指向的对象已经不可能再被访问，但是由于它们互相引用对方，导致它们的引用计数器都不为 0，那么垃圾收集器就永远不会回收它们。

2、可达性分析算法

可达性分析算法是从离散数学中的图论引入的，程序把所有的引用关系看作一张图，从一个节点 GC ROOT 开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，当所有的引用节点寻找完毕之后，剩余的节点则被认为是没有被引用到的节点，即无用的节点，无用的节点将会被判定为是可回收的对象。

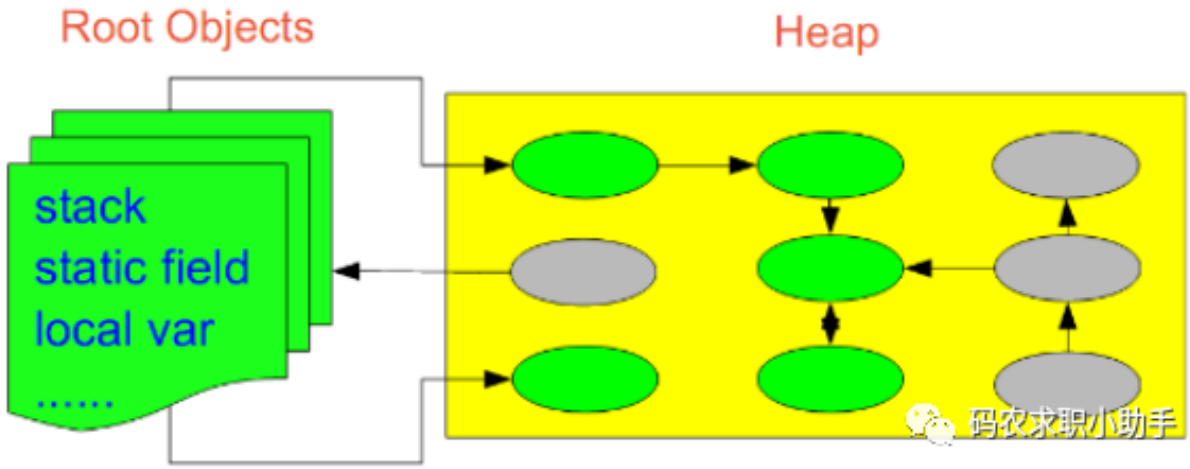


在 Java 语言中，可作为 GC Roots 的对象包括下面几种：

1. 虚拟机栈中引用的对象（栈帧中的本地变量表）；
2. 方法区中类静态属性引用的对象；
3. 方法区中常量引用的对象；
4. 本地方法栈中 JNI（Native方法）引用的对象。

10、垃圾回收是从哪里开始的呢?

查找哪些对象是正在被当前系统使用的。上面分析的堆和栈的区别，其中栈是真正进行程序执行地方，所以要获取哪些对象正在被使用，则需要从 Java 栈开始。同时，一个栈是与一个线程对应的，因此，如果有多个线程的话，则必须对这些线程对应的所有的栈进行检查。



同时，除了栈外，还有系统运行时的寄存器等，也是存储程序运行数据的。这样，以栈或寄存器中的引用为起点，我们可以找到堆中的对象，又从这些对象找到对堆中其他对象的引用，这种引用逐步扩展，最终以 null 引用或者基本类型结束，这样就形成了一颗以 Java 栈中引用所对应的对象为根节点的一颗对象树。如果栈中有多个引用，则最终会形成多颗对象树。在这些对象树上的对象，都是当前系统运行所需要的对象，不能被垃圾回收。而其他剩余对象，则可以视为无法被引用到的对象，可以被当做垃圾进行回收。

11、被标记为垃圾的对象一定会被回收吗?

即使在可达性分析算法中不可达的对象，也并非是“非死不可”，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程。

第一次标记：如果对象在进行可达性分析后发现没有与 GC Roots 相连接的引用链，那它将会被第一次标记；

第二次标记：第一次标记后接着会进行一次筛选，筛选的条件是此对象是否有必要执行 finalize() 方法。在 finalize() 方法中没有重新与引用链建立关联关系的，将被进行第二次标记。第二次标记成功的对象将真的会被回收，如果对象在 finalize() 方法中重新与引用链建立了关联关系，那么将会逃离本次回收，继续存活。

12、谈谈对 Java 中引用的了解?

无论是通过引用计数算法判断对象的引用数量，还是通过可达性分析算法判断对象的引用链是否可达，判定对象是否存活都与“引用”有关。在Java语言中，将引用又分为强引用、软引用、弱引用、虚引用 4 种，这四种引用强度依次逐渐减弱。

- **1. 强引用**

在程序代码中普遍存在的，类似 Object obj = new Object() 这类引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象。

- **2. 软引用**

用来描述一些还有用但并非必须的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行第二次回收。如果这次回收后还没有足够的内

- **3. 弱引用**

也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。

- **4. 虚引用**

也叫幽灵引用或幻影引用，是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。它的作用是能在这个对象被收集器回收时收到一个系统通知。

13、谈谈对内存泄漏的理解?

- **内存泄露的基本概念**

在 Java 中，内存泄漏就是存在一些不会再被使用确没有被回收的对象，这些对象有下面两个特点：

1. 这些对象是可达的，即在有向图中，存在通路可以与其相连；
2. 这些对象是无用的，即程序以后不会再使用这些对象。

如果对象满足这两个条件，这些对象就可以判定为 Java 中的内存泄漏，这些对象不会被 GC 所回收，然而它却占用内存。

14、内存泄露的根本原因是什么?

长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄漏，尽管短生命周期对象已经不再需要，但是因为长生命周期持有它的引用而导致不能被回收，这就是 Java 中内存泄漏的发生场景。

15、举几个可能发生内存泄漏的情况?

1. 静态集合类引起的内存泄漏；
2. 当集合里面的对象属性被修改后，再调用 remove() 方法时不起作用；
3. 监听器：释放对象的时候没有删除监听器；
4. 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；
5. 内部类：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；
6. 单例模式：单例对象在初始化后将在 JVM 的整个生命周期中存在（以静态变量的方式），如果单例对象持有外部的引用，那么这个对象将不能被 JVM 正常回收，导致内存泄漏。

16、尽量避免内存泄漏的方法?

1. 尽量不要使用 static 成员变量，减少生命周期；
2. 及时关闭资源；
3. 不用的对象，可以手动设置为 null。

17、常用的垃圾收集算法有哪些?

- **1. 标记-清除算法（Mark-Sweep）**

标记-清除算法采用从根集合（GC Roots）进行扫描，对存活的对象进行标记，标记完毕后，再扫描整个空间中未被标记的对象，进行回收。标记-清除算法不需要进行对象的移动，只需对不存活的对象进行处理，在存活对象比较多的情况下极为高效，但由于标记-清除算法直接回收不存活的对象，因此会造成内存碎片。

- **2. 复制算法(Copying)**

复制算法的提出是为了克服句柄的开销和解决内存碎片的问题。它开始时把堆分成 一个对象面和多个空闲面， 程序从对象面为对象分配空间，当对象满了，基于 copying 算法的垃圾收集就从根集合（GC Roots）中扫描活动对象，并将每个活动对象复制到空闲面（使得活动对象所占的内存之间没有空闲洞），这样空闲面变成了对象面，原来的对象面变成了空闲面，程序会在新的对象面中分配内存。

- **3. 标记-整理算法(Mark-compact)**

标记-整理算法采用标记-清除算法一样的方式进行对象的标记，但在清除时不同，在回收不存活的对象占用的空间后，会将所有的存活对象往左端空闲空间移动，并更新对应的指针。标记-整理算法是在标记-清除算法的基础上，又进行了对象的移动，因此成本更高，但是却解决了内存碎片的问题。

- **4. 分代收集算法**

分代收集算法是目前大部分 JVM 的垃圾收集器采用的算法。它的核心思想是根据对象存活的生命周期将内存划分为若干个不同的区域。一般情况下将堆区划分为老年代（Tenured Generation）和新生代（Young Generation），在堆区之外还有一个代就是永久代（Permanet Generation）。

老年代的特点是每次垃圾收集时只有少量对象需要被回收，而新生代的特点是每次垃圾回收时都有大量的对象需要被回收，那么就可以根据不同代的特点采取最适合的收集算法。

18、为什么要采用分代收集算法?

分代的垃圾回收策略，是基于这样一个事实：不同的对象的生命周期是不一样的。因此，不同生命周期的对象可以采取不同的收集方式，以便提高回收效率。

在 Java 程序运行的过程中，会产生大量的对象，其中有些对象是与业务信息相关，比如 Http 请求中的 Session 对象、线程、Socket 连接，这类对象跟业务直接挂钩，因此生命周期比较长。但是还有一些对象，主要是程序运行过程中生成的临时变量，这些对象生命周期会比较短，比如：String 对象，由于其不变类的特性，系统会产生大量的这些对象，有些对象甚至只用一次即可回收。

在不进行对象存活时间区分的情况下，每次垃圾回收都是对整个堆空间进行回收，花费时间相对会长，同时，因为每次回收都需要遍历所有存活对象，但实际上，对于生命周期长的对象而言，这种遍历是没有效果的，因为可能进行了很多次遍历，但是他们依旧存在。因此，分代垃圾回收采用分治的思想，进行代的划分，把不同生命周期的对象放在不同代上，不同代上采用最适合它的垃圾回收方式进行回收。

19、分代收集下的年轻代和老年代应该采用什么样的垃圾回收算法?

1、年轻代（Young Generation）的回收算法 (主要以 Copying 为主)

1. 所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。
2. 新生代内存按照 8:1:1 的比例分为一个 eden 区和两个 survivor（survivor0、 survivor1）区。大部分对象在 Eden 区中生成。回收时先将 Eden 区存活对象复制到一个 survivor0 区，然后清空 eden 区，当这个 survivor0 区也存放满了时，则将 eden 区和 survivor0 区存活对象复制到另一个 survivor1 区，然后清空 eden 区 和这个 survivor0 区，此时 survivor0 区是空的，然后将survivor0 区和 survivor1 区交换，即保持 survivor1 区为空， 如此往复。
3. 当 survivor1 区不足以存放 Eden 区 和 survivor0区 的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次Full GC（Major GC），也就是新生代、老年代都进行回收。
4. 新生代发生的 GC 也叫做 Minor GC，MinorGC 发生频率比较高（不一定等 Eden 区满了才触发）。

2、年老代（Old Generation）的回收算法（主要以 Mark-Compact 为主）

1. 在年轻代中经历了 N 次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。
2. 内存比新生代也大很多（大概比例是1 : 2），当老年代内存满时触发 Major GC 即 Full GC，Full GC 发生频率比较低，老年代对象存活时间比较长，存活率标记高。

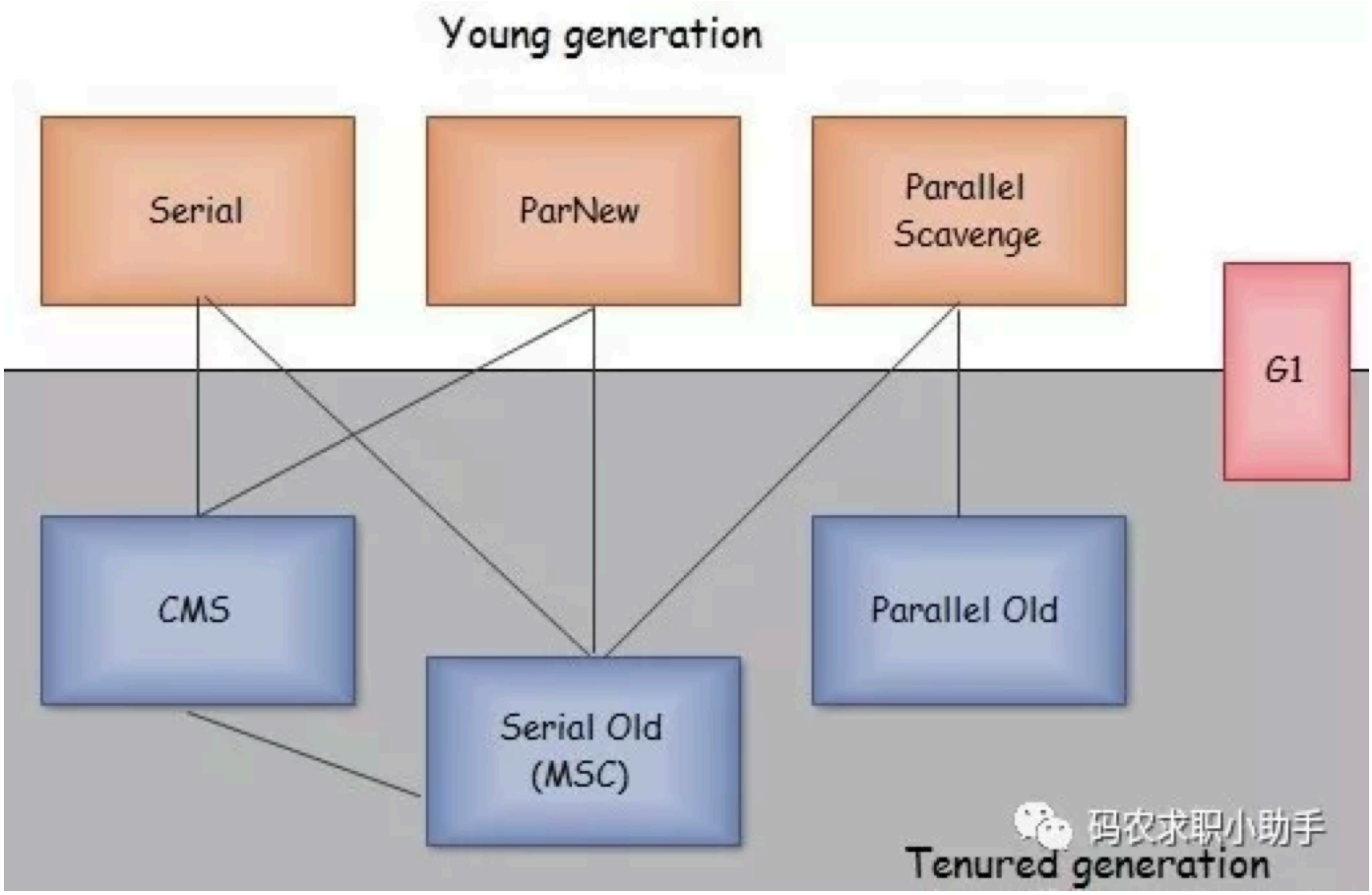
20、什么是浮动垃圾?

由于在应用运行的同时进行垃圾回收，所以有些垃圾可能在垃圾回收进行完成时产生，这样就造成了“Floating Garbage”，这些垃圾需要在下次垃圾回收周期时才能回收掉。所以，并发收集器一般需要20%的预留空间用于这些浮动垃圾。

21、什么是内存碎片？如何解决?

由于不同 Java 对象存活时间是不一定的，因此，在程序运行一段时间以后，如果不进行内存整理，就会出现零散的内存碎片。碎片最直接的问题就是会导致无法分配大块的内存空间，以及程序运行效率降低。所以，在上面提到的基本垃圾回收算法中，“复制”方式和“标记-整理”方式，都可以解决碎片的问题。

22、常用的垃圾收集器有哪些?



- **1. Serial 收集器（复制算法)**

新生代单线程收集器，标记和清理都是单线程，优点是简单高效。是 client 级别默认的 GC 方式，可以通过 -XX:+UseSerialGC 来强制指定。

- **2. Serial Old 收集器（标记-整理算法）**

老年代单线程收集器，Serial 收集器的老年代版本。

- **3. ParNew 收集器（停止-复制算法）**

新生代收集器，可以认为是 Serial 收集器的多线程版本，在多核 CPU 环境下有着比 Serial 更好的表现。

- **4. Parallel Scavenge 收集器（停止-复制算法）**

并行收集器，追求高吞吐量，高效利用 CPU。吞吐量一般为 99%， 吞吐量= 用户线程时间 / (用户线程时间+GC线程时间)。适合后台应用等对交互相应要求不高的场景。是 server 级别默认采用的GC方式，可用 -XX:+UseParallelGC 来强制指定，用 -XX:ParallelGCThreads=4 来指定线程数。

- **5. Parallel Old 收集器（停止-复制算法）**

Parallel Old 收集器的老年代版本，并行收集器，吞吐量优先。

- **6. CMS(Concurrent Mark Sweep)收集器（标记-清除算法）**

高并发、低停顿，追求最短 GC 回收停顿时间，cpu 占用比较高，响应时间快，停顿时间短，多核 cpu 追求高响应时间的选择。

CMS 是英文 Concurrent Mark-Sweep 的简称，是以牺牲吞吐量为代价来获得最短回收停顿时间的垃圾回收器。对于要求服务器响应速度的应用上，这种垃圾回收器非常适合。在启动 JVM 的参数加上“-XX:+UseConcMarkSweepGC”来指定使用 CMS 垃圾回收器。

CMS 使用的是标记-清除的算法实现的，所以在 GC 的时候会产生大量的内存碎片，当剩余内存不能满足程序运行要求时，系统将会出现 Concurrent Mode Failure，临时 CMS 会采用 Serial Old 回收器进行垃圾清除，此时的性能将会被降低。

- **7. G1**

G1 收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的 Region(这也就是它的名字 Garbage-First的由来。

23、谈谈你对 CMS 垃圾收集器的理解?

CMS 是英文 Concurrent Mark-Sweep 的简称，是以牺牲吞吐量为代价来获得最短回收停顿时间的垃圾回收器。是使用标记清除算法实现的，整个过程分为四步：

1. 初始标记：记录下直接与 root 相连的对象，暂停所有的其他线程，速度很快；
2. 并发标记：同时开启 GC 和用户线程，用一个闭包结构去记录可达对象。但在这个阶段结束，这个闭包结构并不能保证包含当前所有的可达对象。因为用户线程可能会不断的更新引用域，所以 GC 线程无法保证可达性分析的实时性。所以这个算法里会跟踪记录这些发生引用更新的地方。
3. 重新标记：重新标记阶段就是为了修正并发标记期间因为用户程序继续运行而导致标记产生变动的那一部分对象的标记记录。【这个阶段的停顿时间一般会比初始标记阶段的时间稍长，远远比并发标记阶段时间短】；
4. 并发清除：开启用户线程，同时 GC 线程开始对为标记的区域做清扫。

CMS 的优缺点：

主要优点：并发收集、低停顿；

主要缺点：对 CPU 资源敏感、无法处理浮动垃圾、它使用的回收算法“标记-清除”算法会导致收集结束时会有大量空间碎片产生。

24、谈谈你对 G1 收集器的理解?

垃圾回收的瓶颈：传统分代垃圾回收方式，已经在一定程度上把垃圾回收给应用带来的负担降到了最小，把应用的吞吐量推到了一个极限。但是他无法解决的一个问题，就是 Full GC 所带来的应用暂停。在一些对实时性要求很高的应用场景下，GC 暂停所带来的请求堆积和请求失败是无法接受的。这类应用可能要求请求的返回时间在几百甚至几十毫秒以内，如果分代垃圾回收方式要达到这个指标，只能把最大堆的设置限制在一个相对较小范围内，但是这样有限制了应用本身的处理能力，同样也是不可接受的。

分代垃圾回收方式确实也考虑了实时性要求而提供了并发回收器，支持最大暂停时间的设置，但是受限于分代垃圾回收的内存划分模型，其效果也不是很理想。

G1 可谓博采众家之长，力求到达一种完美。它吸取了增量收集优点，把整个堆划分为一个一个等大小的区域（region）。内存的回收和划分都以region为单位；同时，它也吸取了 CMS 的特点，把这个垃圾回收过程分为几个阶段，分散一个垃圾回收过程；而且，G1 也认同分代垃圾回收的思想，认为不同对象的生命周期不同，可以采取不同收集方式，因此，它也支持分代的垃圾回收。为了达到对回收时间的可预计性，G1 在扫描了 region 以后，对其中的活跃对象的大小进行排序，首先会收集那些活跃对象小的 region，以便快速回收空间（要复制的活跃对象少了），因为活跃对象小，里面可以认为多数都是垃圾，所以这种方式被称为 Garbage First（G1）的垃圾回收算法，即：垃圾优先的回收。

25、说下你对垃圾回收策略的理解/垃圾回收时机?

- **1. Minor / Scavenge GC**

所有对象创建在新生代的 Eden 区，当 Eden 区满后触发新生代的 Minor GC，将 Eden 区和非空闲 Survivor 区存活的对象复制到另外一个空闲的 Survivor 区中。保证一个 Survivor 区是空的，新生代 Minor GC 就是在两个 Survivor 区之间相互复制存活对象，直到 Survivor 区满为止。

Minor/Scavenge 这种方式的 GC 是在年轻代的 Eden 区进行，不会影响到年老代。因为大部分对象都是从 Eden 区开始的，同时 Eden 区不会分配的很大，所以 Eden 区的 GC 会频繁进行。因而，一般在这里需要使用速度快、效率高的算法，使 Eden 去能尽快空闲出来。

- **2. Full GC**

对整个堆进行整理，包括 Young、Tenured 和 Perm。Full GC 因为需要对整个堆进行回收，所以比 Minor GC 要慢，因此应该尽可能减少 Full GC 的次数。在对 JVM 调优的过程中，很大一部分工作就是对于 Full GC 的调节。

Minor 有如下原因可能导致 Full GC：

1、 调用 System.gc()，会建议虚拟机执行 Full GC。只是建议虚拟机执行 Full GC，但是虚拟机不一定真正去执行。

2、 老年代空间不足，原因：老年代空间不足的常见场景为大对象直接进入老年代、长期存活的对象进入老年代等。为了避免以上原因引起的 Full GC，应当尽量不要创建过大的对象以及数组。除此之外，可以通过 -Xmn 虚拟机参数调大新生代的大小，让对象尽量在新生代被回收掉，不进入老年代。还可以通过 -XX:MaxTenuringThreshold 调大对象进入老年代的年龄，让对象在新生代多存活一段时间；

3、 空间分配担保失败：使用复制算法的 Minor GC 需要老年代的内存空间作担保，如果担保失败会执行一次 Full GC；

4、JDK 1.7 及以前的永久代空间不足。在 JDK1.7 及以前，HotSpot 虚拟机中的方法区是用永久代实现的，永久代中存放的为一些 Class 的信息、常量、静态变量等数据。当系统中要加载的类、反射的类和调用的方法较多时，永久代可能会被占满，在未配置为采用 CMS GC 的情况下也会执行 Full GC。如果经过 Full GC 仍然回收不了，那么虚拟机会抛出 java.lang.OutOfMemoryError。为避免以上原因引起的 Full GC，可采用的方法为增大永久代空间或转为使用 CMS GC。

5、Concurrent Mode Failure 执行 CMS GC 的过程中，同时有对象要放入老年代，而此时老年代空间不足（可能是 GC 过程中浮动垃圾过多导致暂时性的空间不足），便会报 Concurrent Mode Failure 错误，并触发 Full GC。

26、谈谈你对内存分配的理解？大对象怎么分配？空间分配担保?

1. 对象优先在 Eden 区分配：大多数情况下，对象在新生代 Eden 区分配，当 Eden 区空间不够时，发起 Minor GC。
2. 大对象直接进入老年代：大对象是指需要连续内存空间的对象，最典型的大对象是那种很长的字符串以及数组。经常出现大对象会提前触发垃圾收集以获取足够的连续空间分配给大对象。-XX:PretenureSizeThreshold，大于此值的对象直接在老年代分配，避免在 Eden 区和 Survivor 区之间的大量内存复制。
3. 长期存活的对象将进入老年代：为对象定义年龄计数器，对象在 Eden 出生并经过 Minor GC 依然存活，将移动到 Survivor 中，年龄就增加 1 岁，增加到一定年龄则移动到老年代中。-XX:MaxTenuringThreshold 用来定义年龄的阈值。
4. 动态对象年龄判定：为了更好的适应不同程序的内存情况，虚拟机不是永远要求对象年龄必须达到了某个值才能进入老年代，如果 Survivor 空间中相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无需达到要求的年龄。
5. 空间分配担保

（1）在发生 Minor GC 之前，虚拟机先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果条件成立的话，那么 Minor GC 可以确认是安全的；

（2）如果不成立的话，虚拟机会查看 HandlePromotionFailure 设置值是否允许担保失败，如果允许那么就会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次 Minor GC；如果小于，或者 HandlePromotionFailure 设置不允许冒险，那么就要进行一次 Full GC。

27、说下你用过的 JVM 监控工具?

1. jvisualvm：虚拟机监视和故障处理平台
2. jps ： 查看当前 Java 进程
3. jstat：显示虚拟机运行数据
4. jmap：内存监控
5. jhat：分析 heapdump 文件
6. jstack：线程快照
7. jinfo：虚拟机配置信息

28、如何利用监控工具调优?

1、堆信息查看

1. 可查看堆空间大小分配（年轻代、年老代、持久代分配）
2. 提供即时的垃圾回收功能
3. 垃圾监控（长时间监控回收情况）
4. 查看堆内类、对象信息查看：数量、类型等
5. 对象引用情况查看

有了堆信息查看方面的功能，我们一般可以顺利解决以下问题：

1. 年老代年轻代大小划分是否合理
2. 内存泄漏垃
3. 圾回收算法设置是否合理

2、线程监控

线程信息监控：系统线程数量

线程状态监控：各个线程都处在什么样的状态下

Dump 线程详细信息：查看线程内部运行情况

死锁检查

3、 热点分析

1. CPU 热点：检查系统哪些方法占用的大量 CPU 时间；
2. 内存热点：检查哪些对象在系统中数量最大（一定时间内存活对象和销毁对象一起统计）这两个东西对于系统优化很有帮助。我们可以根据找到的热点，有针对性的进行系统的瓶颈查找和进行系统优化，而不是漫无目的的进行所有代码的优化。

4、快照

快照是系统运行到某一时刻的一个定格。在我们进行调优的时候，不可能用眼睛去跟踪所有系统变化，依赖快照功能，我们就可以进行系统两个不同运行时刻，对象（或类、线程等）的不同，以便快速找到问题。

举例说，我要检查系统进行垃圾回收以后，是否还有该收回的对象被遗漏下来的了。那么，我可以在进行垃圾回收前后，分别进行一次堆情况的快照，然后对比两次快照的对象情况。

5、内存泄露检查

内存泄漏是比较常见的问题，而且解决方法也比较通用，这里可以重点说一下，而线程、热点方面的问题则是具体问题具体分析了。

内存泄漏一般可以理解为系统资源（各方面的资源，堆、栈、线程等）在错误使用的情况下，导致使用完毕的资源无法回收（或没有回收），从而导致新的资源分配请求无法完成，引起系统错误。内存泄漏对系统危害比较大，因为它可以直接导致系统的崩溃。

29、JVM 的一些参数?

- **1. 堆设置**

-Xms：初始堆大小

-Xmx：最大堆大小

-XX:NewSize=n：设置年轻代大小

-XX:NewRatio=n：设置年轻代和年老代的比值。如:为3，表示年轻代与年老代比值为 1：3，年轻代占整个年轻代年老代和的 1/4

-XX:SurvivorRatio=n：年轻代中 Eden 区与两个 Survivor 区的比值。注意 Survivor 区有两个。如：3，表示 Eden：Survivor=3：2，一个Survivor区占整个年轻代的 1/5

-XX:MaxPermSize=n：设置持久代大小

- **2. 收集器设置**

-XX:+UseSerialGC：设置串行收集器

-XX:+UseParallelGC：设置并行收集器

-XX:+UseParalledlOldGC：设置并行年老代收集器

-XX:+UseConcMarkSweepGC：设置并发收集器

- **3. 垃圾回收统计信息**

-XX:+PrintGC：开启打印 gc 信息

-XX:+PrintGCDetails：打印 gc 详细信息

-XX:+PrintGCTimeStamps

-Xloggc:filename

- **4. 并行收集器设置**

-XX:ParallelGCThreads=n：设置并行收集器收集时使用的 CPU 数

-XX:MaxGCPauseMillis=n：设置并行收集最大暂停时间

-XX:GCTimeRatio=n：设置垃圾回收时间占程序运行时间的百分比

- **5. 并发收集器设置**

-XX:+CMSIncrementalMode：设置为增量模式。适用于单 CPU 情况

-XX:ParallelGCThreads=n：设置并发收集器年轻代收集方式为并行收集时，使用的 CPU 数。并行收集线程数

30、谈谈你对类文件结构的理解？有哪些部分组成?

Class 文件结构如下标所示：

类型	名称	数量
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count - 1
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attribute_info	attributes	attribu[illegible]s_count

Class 文件没有任何分隔符，严格按照上面结构表中的顺序排列。无论是顺序还是数量，甚至于数据存储的字节序这样的细节，都是被严格限定的，哪个字节代表什么含义，长度是多少，先后顺序如何，都不允许改变。

1. 魔数（magic）：每个 Class 文件的头 4 个字节称为魔数（Magic Number），它的唯一作用是确定这个文件是否为一个能被虚拟机接受的Class 文件，即判断这个文件是否符合 Class 文件规范。
2. 文件的版本：minor_version 和 major_version。
3. 常量池：constant_pool_count 和 constant_pool：常量池中主要存放两大类常量：字面量（Literal）和符号引用（Symbolic References）。
4. 访问标志：access_flags：用于识别一些类或者接口层次的访问信息。包括：这个 Class 是类还是接口、是否

定义了 Public 类型、是否定义为 abstract 类型、如果是类，是否被声明为了 final 等等。

5.类索引、父类索引与接口索引集合：this_class、super_class和interfaces。

6. 字段表集合：field_info、fields_count：字段表（field_info）用于描述接口或者类中声明的变量；fields_count 字段数目：表示Class文件的类和实例变量总数。
7. 方法表集合：methods、methods_count
8. 属性表集合：attributes、attributes_count

31、谈谈你对类加载机制的了解?

虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这就是虚拟机的类加载机制。

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载、验证、准备、解析、初始化、使用、卸载 7 个阶段。其中验证、准备、解析 3 个部分统称为连接，这7个阶段发生的顺序如下图所示：

32、类加载各阶段的作用分别是什么?

- **1. 加载**

在加载阶段，虚拟机需要完成以下三件事情：

1、通过一个类的全限定名来获取定义此类的二进制字节流；

2、 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构；

3、在内存中生成一个代表这个类的 java.lang.Class 对象，作为方法区这个类的各种数据的访问接口。

- **2. 验证**

主要是为了确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。验证阶段大致上分为 4 个阶段的检验动作：文件格式验证、元数据验证、字节码验证、符号引用验证。

1、文件格式校验：验证字节流是否符合 class 文件的规范，并且能被当前版本的虚拟机处理。只有通过这个阶段的验证后，字节流才会进入内存的方法区进行存储，所以后面的3个阶段的全部是基于方法区的存储结构进行的，不会再直接操作字节流；

2、 元数据验证：对字节码描述的信息进行语义分析，以保证其描述的信息符合 Java 语言规范的要求。目的是保证不存在不符合 Java 语言规范的元数据信息；

3、 字节码验证：该阶段主要工作是进行数据流和控制流分析，保证被校验类的方法在运行时不会做出危害虚拟机安全的行为；

4、 符号引用验证：最后一个阶段的校验发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在连接的第三个阶段——解析阶段中发生。符号引用验证的目的是确保解析动作能正常执行。

- **3. 准备**

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法区中进行分配**。这时候进行内存分配的仅包括类变量（被 static 修饰的变量），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。实例化不是类加载的一个过程，类加载发生在所有实例化操作之前，并且类加载只进行一次，实例化可以进行多次。

初始值是默认值 0 或 false 或 null。如果类变量是常量（final），那么会按照表达式来进行初始化，而不是赋值为 0。public static final int value = 123;

- **4. 解析**

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

- **5. 初始化**

在准备阶段，变量已经赋过一次系统要求的初始值了，而在初始化阶段，则根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者可以从另外一个角度来表达：初始化阶段是执行类构造器 () 方法的过程。

33、有哪些类加载器？分别有什么作用？

1. 启动类加载器(Bootstrap ClassLoader)：这个类加载器是由 C++ 语言实现的，是虚拟机自身的一部分。负责将存在 <JAVA_HOME>\lib 目录中的，或者被 -Xbootclasspath 参数所指定的路径中的类库加载到虚拟机内存中。启动内加载器无法被 Java 程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给启动类加载器，直接使用 null 即可；
2. 其他类加载器：由 Java 语言实现，独立于虚拟机外部，并且全都继承自抽象类 java.lang.ClassLoader。如扩展类加载器和应用程序类加载器：

（1）扩展类加载器(Extension ClassLoader)：这个类加载器由sun.misc.Launcher$ExtClassLoader 实现，它负责加载<JAVA_HOME>\lib\ext目录中的，或者被 java.ext.dirs 系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器。

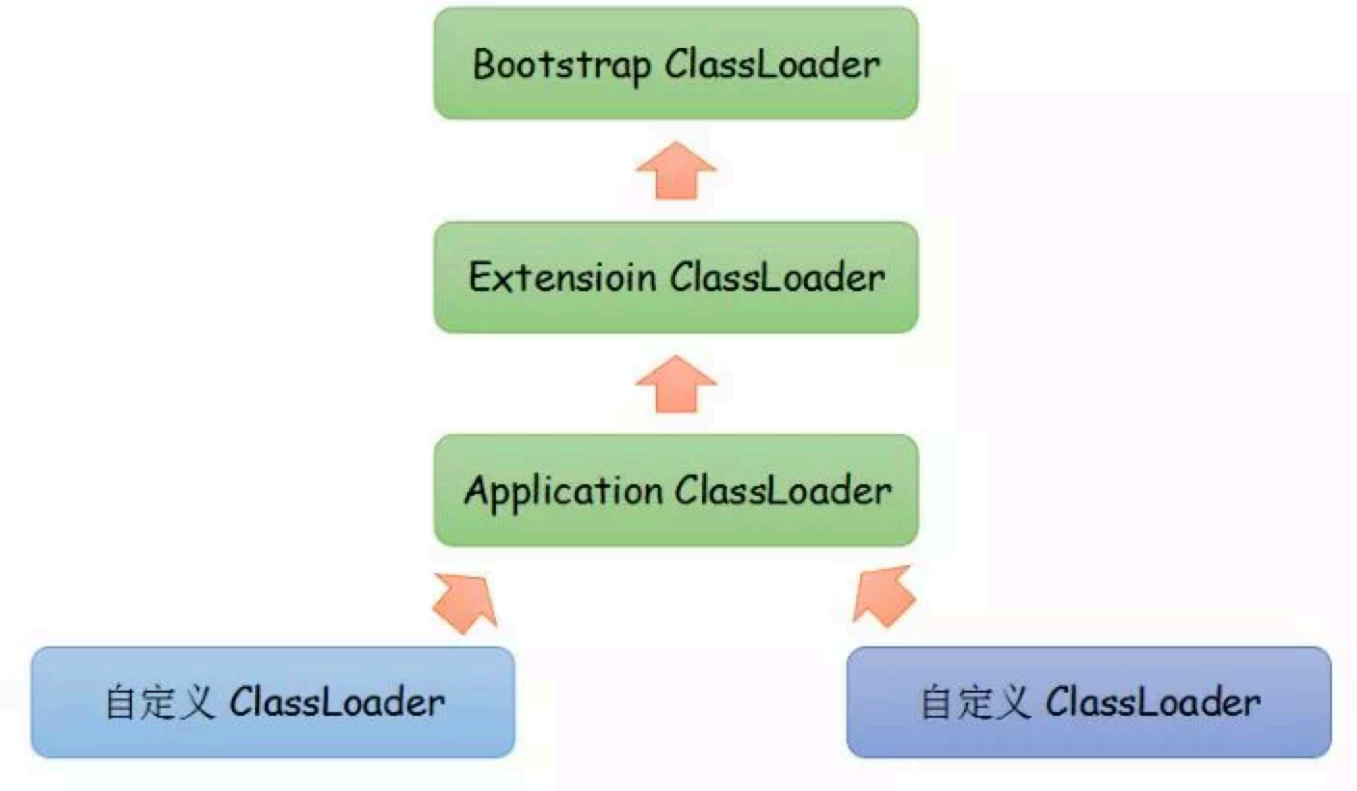
（2）应用程序类加载器 (Application ClassLoader)：这个类加载器由 sun.misc.Launcher$AppClassLoder 实现。由于个类加载器是 ClassLoader 中的 getSystemClassLoader() 方法的返回值，所以一般也称之为系统类加载器。它负责加载用户路径（ClassPath）所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

34、类与类加载器的关系？

类加载器虽然只用于实现类的加载动作，但它在 Java 程序中起到的作用却远远不限于类加载阶段。对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在 Java 虚拟机中的唯一性，每个类加载器，都拥有一个独立的类名称空间。换句话说：比较两个类是否“相等”，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来源于同一个 Class 文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那么这两个类就必定不相等。

35、谈谈你对双亲委派模型的理解？工作过程？为什么要使用？

应用程序一般是由上诉的三种类加载器相互配合进行加载的，如果有必要，还可以加入自己定义的类加载器，它们的关系如下图所示：



- **双亲委派模型的工作过程：**

如果一个类加载器收到了类加载请求，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。每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父类加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

- **使用双亲委派模型的好处：**

Java 类随着它的类加载器一起具备了一种带有优先级的层次关系。例如：类 java.lang.Object，它存放在 rt.jar 中，无论哪一个类加载器需要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此 Object 类在程序的各种类加载器环境中都是同一个类（使用的是同一个类加载器加载的）。相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，如果用户自己编写了一个 java.lang.Object 类，并放在程序的 ClassPath 中，那么系统将会出现多个不同的 Object 类，Java 类型体系中最基础的行为也就无法保证，应用程序也将变得一片混乱。

- **双亲委派模型的主要代码实现：**

实现双亲委派的代码都集中在 java.lang.ClassLoader 的 loadClass() 方法中，逻辑清晰易懂：先检查是否已经被加载过，若没有加载则调用父加载器的 loadClass() 方法，若父加载器为空则默认使用启动类加载器作为父类加载器。如果父类加载失败，抛出 ClassNotFoundException 异常后，再调用自己的 findClass() 方法进行加载。

36、怎么实现一个自定义的类加载器？需要注意什么？

若要实现自定义类加载器，只需要继承 java.lang.ClassLoader 类，并且重写其 findClass() 方法即可。

37、怎么打破双亲委派模型？

1. 自己写一个类加载器；
2. 重写 loadClass() 方法
3. 重写 findClass() 方法

这里最主要的是重写 loadClass 方法，因为双亲委派机制的实现都是通过这个方法实现的，先找父加载器进行加载，如果父加载器无法加载再由自己来进行加载，源码里会直接找到根加载器，重写了这个方法以后就能自己定义加载的方式了。

38、有哪些实际场景是需要打破双亲委派模型的？

JNDI 服务，它的代码由启动类加载器去加载，但 JNDI 的目的就是对资源进行集中管理和查找，它需要调用独立厂商实现部部署在应用程序的 classpath 下的 JNDI 接口提供者(SPI, Service Provider Interface) 的代码，但启动类加载器不可能“认识”之些代码，该怎么办？

为了解决这个困境，Java 设计团队只好引入了一个不太优雅的设计：**线程上下文件类加载器(Thread Context ClassLoader)。这个类加载器可以通过 java.lang.Thread 类的 setContextClassLoader() 方法进行设置，如果创建线程时还未设置，它将会从父线程中继承一个；如果在应用程序的全局范围内都没有设置过，那么这个类加载器默认就是应用程序类加载器。有了线程上下文类加载器，JNDI 服务使用这个线程上下文类加载器去加载所需要的 SPI 代码，也就是父类加载器请求子类加载器去完成类加载动作，这种行为实际上就是打通了双亲委派模型的层次结构来逆向使用类加载器，已经违背了双亲委派模型，但这也是无可奈何的事情。Java 中所有涉及 SPI 的加载动作基本上都采用这种方式，例如 JNDI、JDBC、JCE、JAXB 和 JBI 等。

39、谈谈你对编译期优化和运行期优化的理解？

1、**编译期优化：**

1. 解析与填充符号表的过程
2. 插入式注解处理器的注解处理过程
3. 分析与字节码生成过程

2、**编译优化：**

1. 方法内联
2. 公共子表达式消除
3. 数组范围检查消除
4. 逃逸分析

40、为何 HotSpot 虚拟机要使用解释器与编译器并存的架构？

解释器：程序可以迅速启动和执行，消耗内存小 （类似人工，成本低，到后期效率低）；

编译器：随着代码频繁执行会将代码编译成本地机器码 （类似机器，成本高，到后期效率高）。

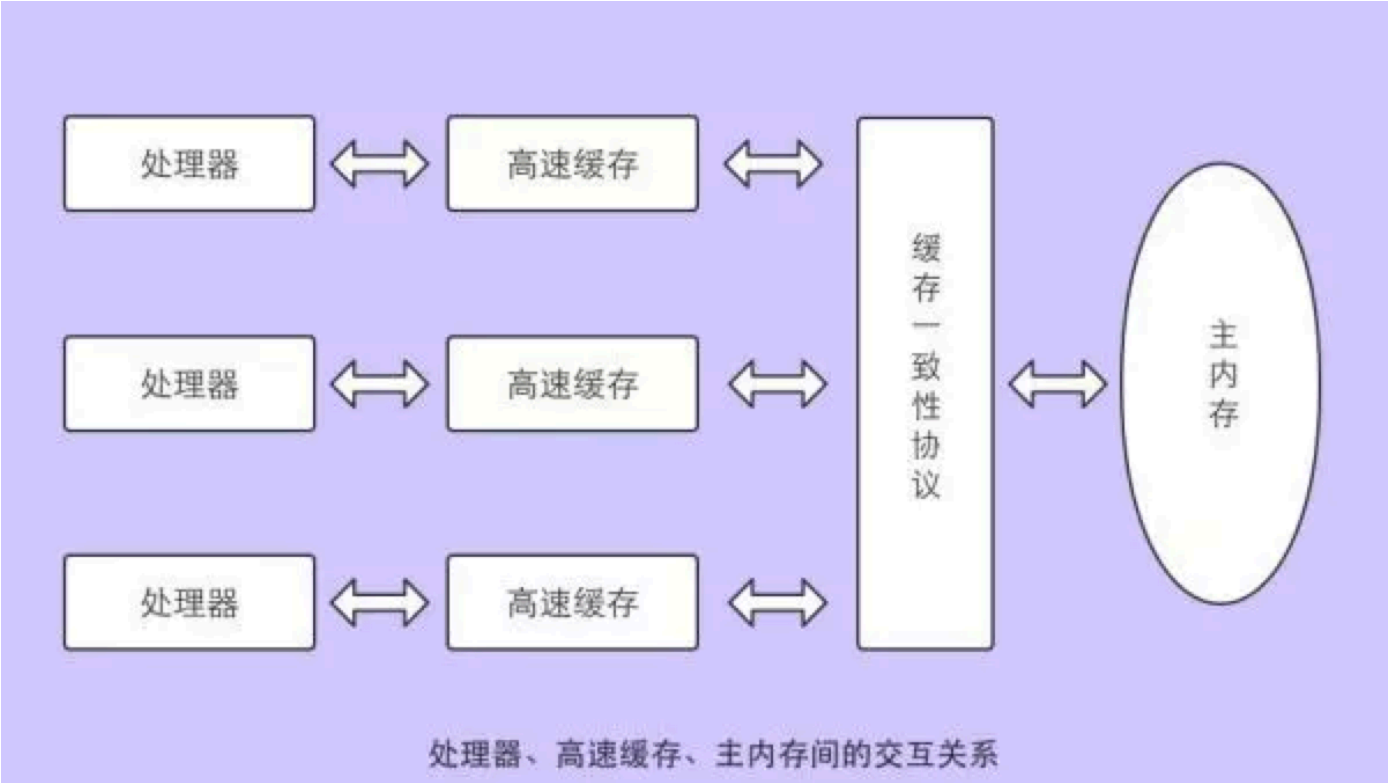
在整个虚拟机执行架构中，解释器与编译器经常配合工作，两者各有优势：当程序需要迅速启动和执行的时候，解释器可以首先发挥作用，省去编译的时间，立即执行。在程序运行后，随着时间的推移，编译器逐渐发挥作用，把越来越多的代码编译成本地代码之后，可以获取更高的执行效率。当程序运行环境中内存资源限制较大（如部分嵌入式系统），可以使用解释执行节约内存，反之可以使用编译执行来提升效率。

解释执行可以节约内存，而编译执行可以提升效率。因此，在整个虚拟机执行架构中，解释器与编译器经常配合工作。

41、说下你对 Java 内存模型的理解?

处理器和内存不是同数量级，所以需要在中间建立中间层，也就是高速缓存，这会引出缓存一致性问题。在多处理器系统中，每个处理器都有自己的高速缓存，而它们又共享同一主内存（Main Memory），有可能操作同一位置引起各自缓存不一致，这时候需要约定协议在保证一致性。

Java 内存模型(Java Memory Model，JMM)：屏蔽掉了各种硬件和操作系统的内存访问差异，以实现让 Java 程序在各种平台下都能达到一致性的内存访问效果



处理器、高速缓存、主内存间的交互关系

- **主内存与工作内存**

Java 内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。

Java 内存模型规定了所有的变量都存储在主内存（Main Memory）中，每个线程有自己的工作线程（Working Memory），保存主内存副本拷贝和自己私有变量，不同线程不能访问工作内存中的变量。线程间变量值的传递需要通过主内存来完成。

42、内存间的交互操作有哪些？需要满足什么规则?

关于主内存与工作内存之间的具体的交互协议，即：一个变量如何从主内存拷贝到工作内存、如何从工作内存同步主内存之类的实现细节，Java内存模型中定义一下八种操作来完成：

1. lock(锁定)：作用于主内存的变量。它把一个变量标志为一个线程独占的状态；
2. unlock(解锁)：作用于主内存的变量，它把处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定；
3. read(读取)：作用于主内存的变量，它把一个变量的值从主内存传输到线程的工作内存中，以便随后的load动作使用；
4. load(载入)：作用于工作内存的变量，它把read操作从主内存中得到变量值放入工作内存的变量的副本中；
5. use(使用)：作用于工作内存的变量， 它把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值的字节码指令时将会执行这个操作；
6. assign(赋值)：作用于工作内存的变量。它把一个从执行引擎接收到的值赋值给工作内存的变量，每当虚拟机遇到需要给一个变量赋值的字节码时执行这个操作；
7. store(存储)：作用于工作内存的变量。它把一个工作内存中一个变量的值传递到主内存中，以便随后的write操作使用；
8. write(写入)：作用于主内存的变量。它把store操作从工作内存中得到的变量的值放入主内存的变量中。

如果要把一个变量从工作内存复制到工作内存，那就要按顺序执行 read 和 load 操作，如果要把变量从工作内存同步回主内存，就要按顺序执行 store 和 write 操作。

上诉 8 种基本操作必须满足的规则：

1. 不允许 read 和 load、store 和 write 操作之一单独出现；
2. 不允许一个线程丢弃它的最近的 assign 操作，即变量在工作内存中改变之后必须把该变化同步回主内存；
3. 不允许一个线程无原因地（没有发生过任何 assign 操作）把数据从线程的工作内存同步回主内存中；
4. 一个新的变量只能在主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化（load 或 assign）的变量，换句话说就是对一个变量实施 use 和 store 操作之前，必须执行过了 assign 和 load 操作；
5. 一个变量在同一时刻只允许一条线程对其进行 lock 操作，但 lock 操作可以被同一线程重复执行多次，多次执行 lock 后，只有执行相同次数的 unlock，变量才会被解锁；
6. 如果对一个变量执行 lock 操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行 load 或 assign 操作初始化变量的值；
7. 如果一个变量事先没有被 lock 操作锁定，则不允许对它执行 unlock 操作，也不允许去 unlock 一个被其他线程锁定主的变量；
8. 对一个变量执行 unlock 操作之前，必须先把此变量同步回主内存中（执行 store 和 write 操作）。

Spring

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**Java面试题**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**Java面试题**」，即可获取最新版 PDF。

另外，也建议大家来网站读，因为如果有错误，我会及时在网站更改，PDF 很难及时更新。

在线网站：https://www.iamshuaidi.com

1、使用 Spring 框架的好处是什么?

1. 轻量：Spring 是轻量的，基本的版本大约 2MB。
2. 控制反转：Spring 通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们。
3. 面向切面的编程(AOP)：Spring 支持面向切面的编程，并且把应用业务逻辑和系统服务分开。
4. 容器：Spring 包含并管理应用中对象的生命周期和配置。
5. MVC框架：Spring 的 Web 框架是个精心设计的框架，是 Web 框架的一个很好的替代品。
6. 事务管理：Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）。
7. 异常处理：Spring 提供方便的 API 把具体技术相关的异常（比如由 JDBC，Hibernate or JDO 抛出的）转化为一致的 unchecked 异常

2、解释下什么是 AOP?

AOP（Aspect-Oriented Programming，面向方面编程），可以说是 OOP（Object-Oriented Programing，面向对象编程）的补充和完善。OOP 引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候，OOP 则显得无能为力。也就是说，OOP 允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能。日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其他类型的代码，如：安全性、异常处理和透明的持续性也是如此。这种散布在各处的无关的代码被称为横切（cross-cutting）代码，在 OOP 设计中，它导致了大量代码的重复，而不利于各个模块的重用。

而 AOP 技术则恰恰相反，它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。AOP 代表的是一个横向的关系，如果说“对象”是一个空心的圆柱体，其中封装的是对象的属性和行为；那么面向方面编程的方法，就仿佛一把利刃，将这些空心圆柱体剖开，以获得其内部的消息。而剖开的切面，也就是所谓的“方面”了。然后它又以巧夺天功的妙手将这些剖开的切面复原，不留痕迹。

使用“横切”技术，AOP 把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，它们经常发生在核心关注点的多处，而各处都基本相似。比如：权限认证、日志、事务处理。AOP 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

3、AOP 的代理有哪几种方式?

AOP 思想的实现一般都是基于代理模式，在 Java 中一般采用 JDK 动态代理模式，但是我们都知道，JDK 动态代理模式只能代理接口而不能代理类。因此，Spring AOP 会按照下面两种情况进行切换，因为 Spring AOP 同时支持 CGLIB、ASPECTJ、JDK 动态代理。

1. 如果目标对象的实现类实现了接口，Spring AOP 将会采用 JDK 动态代理来生成 AOP 代理类；
2. 如果目标对象的实现类没有实现接口，Spring AOP 将会采用 CGLIB 来生成 AOP 代理类。不过这个选择过程对开发者完全透明、开发者也无需关心。

4、怎么实现 JDK 动态代理?

JDK 动态代理最核心的一个接口和方法如下所示：

1. java.lang.reflect 包中的 InvocationHandler 接口：

```
public interface InvocationHandler {
        public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable;
}
```

对于被代理的类的操作都会由该接口中的 invoke 方法实现，其中的参数的含义分别是：

1. proxy：被代理的类的实例；
2. method：调用被代理的类的方法；
3. args：该方法需要的参数。

使用方法首先是需要实现该接口，并且我们可以在 invoke 方法中调用被代理类的方法并获得返回值，自然也可以在调用该方法的前后去做一些额外的事情，从而实现动态代理。

2. java.lang.reflect 包中的 Proxy 类中的 newProxyInstance 方法：

```
public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces,
InvocationHandler h) throws IllegalArgumentException
```

其中的参数含义如下：

1. loader：被代理的类的类加载器；
2. interfaces：被代理类的接口数组；
3. invocationHandler：调用处理器类的对象实例

该方法会返回一个被修改过的类的实例，从而可以自由的调用该实例的方法

5、AOP 的基本概念：切面、连接点、切入点等?

1. 切面（Aspect）：官方的抽象定义为“一个关注点的模块化，这个关注点可能会横切多个对象”。
2. 连接点（Joinpoint）：程序执行过程中的某一行为。
3. 通知（Advice）：“切面”对于某个“连接点”所产生的动作。
4. 切入点（Pointcut）：匹配连接点的断言，在 AOP 中通知和一个切入点表达式关联。
5. 目标对象（Target Object）：被一个或者多个切面所通知的对象。
6. AOP 代理（AOP Proxy）：在 Spring AOP 中有两种代理方式，JDK 动态代理和 CGLIB 代理。

6、通知类型（Advice）型（Advice）有哪些?

1. **前置通知**（Before advice）：在某连接点（JoinPoint）之前执行的通知，但这个通知不能阻止连接点前的执行。ApplicationContext 中在 aop:aspect 里面使用 aop:before 元素进行声明；
2. **后置通知**（After advice）：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。ApplicationContext 中在 aop:aspect 里面使用 aop:after 元素进行声明。
3. **返回后通知**（After return advice：在某连接点正常完成后执行的通知，不包括抛出异常的情况。ApplicationContext 中在 aop:aspect 里面使用 <> 元素进行声明。
4. **环绕通知**（Around advice）：包围一个连接点的通知，类似 Web 中 Servlet规范中的 Filter 的 doFilter 方法。可以在方法的调用前后完成自定义的行为，也可以选择不执行。ApplicationContext 中在 aop:aspect 里面使用 aop:around 元素进行声明。
5. **抛出异常后通知**（After throwing advice）：在方法抛出异常退出时执行的通知。ApplicationContext 中在 aop:aspect 里面使用 aop:after-throwing 元素进行声明。

7、谈谈你对 IOC 的理解?

IOC 是 Inversion of Control 的缩写，多数书籍翻译成“控制反转”。简单来说就是把复杂系统分解成相互合作的对象，这些对象类通过封装以后，内部实现对外部是透明的，从而降低了解决问题的复杂度，而且可以灵活地被重用和扩展。IOC 理论提出的观点大体是这样的：借助于“第三方”实现具有依赖关系的对象之间的解耦。如下图：

由于引进了中间位置的“第三方”，也就是 IOC 容器，使得 A、B、C、D 这 4 个对象没有了耦合关系，齿轮之间的传动全部依靠“第三方”了，全部对象的控制权全部上缴给“第三方”IOC 容器，所以，IOC 容器成了整个系统的关键核心，它起到了一种类似“粘合剂”的作用，把系统中的所有对象粘合在一起发挥作用，如果没有这个“粘合剂”，对象与对象之间会彼此失去联系，这就是有人把 IOC 容器比喻成“粘合剂”的由来。

把上图中间的 IOC 容器拿掉，然后再来看看这套系统：

现在看到的画面，就是我们要实现整个系统所需要完成的全部内容。这时候，A、B、C、D 这 4 个对象之间已经没有了耦合关系，彼此毫无联系，这样的话，当你在实现 A 的时候，根本无须再去考虑 B、C 和 D了，对象之间的依赖关系已经降低到了最低程度。所以，如果真能实现 IOC 容器，对于系统开发而言，这将是一件多么美好的事情，参与开发的每一成员只要实现自己的类就可以了，跟别人没有任何关系！

我们再来看看，控制反转(IOC)到底为什么要起这么个名字？我们来对比一下：

软件系统在没有引入 IOC 容器之前，对象 A 依赖于对象 B，那么对象 A 在初始化或者运行到某一点的时候，自己必须主动去创建对象 B 或者使用已经创建的对象 B。无论是创建还是使用对象 B，控制权都在自己手上。

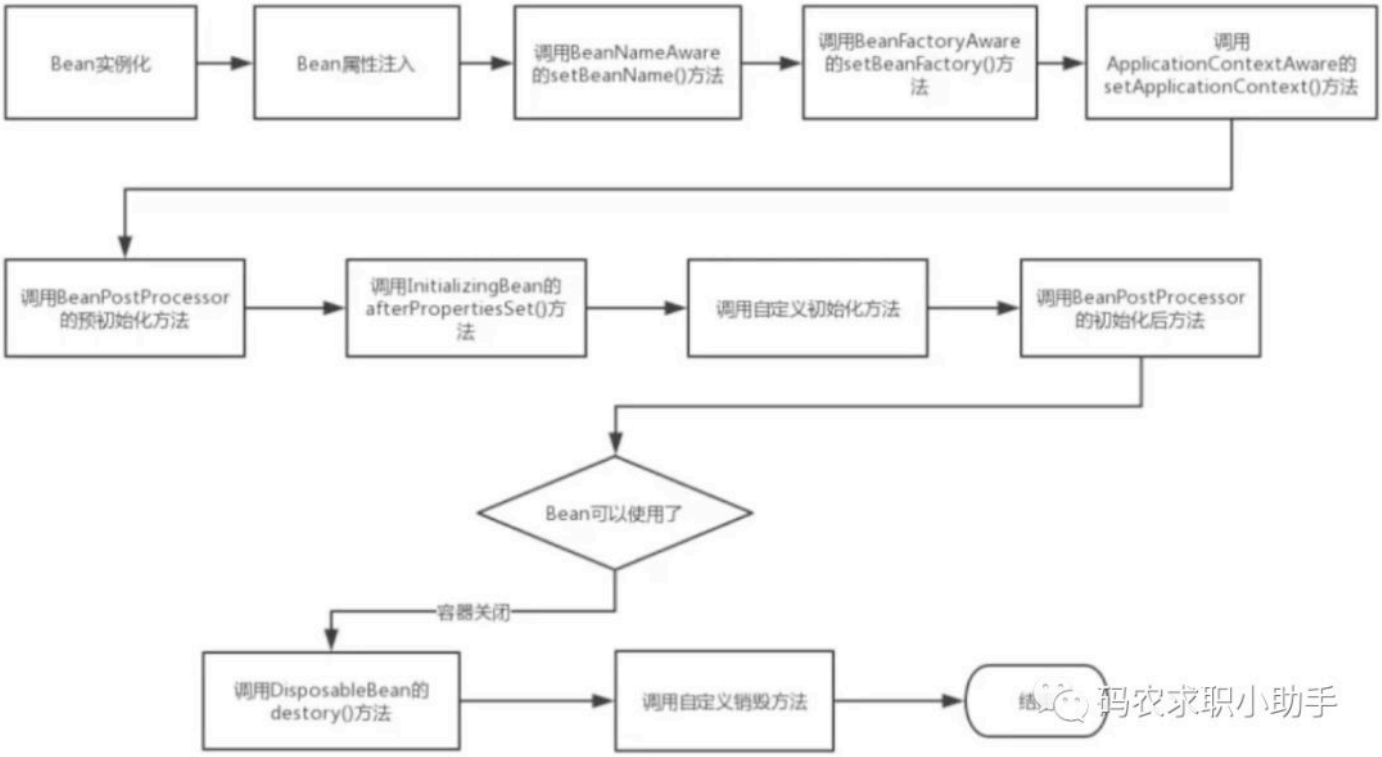
软件系统在引入 IOC 容器之后，这种情形就完全改变了，由于 IOC 容器的加入，对象 A 与对象 B 之间失去了直接联系，所以，当对象 A 运行到需要对象 B 的时候，IOC 容器会主动创建一个对象 B 注入到对象 A 需要的地方。

通过前后的对比，我们不难看出来：对象 A 获得依赖对象 B 的过程，由主动行为变为了被动行为，控制权颠倒过来了，这就是“控制反转”这个名称的由来。

8、Bean 的生命周期?

在传统的 Java 应用中，bean 的生命周期很简单，使用 Java 关键字 new 进行 Bean 的实例化，然后该 Bean 就能够使用了。一旦 Bean 不再被使用，则由 Java 自动进行垃圾回收。

相比之下，Spring 管理 Bean 的生命周期就复杂多了，正确理解 Bean 的生命周期非常重要，因为 Spring 对 Bean 的管理可扩展性非常强，下面展示了一个 Bean 的构造过程：



1. Spring 启动，查找并加载需要被 Spring 管理的 Bean，进行 Bean 的实例化；
2. Bean 实例化后，对 Bean 的引入和值注入到 Bean 的属性中；
3. 如果 Bean 实现了 BeanNameAware 接口的话，Spring 将 Bean 的 Id 传递给 setBeanName() 方法；
4. 如果 Bean 实现了 BeanFactoryAware 接口的话，Spring 将调用 setBeanFactory() 方法，将 BeanFactory 容器实例传入；
5. 如果 Bean 实现了 ApplicationContextAware 接口的话，Spring 将调用 Bean 的 setApplicationContext() 方法，将 Bean 所在应用上下文引用传入进来；
6. 如果 Bean 实现了 BeanPostProcessor 接口，Spring 就将调用它们的 postProcessBeforeInitialization() 方法；
7. 如果 Bean 实现了 InitializingBean 接口，Spring 将调用它们的 afterPropertiesSet() 方法。类似地，如果 Bean 使用 init-method 声明了初始化方法，该方法也会被调用；
8. 如果 Bean 实现了 BeanPostProcessor 接口，Spring 就将调用它们的 postProcessAfterInitialization() 方法；
9. 此时，Bean 已经准备就绪，可以被应用程序使用了。它们将一直驻留在应用上下文中，直到应用上下文被销毁；
10. 如果 Bean 实现了 DisposableBean 接口，Spring 将调用它的 destory() 接口方法，同样，如果 Bean 使用了 destory-method 声明销毁方法，该方法也会被调用。

9、Bean 的作用域?

1. singleton : 唯一 bean 实例，Spring 中的 bean 默认都是单例的；
2. prototype : 每次请求都会创建一个新的 bean 实例；
3. request：每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP request 内有效；
4. session : 每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP session 内有效；
5. global-session：全局 session 作用域，仅仅在基于 portlet 的 web 应用中才有意义，Spring5 已经没有了。Portlet 是能够生成语义代码(例如：HTML)片段的小型 Java Web 插件。它们基于 portlet 容器，可以像 servlet 一样处理 HTTP 请求。但是，与 servlet 不同，每个 portlet 都有不同的会话。

10、Spring 中的单例 Bean 的线程安全问题了解吗?

大部分时候我们并没有在系统中使用多线程，所以很少有人会关注这个问题。单例 bean 存在线程问题，主要是因为：当多个线程操作同一个对象的时候，对这个对象的非静态成员变量的写操作会存在线程安全问题。常见的有两种解决办法：

1. 在 Bean 对象中尽量避免定义可变的成员变量（不太现实）。
2. 在类中定义一个 ThreadLocal 成员变量，将需要的可变成员变量保存在 ThreadLocal 中（推荐的一种方式）。

11、谈谈你对 Spring 中的事物的理解?

事务是逻辑上的一组操作，要么都执行，要么都不执行。

事务特性

原子性：事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；

一致性：执行事务前后，数据保持一致；

隔离性：并发访问数据库时，一个用户的事物不被其他事物所干扰，各并发事务之间数据库是独立的；

持久性: 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

Spring 事务管理接口

1. PlatformTransactionManager：（平台）事务管理器；
2. TransactionDefinition：事务定义信息（事务隔离级别、传播行为、超时、只读、回滚规则）；
3. TransactionStatus：事务运行状态；

所谓事务管理，其实就是“按照给定的事务规则来执行提交或者回滚操作”。

12、Spring 中的事务隔离级别?

TransactionDefinition 接口中定义了五个表示隔离级别的常量：

TransactionDefinition.ISOLATION_DEFAULT：使用后端数据库默认的隔离级别，MySQL 默认采用的 REPEATABLE_READ 隔离级别 Oracle 默认采用的 READ_COMMITTED 隔离级别；

TransactionDefinition.ISOLATION_READ_UNCOMMITTED：最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读；

TransactionDefinition.ISOLATION_READ_COMMITTED：允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生；

TransactionDefinition.ISOLATION_REPEATABLE_READ：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生；

TransactionDefinition.ISOLATION_SERIALIZABLE：最高的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

13、Spring 中的事物传播行为?

事务传播行为是为了解决业务层方法之间互相调用的事务问题。当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。在 TransactionDefinition 定义中包括了如下几个表示传播行为的常量：

- **支持当前事务的情况：**

TransactionDefinition.PROPAGATION_REQUIRED：如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务；

TransactionDefinition.PROPAGATION_SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行；

TransactionDefinition.PROPAGATION_MANDATORY：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。

- **不支持当前事务的情况：**

TransactionDefinition.PROPAGATION_REQUIRES_NEW：创建一个新的事务，如果当前存在事务，则把当前事务挂起；

TransactionDefinition.PROPAGATION_NOT_SUPPORTED：以非事务方式运行，如果当前存在事务，则把当前事务挂起。

TransactionDefinition.PROPAGATION_NEVER：以非事务方式运行，如果当前存在事务，则抛出异常。

- **其他情况：**

TransactionDefinition.PROPAGATION_NESTED：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于 TransactionDefinition.PROPAGATION_REQUIRED。

14、Spring 常用的注入方式有哪些?

1. 构造器依赖注入：构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每个参数代表一个对其他类的依赖。
2. Setter 方法注入：Setter 方法注入是容器通过调用无参构造器或无参 static 工厂方法实例化 bean 之后，调用该 bean 的 Setter 方法，即实现了基于 Setter 的依赖注入。
3. 基于注解的注入：最好的解决方案是用构造器参数实现强制依赖，Setter 方法实现可选依赖。

15、Spring 框架中用到了哪些设计模式?

1. 工厂设计模式 : Spring 使用工厂模式通过 BeanFactory、ApplicationContext 创建 bean 对象；
2. 代理设计模式 : Spring AOP 功能的实现；
3. 单例设计模式 : Spring 中的 Bean 默认都是单例的；
4. 模板方法模式 : Spring 中 jdbcTemplate、hibernateTemplate 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式；
5. 包装器设计模式 : 我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。这种模式让我们可以根据客户的需求能够动态切换不同的数据源；
6. 观察者模式：Spring 事件驱动模型就是观察者模式很经典的一个应用；
7. 适配器模式：Spring AOP 的增强或通知(Advice)使用到了适配器模式、SpringMVC 中也是用到了适配器模式适配 Controller。

16、ApplicationContext 通常的实现有哪些?

1. FileSystemXmlApplicationContext：此容器从一个 XML 文件中加载beans 的定义，XML Bean 配置文件的全路径名必须提供给它的构造函数。
2. ClassPathXmlApplicationContext：此容器也从一个 XML 文件中加载beans 的定义，这里，你需要正确设置 classpath 因为这个容器将在 classpath 里找 bean 配置。
3. WebXmlApplicationContext：此容器加载一个 XML 文件，此文件定义了一个 Web 应用的所有 bean。

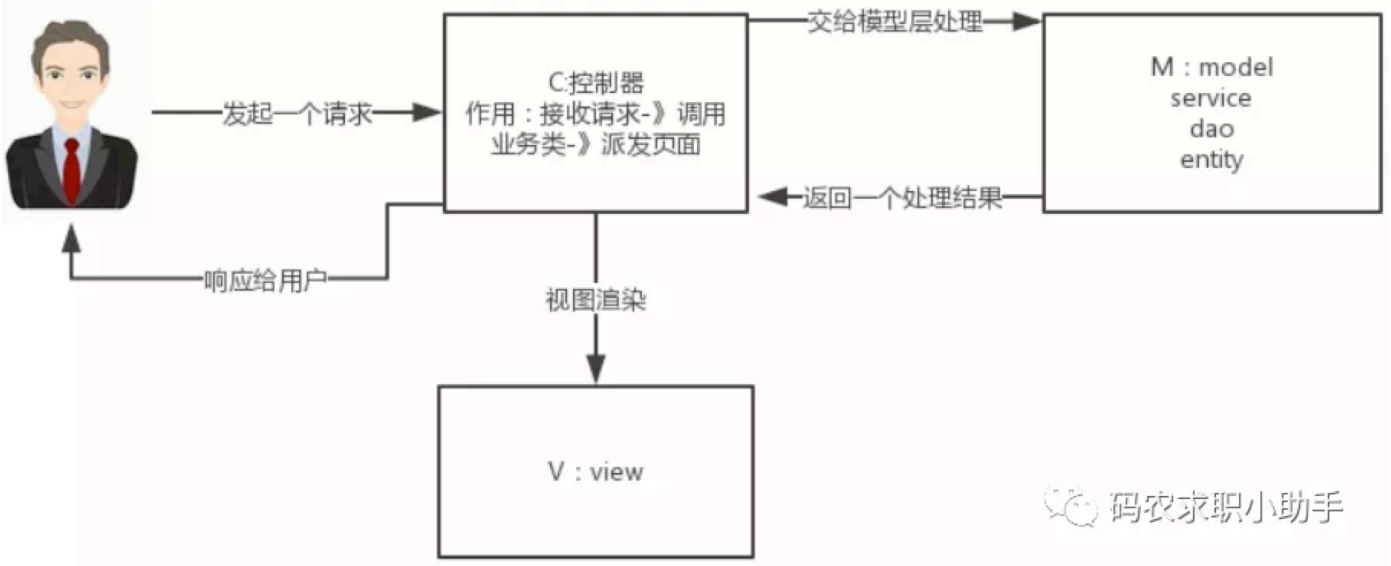
SpringMVC

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**Java面试题**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**Java面试题**」，即可获取最新版 PDF。

1、谈谈你对 MVC 模式的理解?

MVC 是 Model — View — Controler 的简称，它是一种架构模式，它分离了表现与交互。它被分为三个核心部件：模型、视图、控制器。



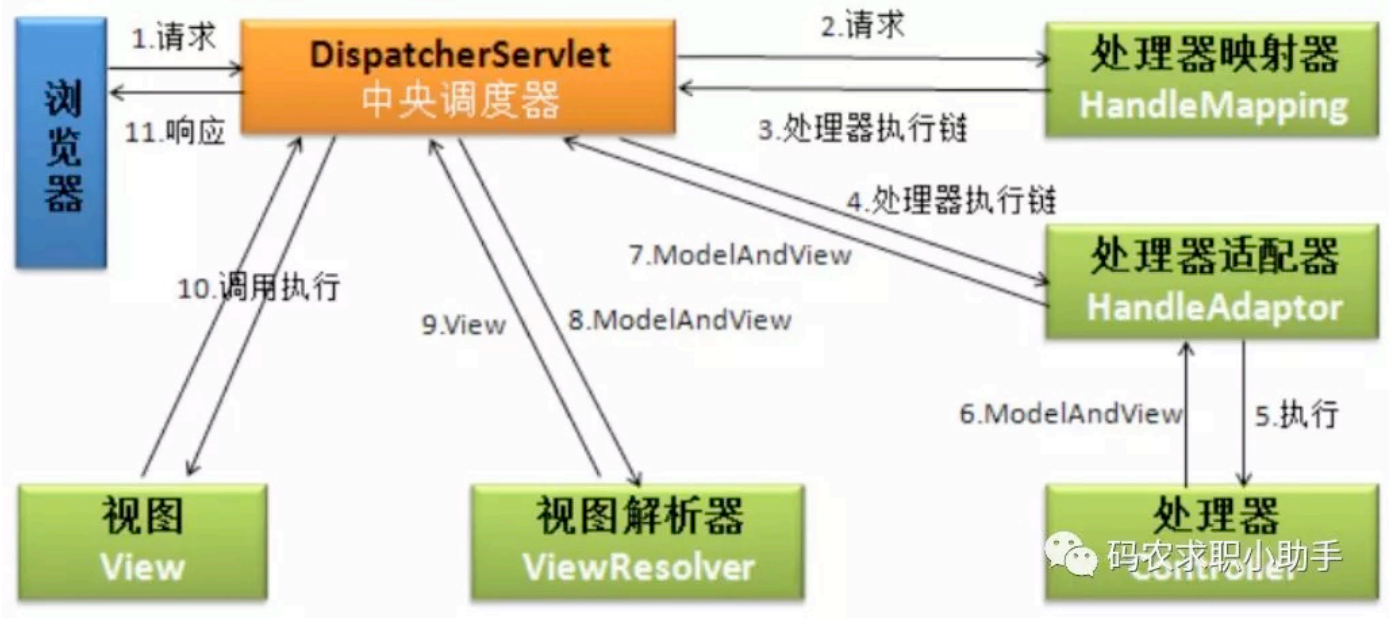
Model（模型）：是程序的主体部分，主要包含业务数据和业务逻辑。在模型层，还会涉及到用户发布的服务，在服务中会根据不同的业务需求，更新业务模型中的数据。

View（视图）：是程序呈现给用户的部分，是用户和程序交互的接口，用户会根据具体的业务需求，在 View 视图层输入自己特定的业务数据，并通过界面的事件交互，将对应的输入参数提交给后台控制器进行处理。

Controller（控制器）：Controller 是用来处理用户输入数据，以及更新业务模型的部分。控制器中接收了用户与界面交互时传递过来的数据，并根据数据业务逻辑来执行服务的调用和更新业务模型的数据和状态。

2、SpringMVC 的工作原理/执行流程?

简单来说：客户端发送请求-> 前端控制器 DispatcherServlet 接受客户端请求 -> 找到处理器映射 HandlerMapping 解析请求对应的 Handler -> HandlerAdapter 会根据 Handler 来调用真正的处理器来处理请求，并处理相应的业务逻辑 -> 处理器返回一个模型视图 ModelAndView -> 视图解析器进行解析 -> 返回一个视图对象 -> 前端控制器 DispatcherServlet 渲染数据（Model）-> 将得到视图对象返回给用户。



上图用于辅助理解，面试时可用下列 8 步描述 SpringMVC 运行流程：

1. 用户向服务器发送请求，请求被 Spring 前端控制Servelt DispatcherServlet 捕获；
2. DispatcherServlet 对请求 URL 进行解析，得到请求资源标识符（URI）。然后根据该 URI，调用

HandlerMapping 获得该 Handler 配置的所有相关的对象（包括 Handler 对象以及 Handler 对象对应的拦截器），最后以 HandlerExecutionChain 对象的形式返回；

3. DispatcherServlet 根据获得的 Handler，选择一个合适的HandlerAdapter；（附注：如果成功获得 HandlerAdapter 后，此时将开始执行拦截器的 preHandler(...)方法）
4. 提取 Request 中的模型数据，填充 Handler 入参，开始执行Handler（Controller)。在填充 Handler 的入参过程中，根据你的配置，Spring 将帮你做一些额外的工作：

（1）HttpMessageConveter：将请求消息（如：Json、xml 等数据）转换成一个对象，将对象转换为指定的响应信息；

（2）数据转换：对请求消息进行数据转换。如：String 转换成 Integer、Double 等；

（3）数据格式化：对请求消息进行数据格式化。如：将字符串转换成格式化数字或格式化日期等；

（4）数据验证：验证数据的有效性（长度、格式等），验证结果存储到 BindingResult 或 Error 中;

5. Handler 执行完成后，向 DispatcherServlet 返回一个 ModelAndView 对象；
6. 根据返回的 ModelAndView，选择一个适合的 ViewResolver（必须是已经注册到 Spring 容器中的 ViewResolver)返回给DispatcherServlet；
7. ViewResolver 结合 Model 和 View，来渲染视图；
8. 将渲染结果返回给客户端。

3、SpringMVC 的核心组件有哪些?

- **1. 前端控制器 DispatcherServlet**

作用：Spring MVC 的入口函数。接收请求，响应结果，相当于转发器，中央处理器。有了 DispatcherServlet 减少了其它组件之间的耦合度。用户请求到达前端控制器，它就相当于 MVC 模式中的 C，DispatcherServlet 是整个流程控制的中心，由它调用其它组件处理用户的请求，DispatcherServlet 的存在降低了组件之间的耦合性。

- **2. 处理器映射器 HandlerMapping**

作用：根据请求的 url 查找 Handler。HandlerMapping 负责根据用户请求找到 Handler 即处理器（Controller），SpringMVC 提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

- **3. 处理器适配器 HandlerAdapter**

作用：按照特定规则（HandlerAdapter 要求的规则）去执行 Handler。通过 HandlerAdapter 对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

- **4. 处理器 Handler**

注意：编写 Handler 时按照 HandlerAdapter 的要求去做，这样适配器才可以去正确执行 Handler。Handler 是继 DispatcherServlet 前端控制器的后端控制器，在 DispatcherServlet 的控制下 Handler 对具体的用户请求进行处理。由于 Handler 涉及到具体的用户业务请求，所以一般情况需要工程师根据业务需求开发 Handler。

- **5. 视图解析器 View resolver**

作用：进行视图解析，根据逻辑视图名解析成真正的视图（View ）。View Resolver 负责将处理结果生成 View 视图，View Resolver 首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成 View 视图对象，最后对 View 进行渲染将处理结果通过页面展示给用户。SpringMVC 框架提供了很多的 View 视图类型，包括：jstlView、freemarkerView、pdfView 等。一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由工程师根据业务需求开发具体的页面。

- **6. 视图 View**

View 是一个接口，实现类支持不同的 View 类型（jsp、freemarker...）。

> 注意：处理器 Handler（也就是我们平常说的 Controller 控制器）以及视图层 View 都是需要我们自己手动开发的。其他的一些组件比如：前端控制器 DispatcherServlet、处理器映射器 HandlerMapping、处理器适配器 HandlerAdapter 等等都是框架提供给我们的，不需要自己手动开发。

4、SpringMVC 常用的注解有哪些?

1. @RequestMapping：用于处理请求 url 映射的注解，可用于类或方法上。用于类上，则表示类中的所有响应请求的方法都是以该地址作为父路径；
2. @RequestBody：注解实现接收 HTTP 请求的 json 数据，将 json 转换为 Java 对象；
3. @ResponseBody：注解实现将 Controller 方法返回对象转化为 json 对象响应给客户。

5、@RequestMapping 的作用是什么?

RequestMapping 是一个用来处理请求地址映射的注解，可用于类或方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。RequestMapping 注解有六个属性，下面我们把它分成三类进行说明。

value、method：

1. value：指定请求的实际地址，指定的地址可以是 URI Template 模式；
2. method：指定请求的method类型， GET、POST、PUT、DELETE 等；

consumes、produces：

1. consumes：指定处理请求的提交内容类型（Content-Type），例如 application/json、text/html；
2. produces：指定返回的内容类型，仅当 request 请求头中的（Accept）类型中包含该指定类型才返回；

params、header：

1. params：指定 request 中必须包含某些参数值是，才让该方法处理。
2. headers：指定 request 中必须包含某些指定的 header 值，才能让该方法处理请求。

6、如何解决 POST 请求中文乱码问题，GET 的又如何处理呢?

1.解决 POST 请求乱码问题：在 web.xml 中配置一个 CharacterEncodingFilter 过滤器，设置成 utf-8；

2. GET 请求中文参数出现乱码解决方法有两个：

（1）修改 tomcat 配置文件添加编码与工程编码一致，如下：

```
<ConnectorURIEncoding="utf-8" connectionTimeout="20000" port="8080" protocol="HTTP/1.1"
redirectPort
```

（2）对参数进行重新编码：

```
String userName = new String(request.getParamter("userName").getBytes("ISO8859-
1"),"utf-8")
```

7、SpringMVC 的控制器是不是单例模式，如果是会有什么问题，怎么解决?

是单例模式，所以在多线程访问的时候有线程安全问题。但是不要使用同步，会影响性能，解决方案是在控制器里面不能写字段。

8、SpringMVC 怎么样设定重定向和转发的?

1. 转发：在返回值前面加 "forward:"，譬如：

```
"forward:user.do?name=method2"
```

2. 重定向：在返回值前面加 "redirect:"，譬如：

```
2. 重定向：在返回值前面加 "redirect:"，譬如：
```

9、SpringMVC 里面拦截器是怎么写的?

方法一：实现 HandlerInterceptor 接口；

方法二：继承适配器类，接着在接口方法当中，实现处理逻辑，然后在 SpringMVC 的配置文件中配置拦截器即可。

10、SpringMVC 和 Struts2 的区别有哪些?

1. SpringMVC 的入口是一个 Servlet 即前端控制器（DispatchServlet），而 Struts2 入口是一个 filter 过虑器（StrutsPrepareAndExecuteFilter）；
2. SpringMVC 是基于方法开发（一个 url 对应一个方法），请求参数传递到方法的形参，可以设计为单例或多例（建议单例），Struts2 是基于类开发，传递参数是通过类的属性，只能设计为多例；
3. Struts2 采用值栈存储请求和响应的数据，通过 OGNL 存取数据；SpringMVC 通过参数解析器是将 request 请求内容解析，并给方法形参赋值，将数据和视图封装成 ModelAndView 对象，最后又将 ModelAndView 中的模型数据通过 request 域传输到页面。jsp 视图解析器默认使用 jstl。

MyBatis

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**Java面试题**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**Java面试题**」，即可获取最新版 PDF。

1、谈谈你对 MyBatis 的理解?

1. Mybatis是一个半ORM（对象关系映射）框架，它内部封装了 JDBC，开发时只需要关注 SQL 语句本身，不需要花费精力去处理加载驱动、创建连接、创建 Statement 等繁杂的过程。程序员直接编写原生态 SQL，可以严格控制 SQL 执行性能，灵活度高。
2. MyBatis 可以使用 XML 或注解来配置和映射原生信息，将 POJO 映射成数据库中的记录，避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。
3. 通过 XML 文件或注解的方式将要执行的各种 Statement 配置起来，并通过 Java 对象和 Statement 中 SQL 的动态参数进行映射生成最终执行的 SQL 语句，最后由 MyBatis 框架执行 SQL并将结果映射为 Java 对象并返回。（从执行 SQL到返回 Result 的过程）。

2、MyBaits 的优缺点有哪些?

优点：

1. 基于 SQL 语句编程，相当灵活，不会对应用程序或者数据库的现有设计造成任何影响，SQL 写在 XML 里，解除 SQL 与程序代码的耦合，便于统一管理；提供XML标签，支持编写动态 SQL 语句，并可重用；
2. 与 JDBC 相比，减少了代码量，消除了 JDBC 大量冗余的代码，不需要手动开关连接；
3. 很好的与各种数据库兼容（因为 MyBatis 使用 JDBC 来连接数据库，所以只要 JDBC 支持的数据库 MyBatis 都支持）；
4. 提供映射标签，支持对象与数据库的 ORM 字段关系映射；提供对象关系映射标签，支持对象关系组件维护。

缺点：

1. SQL 语句的编写工作量较大，尤其当字段多、关联表多时，对开发人员编写 SQL 语句的功底有一定要求；
2. SQL 语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。

3、MyBatis 与 Hibernate 有哪些不同?

1. MyBatis 和 Hibernate不同，它不完全是一个 ORM 框架，因为 MyBatis 需要程序员自己编写 SQL 语句；Hibernate 对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件，如果用 Hibernate 开发可以节省很多代码，提高效率；
2. MyBatis 直接编写原生态 SQL，可以严格控制 SQL 执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，因为这类软件需求变化频繁，一但需求变化要求迅速输出成果。但是灵活的前提是 MyBatis 无

法做到数据库无关性，如果需要实现支持多种数据库的软件，则需要自定义多套 SQL 映射文件，工作量大。

4、MyBatis 中 #{} 和 ${}的区别是什么?

#{} 是预编译处理，${} 是字符串替换

1. Mybatis 在处理 #{} 时，会将 SQL 中的 #{} 替换为 ? 号，调用 PreparedStatement 的 set 方法来赋值；使用 #{} 可以有效的防止 SQL 注入，提高系统安全性；
2. MyBatis 在处理 时，就是把{} 替换成变量的值。

5、MyBatis 是如何进行分页的？分页插件的原理是什么?

MyBatis 使用 RowBounds 对象进行分页，它是针对 ResultSet 结果集执行的内存分页，而非物理分页。可以在 SQL 内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。

分页插件的基本原理是使用 MyBatis 提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的 SQL，然后重写 SQL，根据 dialect 方言，添加对应的物理分页语句和物理分页参数。

6、MyBatis 有几种分页方式?

1. 数组分页
2. SQL 分页
3. 拦截器分页
4. RowBounds 分页

7、MyBatis 逻辑分页和物理分页的区别是什么?

1. 物理分页速度上并不一定快于逻辑分页，逻辑分页速度上也并不一定快于物理分页。
2. 物理分页总是优于逻辑分页：没有必要将属于数据库端的压力加到应用端来，就算速度上存在优势，然而其它性能上的优点足以弥补这个缺点。

8、MyBatis 是否支持延迟加载？如果支持，它的实现原理是什么?

Mybatis 仅支持 association 关联对象和 collection 关联集合对象的延迟加载，association 指的就是一对一，collection 指的就是一对多查询。在MyBatis配置文件中，可以配置是否启用延迟加载 lazyLoadingEnabled=true|false。

它的原理是，使用 CGLIB 创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用 a.getB().getName()，拦截器 invoke() 方法发现 a.getB() 是 null 值，那么就会单独发送事先保存好的查询关联 B 对象的 SQL，把 B 查询上来，然后调用 a.setB(b)，于是 a 的对象 b 属性就有值了，接着完成 a.getB().getName() 方法的调用。这就是延迟加载的基本原理。

9、说一下 MyBatis 的一级缓存和二级缓存?

一级缓存：基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空，默认打开一级缓存；

二级缓存：与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现 Serializable 序列化接口(可用来保存对象的状态)，可在它的映射文件中配置；

对于缓存数据更新机制，当某一个作用域(一级缓存 Session / 二级缓存 Namespaces)的进行了 C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear。

10、Mybatis 有哪些执行器（Executor）？

Mybatis 有 3 种基本的执行器（Executor）：

1. SimpleExecutor：每执行一次 update 或 select，就开启一个 Statement 对象，用完立刻关闭 Statement 对象；
2. ReuseExecutor：执行 update 或 select，以 SQL 作为 key 查找 Statement 对象，存在就使用，不存在就创建，用完后，不关闭 Statement 对象，而是放置于 Map 内，供下一次使用。简言之，就是重复使用 Statement 对象；
3. BatchExecutor：执行 update（没有 select，JDBC 批处理不支持select），将所有 SQL 都添加到批处理中（addBatch()），等待统一执行（executeBatch()），它缓存了多个 Statement 对象，每个 Statement对 象都是 addBatch() 完毕后，等待逐一执行 executeBatch() 批处理。与 JDBC 批处理相同。

11、MyBatis 动态 SQL 是做什么的？都有哪些动态 SQL？能简述一下动态 SQL的执行原理不？

1. MyBatis 动态 SQL 可以让我们在 XML 映射文件内，以标签的形式编写动态 SQL，完成逻辑判断和动态拼接 SQL 的功能；
2. MyBatis 提供了 9 种动态 SQL 标签：trim、where、set、foreach、if、choose、when、otherwise、bind；
3. 执行原理：使用 OGNL 从 SQL 参数对象中计算表达式的值，根据表达式的值动态拼接 SQL，以此来完成动态 SQL 的功能。

SpringBoot

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**Java面试题**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**Java面试题**」，即可获取最新版 PDF。

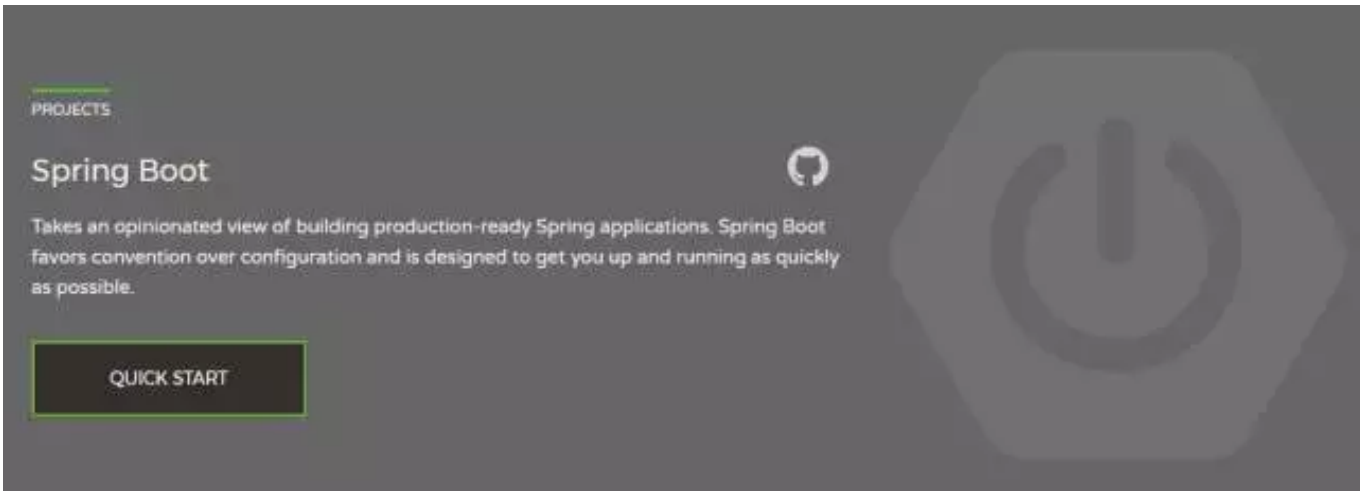
1、什么是 Spring Boot?

快问快答：Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器，开发者能快速上手。

Spring Boot是Spring开源组织下的子项目，是Spring组件一站式解决方案，主要是简化了使用Spring的难度，简省了繁重的配置，提供了各种启动器，开发者能快速上手。

官方网站：http://projects.spring.io/spring-boot/

GitHub源码：https://github.com/spring-projects/spring-boot



2、SpringBoot有哪些优点?

- 独立运行 Spring Boot 而且内嵌了各种 servlet 容器，Tomcat、Jetty 等，现在不再需要打成war 包部署到容器中，Spring Boot 只要打成一个可执行的 jar 包就能独立运行，所有的依赖包都在一个 jar 包内。
- 简化配置 spring-boot-starter-web 启动器自动依赖其他组件，简少了 maven 的配置。
- 自动配置 Spring Boot 能根据当前类路径下的类、jar 包来自动配置 bean，如添加一个 spring-boot-starter-web 启动器就能拥有 web 的功能，无需其他配置。
- 无代码生成和XML配置 Spring Boot 配置过程中无代码生成，也无需 XML 配置文件就能完成所有配置工作，这一切都是借助于条件注解完成的，这也是 Spring4.x 的核心功能之一。
- 避免大量的Maven导入和各种版本冲突
- 应用监控 Spring Boot 提供一系列端点可以监控服务及应用，做健康检测。

3、SpringBoot有哪些缺点?

任何技术都是有优缺点的，没有银弹，解决一切问题，不留任何小尾巴

SpringBoot优点概括起来就是简化：简化编码，简化配置，简化部署，简化监控，简化依赖坐标导入，简化整合其他技术....

SpringBoot的**缺点**是入门简单精通难，各种强大的功能封装的太好了，内部原理比较难得参透！再就是用多了容易产生依赖，就像嗑药似的，用了就离不开了；SpringBoot一旦出了错误，由于内部封装比较深，部分错误调试难度比一般Spring应用程序要大很多！

当然完全不必纠结与SpringBoot的缺点，毕竟SpringBoot的优点太突出了。

4、Spring Boot 的核心配置文件有哪几个？它们的区别是什么?

Spring Boot 的核心配置文件是 application 和 bootstrap 配置文件。

application 配置文件这个容易理解，主要用于 Spring Boot 项目的自动化配置。

bootstrap 配置文件有以下几个应用场景。

- 使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中添加连接到配置中心的配置属性来加载外部配置中心的配置信息；
- 一些固定的不能被覆盖的属性；
- 一些加密/解密的场景；

5、Spring Boot 的配置文件有哪几种格式？它们有什么区别?

.properties 和 .yml，它们的区别主要是书写格式不同。

1).properties

app.user.name = javastack

2).yml

app: user: name: javastack

另外，.yml 格式不支持 `@PropertySource` 注解导入配置。

6、Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的?

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
      @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
      @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class)
})
public @interface SpringBootApplication {
```

主要组合包含了以下 3 个注解：

- **@SpringBootConfiguration**： 组合了 @Configuration 注解，实现配置文件的功能。
- **@EnableAutoConfiguration**： 打开自动配置的功能，也可以关闭某个自动配置的选项，如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。
- **@ComponentScan**： Spring组件扫描。

7、开启 Spring Boot 特性有哪几种方式?

快问快答

1）继承spring-boot-starter-parent项目

2）导入spring-boot-dependencies项目依赖

详细介绍启动方式

Spring Boot依赖

使用Spring Boot很简单，先添加基础依赖包，有以下两种方式

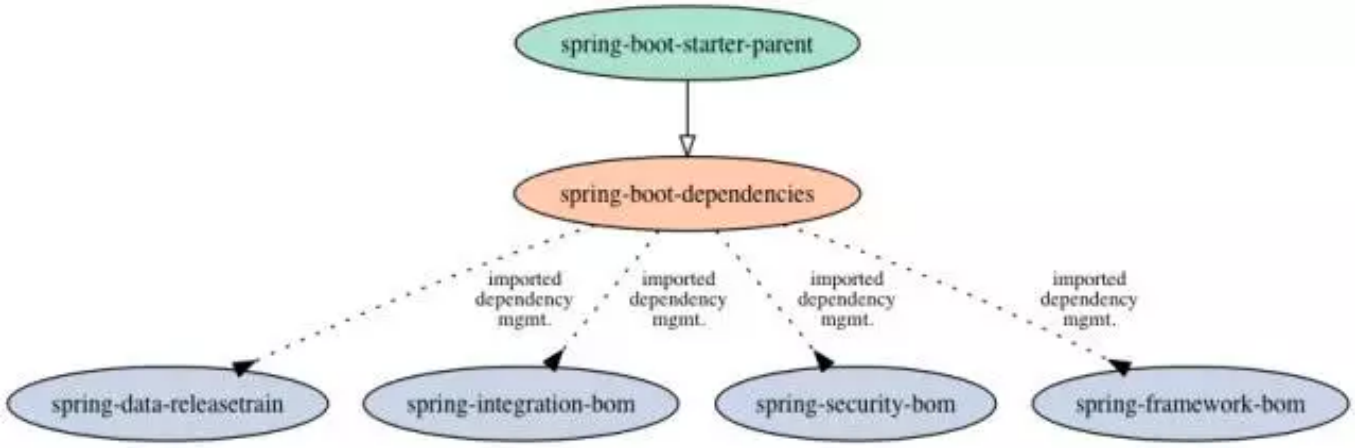
1. 继承spring-boot-starter-parent项目

1. 继承spring-boot-starter-parent项目

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.6.RELEASE</version>
</parent>
```

2. 导入spring-boot-dependencies项目依赖

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>1.5.6.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
  </dependency>
</dependencyManagement>
```



Spring Boot依赖注意点

1. 属性覆盖只对继承有效

> This only works if your Maven project inherits (directly or indirectly) from spring-boot-dependencies. If you have added spring-boot-dependencies in your own dependencyManagement section withimportyou have to redefine the artifact yourself instead of overriding the property.

Spring Boot依赖包里面的组件的版本都是和当前Spring Boot绑定的，如果要修改里面组件的版本，只需要添加如下属性覆盖即可，但这种方式只对继承有效，导入的方式无效。

```
<properties>
  <slf4j.version>1.7.25<slf4j.version>
</properties>
```

如果导入的方式要实现版本的升级，达到上面的效果，这样也可以做到，把要升级的组件依赖放到Spring Boot之前。

```
<dependencyManagement>
    <dependencies>
        <!-- Override Spring Data release train provided by Spring Boot -->
        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-releasetrain</artifactId>
            <version>Fowler-SR2</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>1.5.6.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

> Each Spring Boot release is designed and tested against a specific set of third-party dependencies. Overriding versions may cause compatibility issues.

需要注意，要修改Spring Boot的依赖组件版本可能会造成不兼容的问题。

2. 资源文件过滤问题

使用继承Spring Boot时，如果要使用Maven resource filter过滤资源文件时，资源文件里面的占位符为了使${}和Spring Boot区别开来，此时要用@...@包起来，不然无效。另外，@...@占位符在yaml文件编辑器中编译报错，所以使用继承方式有诸多问题，坑要慢慢趟。

8、什么是JavaConfig?

Spring JavaConfig 是 Spring 社区的产品，它提供了配置 Spring IoC 容器的纯Java 方法。因此它有助于避免使用 XML 配置。使用 JavaConfig 的优点在于：

- **面向对象的配置**。 由于配置被定义为 JavaConfig 中的类，因此用户可以充分利用 Java 中的面向对象功能。一个配置类可以继承另一个，重写它的@Bean 方法等。
- **减少或消除 XML 配置**。 基于依赖注入原则的外化配置的好处已被证明。但是，许多开发人员不希望在 XML 和 Java 之间来回切换。JavaConfig 为开发人员提供了一种纯 Java 方法来配置与 XML 配置概念相似的 Spring 容器。从技术角度来讲，只使用 JavaConfig 配置类来配置容器是可行的，但实际上很多人认为将JavaConfig 与 XML 混合匹配是理想的。
- **类型安全和重构友好**。 JavaConfig 提供了一种类型安全的方法来配置 Spring容器。由于 Java 5.0 对泛型的支

持，现在可以按类型而不是按名称检索 bean，不需要任何强制转换或基于字符串的查找。

9、SpringBoot自动配置原理是什么?

1. SpringBoot启动会加载大量的自动配置类
2. 我们看我们需要的功能有没有在SpringBoot默认写好的自动配置类当中；
3. 我们再来看这个自动配置类中到底配置了哪些组件；（只要我们要用的组件存在在其中，我们就不需要再手动配置了）
4. 给容器中自动配置类添加组件的时候，会从properties类中获取某些属性。我们只需要在配置文件中指定这些属性的值即可； xxxxAutoConfigurartion：自动配置类；给容器中添加组件

xxxxProperties:封装配置文件中相关属性；

10、SpringBoot、Spring MVC和Spring有什么区别?

- **Spring** Spring最重要的特征是依赖注入。所有Spring Modules不是依赖注入就是IOC控制反转。 当我们恰当的使用DI或者是IOC的时候，可以开发松耦合应用。
- **Spring MVC** Spring MVC提供了一种分离式的方法来开发Web应用。通过运用像DispatcherServelet，MoudlAndView 和 ViewResolver 等一些简单的概念，开发 Web 应用将会变的非常简单。
- **SpringBoot** Spring和Spring MVC的问题在于需要配置大量的参数。 SpringBoot通过一个自动配置和启动的项来解决这个问题。

11、SpringBoot启动时都做了什么?

1. SpringBoot在启动的时候从类路径下的META-INF/spring.factories中获取EnableAutoConfiguration指定的值
2. 将这些值作为自动配置类导入容器， 自动配置类就生效， 帮我们进行自动配置工作；
3. 整个J2EE的整体解决方案和自动配置都在springboot-autoconfigure的jar包中；
4. 它会给容器中导入非常多的自动配置类 （xxxAutoConfiguration）, 就是给容器中导入这个场景需要的所有组件， 并配置好这些组件 ；
5. 有了自动配置类， 免去了我们手动编写配置注入功能组件等的工作；

12、SpringBoot 需要独立的容器运行吗?

可以不需要，内置了 Tomcat/ Jetty 等容器。

13、什么是YAML?

YAML 是一种人类可读的数据序列化语言。它通常用于配置文件。与属性文件相比，如果我们想要在配置文件中添加复杂的属性，YAML 文件就更加结构化，而且更少混淆。可以看出 YAML 具有分层配置数据。

14、YAML 配置的优势在哪里 ?

YAML 现在可以算是非常流行的一种配置文件格式了，无论是前端还是后端，都可以见到 YAML 配置。那么 YAML 配置和传统的 properties 配置相比到底有哪些优势呢?

1. 配置有序，在一些特殊的场景下，配置有序很关键

2. 支持数组，数组中的元素可以是基本数据类型也可以是对象
3. 简洁

相比 properties 配置文件，YAML 还有一个缺点，就是不支持 @PropertySource 注解导入自定义的 YAML 配置。

15、SpringBoot 是否可以使用 XML 配置？

Spring Boot 推荐使用 Java 配置而非 XML 配置，但是 Spring Boot 中也可以使用 XML 配置，通过 @ImportResource 注解可以引入一个 XML 配置。

16、SpringBoot核心配置文件是什么?

bootstrap.properties和application.properties

17、bootstrap.properties和application.properties 有何区别？

SpringBoot两个核心的配置文件：

- **bootstrap**(.yml 或者 .properties)：boostrap 由父 ApplicationContext 加载的，比applicaton优先加载，配置在应用程序上下文的引导阶段生效。一般来说我们在 SpringCloud Config 或者Nacos中会用到它。且 boostrap里面的属性不能被覆盖；
- **application** (.yml或者.properties)：由ApplicatonContext 加载，用于 SpringBoot项目的自动化配置。

18、什么是Spring Profiles?

主要用来**区分环境**； Spring Profiles 允许用户根据配置文件（dev，test，prod 等）来注册 bean。因此，当应用程序在开发中运行时，只有某些 bean 可以加载，而在 PRODUCTION中，某些其他 bean 可以加载。假设我们的要求是 Swagger 文档仅适用于 QA 环境，并且禁用所有其他文档。这可以使用配置文件来完成。Spring Boot 使得使用配置文件非常简单。

19、如何在自定义端口上运行SpringBoot应用程序?

SpringBoot默认监听的是8080端口；为了在自定义端口上运行 SpringBoot 应用程序，您可以在 application.properties 中通过

```
server.port = 8888
```

指定端口；这样就可以将监听的端口修改为8888。

20、 如何实现SpringBoot应用程序的安全性?

为了实现SpringBoot的安全性，我们使用spring-boot-starter-security依赖项，并且必须添加安全配置。它只需要很少的代码。配置类将必须扩展WebSecurityConfigurerAdapter并覆盖其方法。

21、 比较一下Spring Security 和Shiro各自的优缺点？

由于SpringBoot官方提供了大量的非常方便的开箱即用的Starter，包括Spring Security的Starter，使得在SpringBoot中使用Spring Security变得更加容易，甚至只需要添加一个依赖就可以保护所有的接口，所以，如果是SpringBoot 项目，一般选择 Spring Security 。当然这只是一个建议的组合，单纯从技术上来说，无论怎么组合，都是没有问题的。Shiro和Spring Security相比，主要有如下一些特点：

1. Spring Security 是一个重量级的安全管理框架；Shiro 则是一个轻量级的安全管理框架
2. Spring Security 概念复杂，配置繁琐；Shiro 概念简单、配置简单
3. Spring Security 功能强大；Shiro 功能简单

22、 SpringBoot中如何解决跨域问题？

跨域可以在前端通过 JSONP 来解决，但是 JSONP 只可以发送 GET 请求，无法发送其他类型的请求，在 RESTful 风格的应用中，就显得非常鸡肋，因此我们推荐在后端通过 （CORS，Cross-origin resource sharing） 来解决跨域问题。这种解决方案并非 Spring Boot 特有的，在传统的 SSM 框架中，就可以通过 CORS 来解决跨域问题，只不过之前我们是在 XML 文件中配置 CORS ，现在可以通过实现WebMvcConfigurer接口然后重写addCorsMappings方法解决跨域问题。

```
@Configuration
public class CorsConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOrigins("*")
                .allowCredentials(true)
                .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")
                .maxAge(3600);
    }
}
```

项目中前后端分离部署，所以需要解决跨域的问题。

我们使用cookie存放用户登录的信息，在spring拦截器进行权限控制，当权限不符合时，直接返回给用户固定的json结果。

当用户登录以后，正常使用；当用户退出登录状态时或者token过期时，由于拦截器和跨域的顺序有问题，出现了跨域的现象。

我们知道一个http请求，先走filter，到达servlet后才进行拦截器的处理，如果我们把cors放在filter里，就可以优先于权限拦截器执行。

```
@Configuration
public class CorsConfig {
    @Bean
    public CorsFilter corsFilter() {
        CorsConfiguration corsConfiguration = new CorsConfiguration();
        corsConfiguration.addAllowedOrigin("*");
```

```
        corsConfiguration.addAllowedHeader("*");
        corsConfiguration.addAllowedMethod("*");
        corsConfiguration.setAllowCredentials(true);
        UrlBasedCorsConfigurationSource urlBasedCorsConfigurationSource = new
UrlBasedCorsConfigurationSource();
        urlBasedCorsConfigurationSource.registerCorsConfiguration("/**",
corsConfiguration);
        return new CorsFilter(urlBasedCorsConfigurationSource);
    }
}
```

23、 什么是 CSRF 攻击?

CSRF 代表跨站请求伪造。这是一种攻击，迫使最终用户在当前通过身份验证的Web 应用程序上执行不需要的操作。CSRF 攻击专门针对状态改变请求，而不是数据窃取，因为攻击者无法查看对伪造请求的响应。

24、 SpringBoot 中的监视器是什么

Spring boot actuator是spring启动框架中的重要功能之一。Spring boot监视器可帮助您访问生产环境中正在运行的应用程序的当前状态。有几个指标必须在生产环境中进行检查和监控。即使一些外部应用程序可能正在使用这些服务来向相关人员触发警报消息。监视器模块公开了一组可直接作为HTTPURL访问的REST端点来检查状态。

25、 如何在SpringBoot中禁用Actuator端点安全性?

默认情况下，所有敏感的HTTP端点都是安全的，只有具有ACTUATOR角色的用户才能访问它们。安全性是使用标准的 `HttpServletRequest.isUserlnRole` 方法实施的。我们可以使用 `management.security.enabled=false` 来禁用安全性。只有在执行机构端点在防火墙后访问时，才建议禁用安全性。

25、 如何监视所有SpringBoot微服务?

SpringBoot提供监视器端点以监控各个微服务的度量。这些端点对于获取有关应用程序的信息（如它们是否已启动）以及它们的组件（如数据库等）是否正常运行很有帮助。但是，使用监视器的一个主要缺点或困难是，我们必须单独打开应用程序的知识点以了解其状态或健康状况。想象一下涉及 50 个应用程序的微服务，管理员将不得不击中所有 50 个应用程序的执行终端。为了帮助我们处理这种情况，我们将使用位于的开源项目。 它建立在 Spring Boot Actuator 之上，它提供了一个 Web UI，使我们能够可视化多个应用程序的度量。

27、 什么是 WebSockets?

WebSocket是一种计算机通信协议，通过单个TCP连接提供全双工通信信道。

1. WebSocket是双向的 -使用 WebSocket 客户端或服务器可以发起消息发送。
2. WebSocket是全双工的 -客户端和服务器通信是相互独立的。
3. 单个TCP连接 -初始连接使用 HTTP，然后将此连接升级到基于套接字的连接。然后这个单一连接用于所有未来的通信
4. Light与http相比，WebSocket消息数据交换要轻得多。

28、 什么是 Spring Data?

Spring Data 是 Spring 的一个子项目。用于简化数据库访问，支持NoSQL 和 关系数据存储。其主要目标是使数据库的访问变得方便快捷。Spring Data 具有如下特点：

SpringData 项目支持 NoSQL 存储：

1. MongoDB （文档数据库）
2. Neo4j（图形数据库）
3. Redis（键/值存储）
4. Hbase（列族数据库）

SpringData 项目所支持的关系数据存储技术：

1. JDBC
2. JPA

Spring Data Jpa 致力于减少数据访问层 (DAO) 的开发量. 开发者唯一要做的，就是声明持久层的接口，其他都交给 Spring Data JPA 来帮你完成！Spring Data JPA 通过规范方法的名字，根据符合规范的名字来确定方法需要实现什么样的逻辑。

29、 什么是 Spring Batch?

Spring Boot Batch 提供可重用的函数，这些函数在处理大量记录时非常重要，包括日志/跟踪，事务管理，作业处理统计信息，作业重新启动，跳过和资源管理。它还提供了更先进的技术服务和功能，通过优化和分区技术，可以实现极高批量和高性能批处理作业。简单以及复杂的大批量批处理作业可以高度可扩展的方式利用框架处理重要大量的信息。

30、 什么是 FreeMarker 模板?

FreeMarker 是一个基于 Java 的模板引擎，最初专注于使用 MVC 软件架构进行动态网页生成。使用 Freemarker 的主要优点是表示层和业务层的完全分离。程序员可以处理应用程序代码，而设计人员可以处理 html 页面设计。最后使用freemarker 可以将这些结合起来，给出最终的输出页面。

31、 如何集成 SpringBoot和ActiveMQ?

对于集成 Spring Boot 和 ActiveMQ，我们使用依赖关系。 它只需要很少的配置，并且不需要样板代码。

32、 Swagger用过麽？他用来做什么?

Swagger广泛用于可视化API，使用SwaggerUI**为前端开发人员提供在线沙箱**。Swagger 是用于生成RESTful Web 服务的可视化表示的工具，规范和完整框架实现。它**使文档能够以与服务器相同的速度更新**。当通过Swagger 正确定义时，消费者可以使用最少量的实现逻辑来理解远程服务并与其进行交互。因此，Swagger 消除了调用服务时的猜测。

33、 前后端分离，如何维护接口文档？

前后端分离开发日益流行，大部分情况下，我们都是通过 Spring Boot 做前后端分离开发，前后端分离一定会有接口文档，不然会前后端会深深陷入到扯皮中。一个比较笨的方法就是使用 word 或者 md 来维护接口文档，但是效率太低，接口一变，所有人手上的文档都得变。在 Spring Boot 中，这个问题常见的解决方案是 Swagger ，使用 Swagger 我们可以快速生成一个接口文档网站，接口一旦发生变化，文档就会自动更新，所有开发工程师访问这一个在线网站就可以获取到最新的接口文档，非常方便。

34、 SpringBoot项目如何热部署?

这可以使用 DEV 工具来实现。通过这种依赖关系，您可以节省任何更改，嵌入式tomcat 将重新启动。Spring Boot 有一个开发工具（DevTools）模块，它有助于提高开发人员的生产力。Java 开发人员面临的一个主要挑战是将文件更改自动部署到服务器并自动重启服务器。开发人员可以重新加载 Spring Boot 上的更改，而无需重新启动服务器。这将消除每次手动部署更改的需要。Spring Boot 在发布它的第一个版本时没有这个功能。这是开发人员最需要的功能。DevTools 模块完全满足开发人员的需求。该模块将在生产环境中被禁用。它还提供 H2 数据库控制台以更好地测试应用程序。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

35、 SpringBoot 中的starter到底是什么？

首先，这个 Starter 并非什么新的技术点，基本上还是基于 Spring 已有功能来实现的。首先它提供了一个自动化配置类，一般命名为 `XXXAutoConfiguration` ，在这个配置类中通过条件注解来决定一个配置是否生效（条件注解就是 Spring 中原本就有的），然后它还会提供一系列的默认配置，也允许开发者根据实际情况自定义相关配置，然后通过类型安全的属性注入将这些配置属性注入进来，新注入的属性会代替掉默认属性。正因为如此，很多第三方框架，我们只需要引入依赖就可以直接使用了。当然，开发者也可以自定义 Starter

36、 spring-boot-starter-parent 有什么用?

新创建一个 SpringBoot 项目，默认都是有 parent 的，这个 parent 就是 spring-boot-starter-parent ，spring-boot-starter-parent 主要有如下作用：

1. 定义了 Java 编译版本为 1.8 。
2. 使用 UTF-8 格式编码。
3. 继承自 spring-boot-dependencies，这个里边定义了依赖的版本，也正是因为继承了这个依赖，所以我们在写依赖时才不需要写版本号。
4. 执行打包操作的配置。
5. 自动化的资源过滤。
6. 自动化的插件配置。
7. 针对 application.properties 和 application.yml 的资源过滤，包括通过 profile 定义的不同环境的配置文件，例如 application-dev.properties 和 application-dev.yml。

37、 SpringBoot 打成的jar和普通的jar有什么区别？

Spring oot 项目最终打包成的 jar 是可执行 jar ，这种 jar 可以直接通过 `java -jar xxx.jar` 命令来运行，这种 jar 不可以作为普通的 jar 被其他项目依赖，即使依赖了也无法使用其中的类。

SpringBoot 的 jar 无法被其他项目依赖，主要还是他和普通 jar 的结构不同。普通的 jar 包，解压后直接就是包名，包里就是我们的代码，而 Spring Boot 打包成的可执行 jar 解压后，在 `\BOOT-INF\classes` 目录下才是我们的代码，因此无法被直接引用。如果非要引用，可以在 pom.xml 文件中增加配置，将 Spring Boot 项目打包成两个 jar ，一个可执行，一个可引用。

38、 如何使用SpringBoot实现异常处理?

Spring 提供了一种使用 ControllerAdvice 处理异常的非常有用的方法。 我们通过实现一个 ControlerAdvice 类，来处理控制器类抛出的所有异常。

4.13 如何使用SpringBoot实现分页和排序?

使用Spring Boot实现分页非常简单。使用Spring Data-JPA可以实现将可分页的 `org.springframework.data.domain.Pageable` 传递给存储库方法。

```
public Page find(Integer page, Integer size) {
    if (null == page) {
        page = 0;
    }
    if (CheckUtils.isEmpty(size)) {
        size = 10;
    }
    PageRequest pageable = PageRequest.of(page, size, Sort.Direction.DESC,
&amp;quot;updateTime&amp;quot;);
    Page users = userRepository.findAll(pageable);
    return users;
}
```

40、 微服务中如何实现 session 共享?

在微服务中，一个完整的项目被拆分成多个不相同的独立的服务，各个服务独立部署在不同的服务器上，各自的 session 被从物理空间上隔离开了，但是经常，我们需要在不同微服务之间共享 session ，常见的方案就是 Spring Session + Redis 来实现 session 共享。将所有微服务的 session 统一保存在 Redis 上，当各个微服务对 session 有相关的读写操作时，都去操作 Redis 上的 session 。这样就实现了 session 共享，Spring Session 基于 Spring 中的代理过滤器实现，使得 session 的同步操作对开发人员而言是透明的，非常简便。

41、 SpringBoot 中如何实现定时任务?

定时任务也是一个常见的需求，SpringBoot 中对于定时任务的支持主要还是来自 Spring 框架。

在 SpringBoot 中使用定时任务主要有两种不同的方式，一个就是使用 Spring 中的 @Scheduled 注解，另一个则是使用第三方框架 Quartz。

- 使用Spring中的 @Scheduled的方式主要通过@Scheduled注解来实现。

- 使用Quartz，则按照Quartz的方式，定义Job和Trigger即可。

42、Spring Boot 中的 starter 到底是什么？

首先，这个 Starter 并非什么新的技术点，基本上还是基于 Spring 已有功能来实现的。首先它提供了一个自动化配置类，一般命名为 XXXAutoConfiguration ，在这个配置类中通过条件注解来决定一个配置是否生效（条件注解就是 Spring 中原本就有的），然后它还会提供一系列的默认配置，也允许开发者根据实际情况自定义相关配置，然后通过类型安全的属性注入将这些配置属性注入进来，新注入的属性会代替掉默认属性。正因为如此，很多第三方框架，我们只需要引入依赖就可以直接使用了。当然，开发者也可以自定义 Starter

43、微服务中如何实现 session 共享？

在微服务中，一个完整的项目被拆分成多个不相同的独立的服务，各个服务独立部署在不同的服务器上，各自的 session 被从物理空间上隔离开了，但是经常，我们需要在不同微服务之间共享 session ，常见的方案就是 Spring Session + Redis 来实现 session 共享。将所有微服务的 session 统一保存在 Redis 上，当各个微服务对 session 有相关的读写操作时，都去操作 Redis 上的 session 。这样就实现了 session 共享，Spring Session 基于 Spring 中的代理过滤器实现，使得 session 的同步操作对开发人员而言是透明的，非常简便。

44、Spring Boot 中如何实现定时任务？

定时任务也是一个常见的需求，Spring Boot 中对于定时任务的支持主要还是来自 Spring 框架。

在 Spring Boot 中使用定时任务主要有两种不同的方式，一个就是使用 Spring 中的 @Scheduled 注解，另一个则是使用第三方框架 Quartz。

使用 Spring 中的 @Scheduled 的方式主要通过 @Scheduled 注解来实现。

使用 Quartz ，则按照 Quartz 的方式，定义 Job 和 Trigger 即可。

计算机网络

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**Java面试题**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**Java面试题**」，即可获取最新版 PDF。

另外，也建议大家来网站读，因为如果有错误，我会及时在网站更改，PDF 很难及时更新。

在线网站：https://www.iamshuaidi.com

1、说一说三次握手

当面试官问你为什么需要有三次握手、三次握手的作用、讲讲三次三次握手的时候，我想很多人会这样回答：

首先很多人会先讲下握手的过程：

1、第一次握手：客户端给服务器发送一个 SYN 报文。

2、第二次握手：服务器收到 SYN 报文之后，会应答一个 SYN+ACK 报文。

3、第三次握手：客户端收到 SYN+ACK 报文之后，会回应一个 ACK 报文。

4、服务器收到 ACK 报文之后，三次握手建立完成。

作用是为了确认双方的接收与发送能力是否正常。

> **这里我顺便解释一下为啥只有三次握手才能确认双方的接受与发送能力是否正常，而两次却不可以**：第一次握手：客户端发送网络包，服务端收到了。这样服务端就能得出结论：客户端的发送能力、服务端的接收能力是正常的。
>
> 第二次握手：服务端发包，客户端收到了。这样客户端就能得出结论：服务端的接收、发送能力，客户端的接收、发送能力是正常的。不过此时服务器并不能确认客户端的接收能力是否正常。
>
> 第三次握手：客户端发包，服务端收到了。这样服务端就能得出结论：客户端的接收、发送能力正常，服务器自己的发送、接收能力也正常。

因此，需要三次握手才能确认双方的接收与发送能力是否正常。

这样回答其实也是可以的，但我觉得，这个过程的我们应该要描述的更详细一点，因为三次握手的过程中，双方是由很多状态的改变的，而这些状态，也是面试官可能会问的点。所以我觉得在回答三次握手的时候，我们应该要描述的详细一点，而且描述的详细一点意味着可以扯久一点。加分的描述我觉得应该是这样：

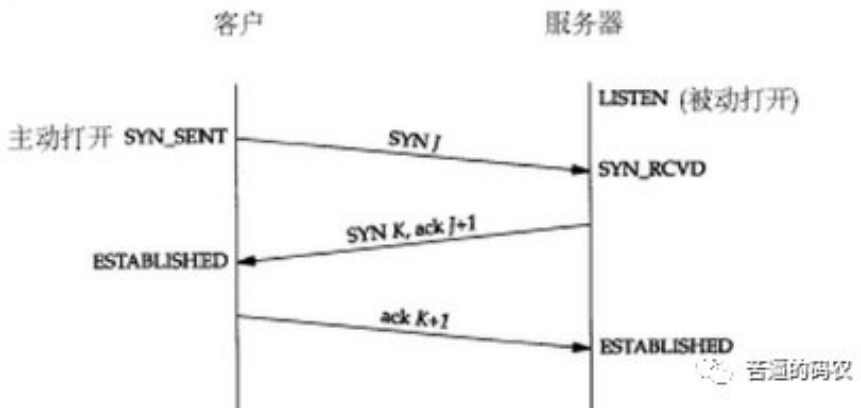
刚开始客户端处于 closed 的状态，服务端处于 listen 状态。然后

1、第一次握手：客户端给服务端发一个 SYN 报文，并指明客户端的初始化序列号 **ISN(c)**。此时客户端处于 **SYN_Send** 状态。

2、第二次握手：服务器收到客户端的 SYN 报文之后，会以自己的 SYN 报文作为应答，并且也是指定了自己的初始化序列号 ISN(s)，同时会把客户端的 ISN + 1 作为 ACK 的值，表示自己已经收到了客户端的 SYN，此时服务器处于 **SYN_REVD** 的状态。

3、第三次握手：客户端收到 SYN 报文之后，会发送一个 ACK 报文，当然，也是一样把服务器的 ISN + 1 作为 ACK 的值，表示已经收到了服务端的 SYN 报文，此时客户端处于 **establised** 状态。

4、服务器收到 ACK 报文之后，也处于 **establised 状态**，此时，双方以建立起了链接



三次握手的作用

三次握手的作用也是有好多的，多记住几个，保证不亏。例如：

1、确认双方的接受能力、发送能力是否正常。

2、指定自己的初始化序列号，为后面的可靠传送做准备。

1、（ISN）是固定的吗

三次握手的一个重要功能是客户端和服务端交换ISN(Initial Sequence Number), 以便让对方知道接下来接收数据的时候如何按序列号组装数据。

如果ISN是固定的，攻击者很容易猜出后续的确认号，因此 ISN 是动态生成的。

2、什么是半连接队列

服务器第一次收到客户端的 SYN 之后，就会处于 SYN_RCVD 状态，此时双方还没有完全建立其连接，服务器会把此种状态下请求连接放在一个队列里，我们把这种队列称之为**半连接队列**。当然还有一个**全连接队列**，就是已经完成三次握手，建立起连接的就会放在全连接队列中。如果队列满了就有可能会出现丢包现象。

> 这里在补充一点关于**SYN-ACK 重传次数**的问题： 服务器发送完SYN－ACK包，如果未收到客户确认包，服务器进行首次重传，等待一段时间仍未收到客户确认包，进行第二次重传，如果重传次数超 过系统规定的最大重传次数，系统将该连接信息从半连接队列中删除。注意，每次重传等待的时间不一定相同，一般会是指数增长，例如间隔时间为 1s, 2s, 4s, 8s,

3、三次握手过程中可以携带数据吗

很多人可能会认为三次握手都不能携带数据，其实第三次握手的时候，是可以携带数据的。也就是说，第一次、第二次握手不可以携带数据，而第三次握手是可以携带数据的。

为什么这样呢？大家可以想一个问题，假如第一次握手可以携带数据的话，如果有人要恶意攻击服务器，那他每次都在第一次握手中的 SYN 报文中放入大量的数据，因为攻击者根本就不理服务器的接收、发送能力是否正常，然后疯狂着重复发 SYN 报文的话，这会让服务器花费很多时间、内存空间来接收这些报文。也就是说，第一次握手可以放数据的话，其中一个简单的原因就是会让服务器更加容易受到攻击了。

而对于第三次的话，此时客户端已经处于 established 状态，也就是说，对于客户端来说，他已经建立起连接了，并且也已经知道服务器的接收、发送能力是正常的了，所以能携带数据页没啥毛病。

2、说一说四次挥手

四次挥手也一样，千万不要对方一个 FIN 报文，我方一个 ACK 报文，再我方一个 FIN 报文，我方一个 ACK 报文。然后结束，最好是说的详细一点，例如想下面这样就差不多了，要把每个阶段的**状态**记好，我上次面试就被问了几个了，呵呵。我答错了，还以为自己答对了，当时还解释的头头是道，呵呵。

刚开始双方都处于 establised 状态，假如是客户端先发起关闭请求，则：

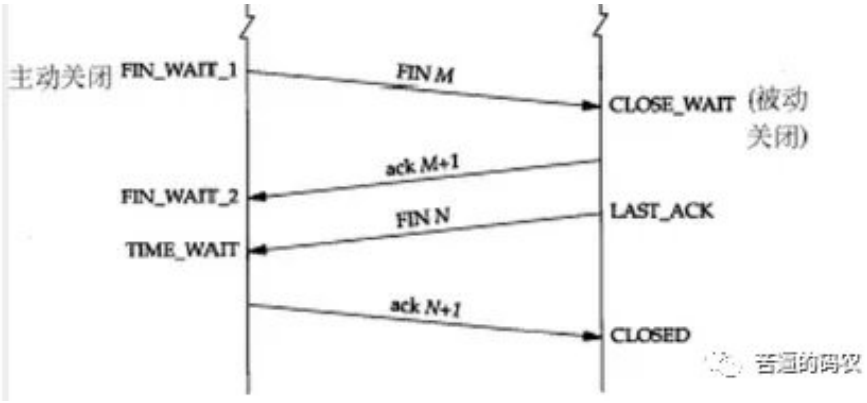
1、第一次挥手：客户端发送一个 FIN 报文，报文中会指定一个序列号。此时客户端处于**CLOSED_WAIT1**状态。

2、第二次握手：服务端收到 FIN 之后，会发送 ACK 报文，且把客户端的序列号值 + 1 作为 ACK 报文的序列号值，表明已经收到客户端的报文了，此时服务端处于 **CLOSE_WAIT2**状态。

3、第三次挥手：如果服务端也想断开连接了，和客户端的第一次挥手一样，发给 FIN 报文，且指定一个序列号。此时服务端处于 **LAST_ACK** 的状态。

4、第四次挥手：客户端收到 FIN 之后，一样发送一个 ACK 报文作为应答，且把服务端的序列号值 + 1 作为自己 ACK 报文的序列号值，此时客户端处于 **TIME_WAIT** 状态。需要过一阵子以确保服务端收到自己的 ACK 报文之后才会进入 CLOSED 状态

5、服务端收到 ACK 报文之后，就处于关闭连接了，处于 CLOSED 状态。



这里特别需要主要的就是**TIME_WAIT**这个状态了，这个是面试的高频考点，就是要理解，为什么客户端发送 ACK 之后不直接关闭，而是要等一阵子才关闭。这其中的原因就是，要确保服务器是否已经收到了我们的 ACK 报文，如果没有收到的话，服务器会重新发 FIN 报文给客户端，客户端再次收到 FIN 报文之后，就知道之前的 ACK 报文丢失了，然后再次发送 ACK 报文。

至于 TIME_WAIT 持续的时间至少是一个报文的来回时间。一般会设置一个计时，如果过了这个计时没有再次收到 FIN 报文，则代表对方成功就是 ACK 报文，此时处于 CLOSED 状态。

这里我给出每个状态所包含的含义，有兴趣的可以看看。

LISTEN - 侦听来自远方TCP端口的连接请求；

SYN-SENT -在发送连接请求后等待匹配的连接请求；

SYN-RECEIVED - 在收到和发送一个连接请求后等待对连接请求的确认；

ESTABLISHED- 代表一个打开的连接，数据可以传送给用户；

FIN-WAIT-1 - 等待远程TCP的连接中断请求，或先前的连接中断请求的确认；

FIN-WAIT-2 - 从远程TCP等待连接中断请求；

CLOSE-WAIT - 等待从本地用户发来的连接中断请求；

CLOSING -等待远程TCP对连接中断的确认；

LAST-ACK - 等待原来发向远程TCP的连接中断请求的确认；

TIME-WAIT -等待足够的时间以确保远程TCP接收到连接中断请求的确认；

CLOSED - 没有任何连接状态；

3、说一说POST与GET有哪些区别

使用场景

GET 用于获取资源，而 POST 用于传输实体主体。

参数

GET 和 POST 的请求都能使用额外的参数，但是 GET 的参数是以查询字符串出现在 URL 中，而 POST 的参数存储在实体主体中。不能因为 POST 参数存储在实体主体中就认为它的安全性更高，因为照样可以通过一些抓包工具（Fiddler）查看。

因为 URL 只支持 ASCII 码，因此 GET 的参数中如果存在中文等字符就需要先进行编码。例如 `中文` 会转换为 `%E4%B8%AD%E6%96%87`，而空格会转换为 `%20`。POST 参数支持标准字符集。

```
GET /test/demo_form.asp?name1=value1&name2=value2 HTTP/1.1Copy to clipboardErrorCopied
POST /test/demo_form.asp HTTP/1.1
Host: w3schools.com
name1=value1&name2=value2Copy to clipboardErrorCopied
```

安全性

安全的 HTTP 方法不会改变服务器状态，也就是说它只是可读的。

GET 方法是安全的，而 POST 却不是，因为 POST 的目的是传送实体主体内容，这个内容可能是用户上传的表单数据，上传成功之后，服务器可能把这个数据存储到数据库中，因此状态也就发生了改变。

安全的方法除了 GET 之外还有：HEAD、OPTIONS。

不安全的方法除了 POST 之外还有 PUT、DELETE。

幂等性

幂等的 HTTP 方法，同样的请求被执行一次与连续执行多次的效果是一样的，服务器的状态也是一样的。换句话说就是，幂等方法不应该具有副作用（统计用途除外）。

所有的安全方法也都是幂等的。

在正确实现的条件下，GET，HEAD，PUT 和 DELETE 等方法都是幂等的，而 POST 方法不是。

GET /pageX HTTP/1.1 是幂等的，连续调用多次，客户端接收到的结果都是一样的：

```
GET /pageX HTTP/1.1
GET /pageX HTTP/1.1
GET /pageX HTTP/1.1
GET /pageX HTTP/1.1Copy to clipboardErrorCopied
```

POST /add_row HTTP/1.1 不是幂等的，如果调用多次，就会增加多行记录：

```
POST /add_row HTTP/1.1   -> Adds a 1nd row
POST /add_row HTTP/1.1   -> Adds a 2nd row
POST /add_row HTTP/1.1   -> Adds a 3rd rowCopy to clipboardErrorCopied
```

DELETE /idX/delete HTTP/1.1 是幂等的，即使不同的请求接收到的状态码不一样：

```
DELETE /idX/delete HTTP/1.1   -> Returns 200 if idX exists
DELETE /idX/delete HTTP/1.1   -> Returns 404 as it just got deleted
DELETE /idX/delete HTTP/1.1   -> Returns 404Copy to clipboardErrorCopied
```

可缓存

如果要对响应进行缓存，需要满足以下条件：

- 请求报文的 HTTP 方法本身是可缓存的，包括 GET 和 HEAD，但是 PUT 和 DELETE 不可缓存，POST 在多数情况下不可缓存的。
- 响应报文的状态码是可缓存的，包括：200, 203, 204, 206, 300, 301, 404, 405, 410, 414, and 501。
- 响应报文的 Cache-Control 首部字段没有指定不进行缓存。

XMLHttpRequest

为了阐述 POST 和 GET 的另一个区别，需要先了解 XMLHttpRequest：

> XMLHttpRequest 是一个 API，它为客户端提供了在客户端和服务器之间传输数据的功能。它提供了一个通过 URL 来获取数据的简单方式，并且不会使整个页面刷新。这使得网页只更新一部分页面而不会打扰到用户。XMLHttpRequest 在 AJAX 中被大量使用。

- 在使用 XMLHttpRequest 的 POST 方法时，浏览器会先发送 Header 再发送 Data。但并不是所有浏览器会这么做，例如火狐就不会。

- 而 GET 方法 Header 和 Data 会一起发送。

4、面试官：说一说TCP与UDP的区别

TCP协议的主要特点

（1）TCP是面向连接的运输层协议；所谓面向连接就是双方传输数据之前，必须先建立一条通道，例如三次握手就是建议通道的一个过程，而四次挥手则是结束销毁通道的一个其中过程。

（2）每一条TCP连接只能有两个端点（即两个套接字），只能是点对点的；

（3）TCP提供可靠的传输服务。传送的数据无差错、不丢失、不重复、按序到达；

（4）TCP提供全双工通信。允许通信双方的应用进程在任何时候都可以发送数据，因为两端都设有发送缓存和接受缓存；

（5）面向字节流。虽然应用程序与TCP交互是一次一个大小不等的数据块，但TCP把这些数据看成一连串无结构的字节流，它不保证接收方收到的数据块和发送方发送的数据块具有对应大小关系，例如，发送方应用程序交给发送方的TCP10个数据块，但就受访的TCP可能只用了4个数据块久保收到的字节流交付给上层的应用程序，但字节流完全一样。

TCP的可靠性原理

可靠传输有如下两个特点:

a.传输信道无差错,保证传输数据正确;

b.不管发送方以多快的速度发送数据,接收方总是来得及处理收到的数据;

（1）首先，采用三次握手来建立TCP连接，四次握手来释放TCP连接，从而保证建立的传输信道是可靠的。

（2）其次，TCP采用了连续ARQ协议（回退N，Go-back-N；超时自动重传）来保证数据传输的正确性，使用滑动窗口协议来保证接方能够及时处理所接收到的数据，进行流量控制。

（3）最后，TCP使用慢开始、拥塞避免、快重传和快恢复来进行拥塞控制，避免网络拥塞。

UDP协议特点

（1）UDP是无连接的传输层协议；

（2）UDP使用尽最大努力交付，不保证可靠交付；

（3）UDP是面向报文的，对应用层交下来的报文，不合并，不拆分，保留原报文的边界；

（4）UDP没有拥塞控制，因此即使网络出现拥塞也不会降低发送速率；

（5）UDP支持一对一　一对多　多对多的交互通信；

（6）UDP的首部开销小，只有 8 字节.

TCP和UDP的区别

(1)TCP是可靠传输,UDP是不可靠传输;

(2)TCP面向连接,UDP无连接;

(3)TCP传输数据有序,UDP不保证数据的有序性;

(4)TCP不保存数据边界,UDP保留数据边界;

(5)TCP传输速度相对UDP较慢;

(6)TCP有流量控制和拥塞控制,UDP没有;

(７)TCP是重量级协议,UDP是轻量级协议;

(８)TCP首部较长２０字节,UDP首部较短８字节;

基于TCP和UDP的常用协议

HTTP、HTTPS、FTP、TELNET、SMTP(简单邮件传输协议)协议基于可靠的TCP协议。TFTP、DNS、DHCP、TFTP、SNMP(简单网络管理协议)、RIP基于不可靠的UDP协议

TCP 和 UDP 的常用场景，这个问的好挺多的，例如我当时面试时，就被问过：QQ 登录的过程中，用到了 TCP 和 UDP，QQ 通话呢?

5、面试题：说一说HTTP1.0，1.1，2.0 的区别

http1.0：不能长连接，只能短连接

http1.1：可以长连接，也就是说，一个TCP连接可以发送多个HTTP请求，不过服务器只能按照顺序一个一个响应。不过规定了用Pipelining来试图解决这个问题，不过该功能默认是关闭的。Pipelining的意思就是说，可以在一个连接中发送多个请求（不需要等待响应），不过服务器端必须要按照收到的顺序

```

connetion:keep-alive | closed;

```

默认情况下浏览器不开启该功能。

http2.0：采用了多路复用，它把HTTP报文分解成更小的二进制帧来传送，不同的HTTP请求报文可以混合在一个TCP连接上传输，服务器收到后，在根据二进制帧里面存放的ID来进行分类，拼接。采用这种方式，相当于服务器可以同时处理几个不同的HTTP请求，也就是说，不需要吧整个HTTP请求处理完，就可以去处理其他的HTTP请求了。

并且HTTP还对请求头部进行了压缩。

6、什么是SQL 注入？举个例子?

SQL注入就是通过把SQL命令插入到Web表单提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。

1). SQL注入攻击的总体思路

(1). 寻找到SQL注入的位置　(2). 判断服务器类型和后台数据库类型　(3). 针对不通的服务器和数据库特点进行SQL注入攻击

2). SQL注入攻击实例

比如，在一个登录界面，要求输入用户名和密码，可以这样输入实现免帐号登录：

```
用户名:  'or 1 = 1 --
密 码:
```

用户一旦点击登录，如若没有做特殊处理，那么这个非法用户就很得意的登陆进去了。这是为什么呢?

下面我们分析一下：从理论上说，后台认证程序中会有如下的SQL语句：

```
String sql = “select * from user_table where username=’ “+userName+” ’ and password=’ “+password+” ‘”;
```

因此，当输入了上面的用户名和密码，上面的SQL语句变成：

```
SELECT * FROM user_table WHERE username=’’or 1 = 1 –– and password=’’
```

分析上述SQL语句我们知道，username=‘ or 1=1 这个语句一定会成功；然后后面加两个-，这意味着注释，它将后面的语句注释，让他们不起作用。这样，上述语句永远都能正确执行，用户轻易骗过系统，获取合法身份。

3). 应对方法

(1). 参数绑定

使用预编译手段，绑定参数是最好的防SQL注入的方法。目前许多的ORM框架及JDBC等都实现了SQL预编译和参数绑定功能，攻击者的恶意SQL会被当做SQL的参数而不是SQL命令被执行。在mybatis的mapper文件中，对于传递的参数我们一般是使用#和Error: You can't use 'macro parameter character #' in math mode时，变量就是直接追加在sql中，一般会有sql注入问题。

(2). 使用正则表达式过滤传入的参数

7、谈一谈 XSS 攻击，举个例子?

XSS是一种经常出现在web应用中的计算机安全漏洞，与SQL注入一起成为web中最主流的攻击方式。

XSS是指恶意攻击者利用网站没有对用户提交数据进行转义处理或者过滤不足的缺点，进而添加一些脚本代码嵌入到web页面中去，使别的用户访问都会执行相应的嵌入代码，从而盗取用户资料、利用用户身份进行某种动作或者对访问者进行病毒侵害的一种攻击方式。

1). XSS攻击的危害

盗取各类用户帐号，如机器登录帐号、用户网银帐号、各类管理员帐号

控制企业数据，包括读取、篡改、添加、删除企业敏感数据的能力

盗窃企业重要的具有商业价值的资料

非法转账

强制发送电子邮件

网站挂马

控制受害者机器向其它网站发起攻击

2). 原因解析

主要原因：过于信任客户端提交的数据!

解决办法：不信任任何客户端提交的数据，只要是客户端提交的数据就应该先进行相应的过滤处理然后方可进行下一步的操作。

进一步分析细节：客户端提交的数据本来就是应用所需要的，但是恶意攻击者利用网站对客户端提交数据的信任，在数据中插入一些符号以及javascript代码，那么这些数据将会成为应用代码中的一部分了，那么攻击者就可以肆无忌惮地展开攻击啦，因此我们绝不可以信任任何客户端提交的数据！！！

3). XSS 攻击分类

(1). 反射性XSS攻击 (非持久性XSS攻击)

漏洞产生的原因是攻击者注入的数据反映在响应中。一个典型的非持久性XSS攻击包含一个带XSS攻击向量的链接(即每次攻击需要用户的点击)，例如，正常发送消息：

```
http://www.test.com/message.php?send=Hello,World!
```

接收者将会接收信息并显示Hello,World；但是，非正常发送消息：

```
http://www.test.com/message.php?send=<script>alert(‘foolish!’)</script>!


接收者接收消息显示的时候将会弹出警告窗口!

(2). 持久性xss攻击 (留言板场景)

xss攻击向量(一般指xss攻击代码)存储在网站数据库，当一个页面被用户打开的时候执行。也就是说，每当用户使用浏览器打开指定页面时，脚本便执行。

与非持久性xss攻击相比，持久性xss攻击危害性更大。从名字就可以了解到，持久性xss攻击就是将攻击代码存入数据库中，然后客户端打开时就执行这些攻击代码。

例如，留言板表单中的表单域：

•```html
<input type="text" name="content" value="这里是用户填写的数据">
```

正常操作流程是：用户是提交相应留言信息 —— 将数据存储到数据库 —— 其他用户访问留言板，应用去数据并显示；而非正常操作流程是攻击者在value填写：

```
<script>alert(‘foolish!’);</script> <!--或者html其他标签（破坏样式。。。）、一段攻击型代码-->
```

并将数据提交、存储到数据库中；当其他用户取出数据显示的时候，将会执行这些攻击性代码。

4). 修复漏洞方针

漏洞产生的根本原因是 太相信用户提交的数据，对用户所提交的数据过滤不足所导致的，因此解决方案也应该从这个方面入手，具体方案包括：

将重要的cookie标记为http only, 这样的话Javascript 中的document.cookie语句就不能获取到cookie了（如果在cookie中设置了HttpOnly属性，那么通过js脚本将无法读取到cookie信息，这样能有效的防止XSS攻击）；

表单数据规定值的类型，例如：年龄应为只能为int、name只能为字母数字组合。。。。

对数据进行Html Encode 处理

过滤或移除特殊的Html标签，例如: ,

, < for <, > for>, &quot for

过滤JavaScript 事件的标签，例如 “onclick=”, “onfocus” 等等。

需要注意的是，在有些应用中是允许html标签出现的，甚至是javascript代码出现。因此，我们在过滤数据的时候需要仔细分析哪些数据是有特殊要求（例如输出需要html代码、javascript代码拼接、或者此表单直接允许使用等等），然后区别处理！

8、在交互过程中如果数据传送完了，还不想断开连接怎么办，怎么维持?

在 HTTP 中响应体的 Connection 字段指定为 keep-alive

```
connetion:keep-alive;
```

9、GET请求中URL编码的意义

我们知道，在GET请求中会对URL中非西文字符进行编码，这样做的目的就是为了 **避免歧义**。看下面的例子，

针对“name1=value1&name2=value2”的例子，我们来谈一下数据从客户端到服务端的解析过程。首先，上述字符串在计算机中用ASCII吗表示为：

```
    6E616D6531 3D 76616C756531 26 6E616D6532 3D 76616C756532
    6E616D6531：name1
    3D：=
    76616C756531：value1
    26：&
    6E616D6532：name2
    3D：=
    76616C756532：value2
```

服务端在接收到该数据后就可以遍历该字节流，一个字节一个字节的吃，当吃到3D这字节后，服务端就知道前面吃得字节表示一个key，再往后吃，如果遇到26，说明从刚才吃的3D到26子节之间的是上一个key的value，以此类推就可以解析出客户端传过来的参数。

现在考虑这样一个问题，如果我们的参数值中就包含 = 或 & 这种特殊字符的时候该怎么办？比如，“name1=value1”，其中value1的值是“va&lu=e1”字符串，那么实际在传输过程中就会变成这样“name1=va&lu=e1”。这样，我们的本意是只有一个键值对，但是服务端却会解析成两个键值对，这样就产生了歧义。

那么，如何解决上述问题带来的歧义呢？解决的办法就是对参数进行URL编码：例如，我们对上述会产生歧义的字符进行URL编码后结果：“name1=va%26lu%3D”，这样服务端会把紧跟在“%”后的字节当成普通的字节，就是不会把它当成各个参数或键值对的分隔符

10、HTTP 哪些常用的状态码及使用场景?

状态码分类

1xx：表示目前是协议的中间状态，还需要后续请求

2xx：表示请求成功

3xx：表示重定向状态，需要重新请求

4xx：表示请求报文错误

5xx：服务器端错误

常用状态码

101 切换请求协议，从 HTTP 切换到 WebSocket

200 请求成功，有响应体

301 永久重定向：会缓存

302 临时重定向：不会缓存

304 协商缓存命中

403 服务器禁止访问

404 资源未找到

400 请求错误

500 服务器端错误

503 服务器繁忙

11、HTTP 如何实现长连接？在什么时候会超时?

通过在头部（请求和响应头）设置 Connection: keep-alive，HTTP1.0协议支持，但是默认关闭，从HTTP1.1协议以后，连接默认都是长连接

1、HTTP 一般会有 httpd 守护进程，里面可以设置 keep-alive timeout，当 tcp 链接闲置超过这个时间就会关闭，也可以在 HTTP 的 header 里面设置超时时间

2、TCP 的 keep-alive 包含三个参数，支持在系统内核的 net.ipv4 里面设置：当 TCP 链接之后，闲置了 tcp_keepalive_time，则会发生侦测包，如果没有收到对方的 ACK，那么会每隔 tcp_keepalive_intvl 再发一次，直到发送了 tcp_keepalive_probes，就会丢弃该链接。

（1）tcp_keepalive_intvl = 15 （2）tcp_keepalive_probes = 5 （3）tcp_keepalive_time = 1800

实际上 HTTP 没有长短链接，只有 TCP 有，TCP 长连接可以复用一个 TCP 链接来发起多次 HTTP 请求，这样可以减少资源消耗，比如一次请求 HTML，可能还需要请求后续的 JS/CSS/图片等

12、HTTP状态码301和302的区别，都有哪些用途?

一. 301重定向的概念

301重定向（301 Move Permanently），指页面永久性转移，表示为资源或页面永久性地转移到了另一个位置。301是HTTP协议中的一种状态码，当用户或搜索引擎向服务器发出浏览请求时，服务器返回的HTTP数据流中头信息（header）中包含状态码 301 ，表示该资源已经永久改变了位置。

301重定向是一种非常重要的"自动转向"技术，网址重定向最为可行的一种方法。

二. 哪些情况需要做301重定向?

网页开发过程中，时常会遇到网站目录结构的调整，将页面转移到一个新地址；网页扩展名的改变，这些变化都会导致网页地址发生改变，此时用户收藏夹和搜索引擎数据库中的旧地址是一个错误的地址，访问之后会出现404页面，直接导致网站流量的损失。或者是我们需要多个域名跳转至同一个域名，例如本站主站点域名为 www.conimi.com ，而还有一个域名 www.nico.cc，由于对该域名设置了301重定向，当输入www.nico.cc 时，自动跳转至 www.conimi.com 。

三. 301重定向有什么优点?

有利于网站首选域的确定，对于同一资源页面多条路径的301重定向有助于URL权重的集中。例如 www.conimi.com和 conimi.com 是两个不同的域名，但是指向的内容完全相同，搜索引擎会对两个域名收录情况不同，这样导致网站权重和排名被分散；对conimi.com 做301重定向跳转至www.conimi.com 后，权重和排名集中到www.conimi.com，从而提升自然排名。

四. 302重定向又是什么鬼?

302重定向（302 Move Temporarily），指页面暂时性转移，表示资源或页面暂时转移到另一个位置，常被用作网址劫持，容易导致网站降权，严重时网站会被封掉，不推荐使用。

五. 301与302的区别

301重定向是页面永久性转移，搜索引擎在抓取新内容的同时也将旧的网址替换成重定向之后的网址；

302重定向是页面暂时性转移，搜索引擎会抓取新的内容而保存旧的网址并认为新的网址只是暂时的。

13、IP地址有哪些分类?

A类地址(1~126)：网络号占前8位，以0开头，主机号占后24位。

B类地址(128~191)：网络号占前16位，以10开头，主机号占后16位。

C类地址(192~223)：网络号占前24位，以110开头，主机号占后8位。

D类地址(224~239)：以1110开头，保留位多播地址。

E类地址(240~255)：以1111开头，保留位今后使用

这种两级的 IP 地址可以记为：

$$\text{IP 地址} ::= \{ <\text{网络号}>, <\text{主机号}> \} \tag{4-1}$$

式(4-1)中的符号“::=”表示“**定义为**”。图 4-5 给出了各种 IP 地址的网络号字段和主机号字段，这里 A 类、B 类和 C 类地址都是**单播地址**（一对一通信），是最常用的。

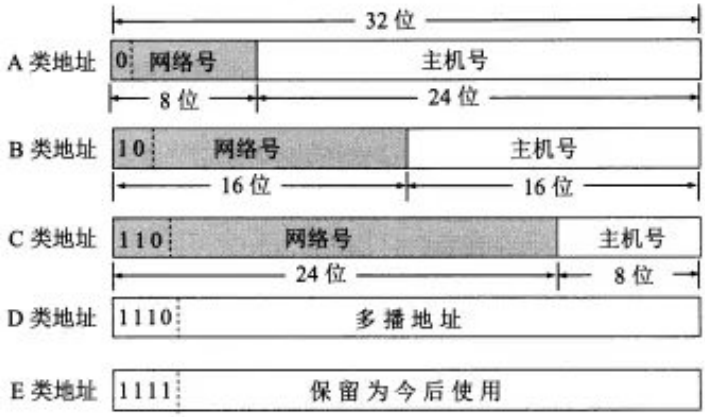


图 4-5　IP 地址中的网络号字段和主机号字段

从图 4-5 可以看出：

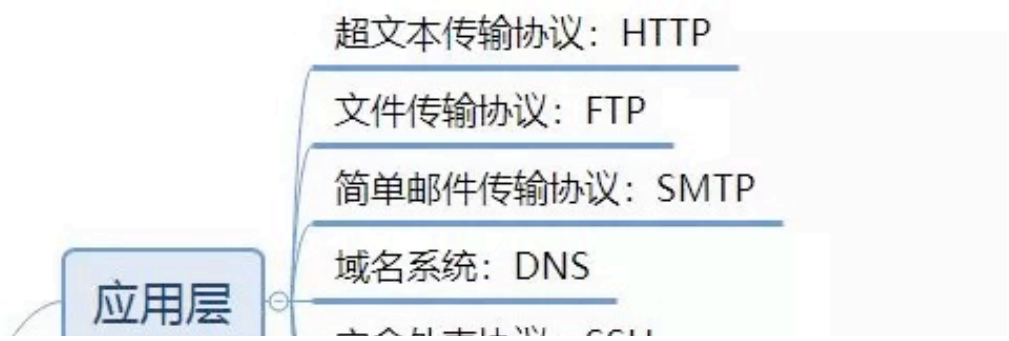
- A 类、B 类和 C 类地址的网络号字段（在图中这个字段是灰色的）分别为 1 个、2 个和 3 个字节长，而在网络号字段的最前面有 1~3 位的**类别位**，其数值分别规定为 0，10 和 110。
- A 类、B 类和 C 类地址的主机号字段分别为 3 个、2 个和 1 个字节长。
- D 类地址（前 4 位是 1110）用于**多播**（一对多通信）。我们将在 4.6 节讨论 IP 多播。
- E 类地址（前 4 位是 1111）保留为以后用。

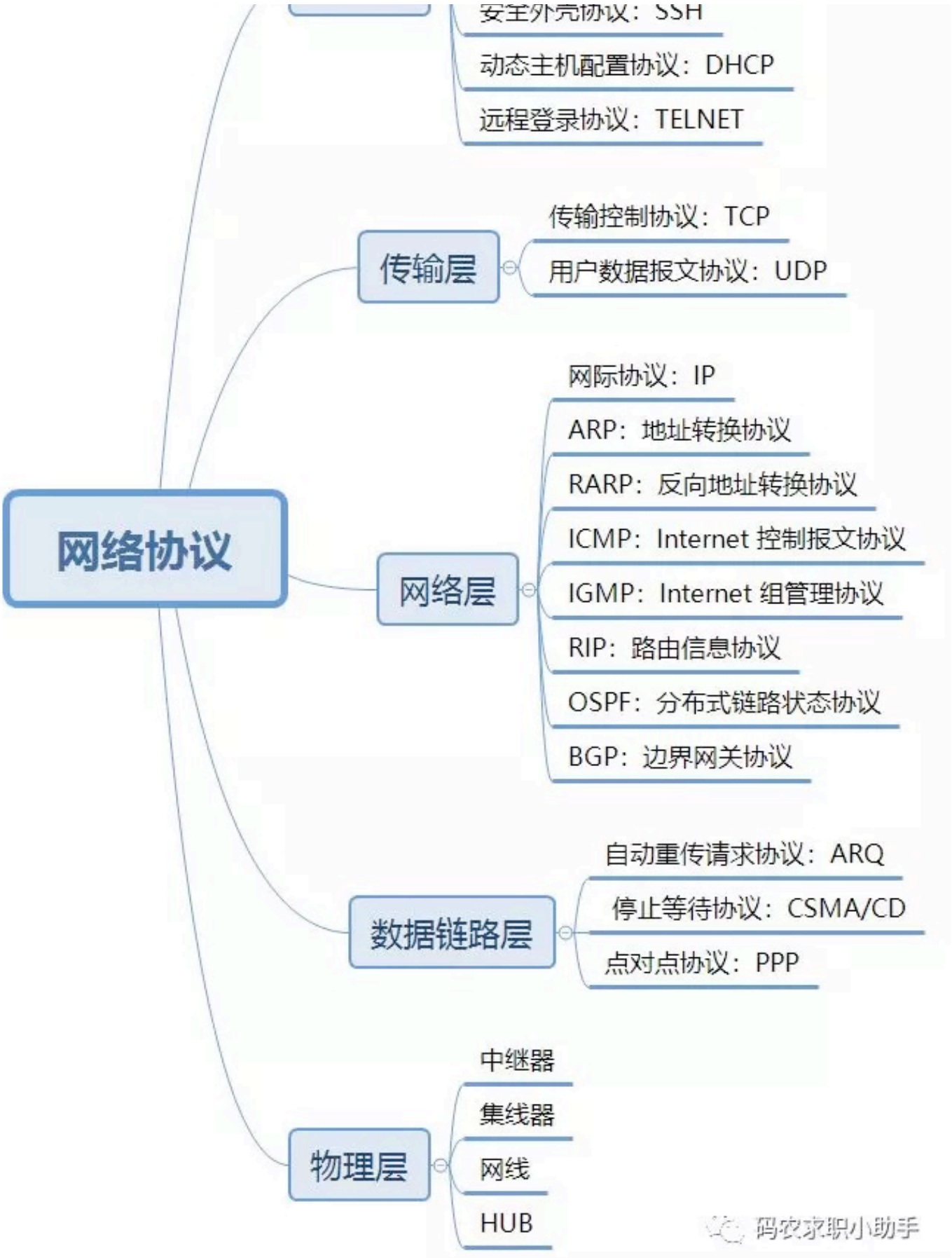
表 4-2　IP 地址的指派范围

网络类别	最大可指派的网络数	第一个可指派的网络号	最后一个可指派的网络号	每个网络中的最大主机数
A	126 (2^7-2)	1	126	16777214
B	16383 ($2^{14}-1$)	128.1	191.255	65534
C	2097151 ($2^{21}-1$)	192.0.1	223.255.255	254

14、简单说下每一层对应的网络协议有哪些?

计算机五层网络体系中涉及的协议非常多，下面就常用的做了列举：




网络协议
安全外壳协议：SSH
动态主机配置协议：DHCP
远程登录协议：TELNET
传输层
传输控制协议：TCP
用户数据报文协议：UDP
网络层
网际协议：IP
ARP：地址转换协议
RARP：反向地址转换协议
ICMP：Internet 控制报文协议
IGMP：Internet 组管理协议
RIP：路由信息协议
OSPF：分布式链路状态协议
BGP：边界网关协议
数据链路层
自动重传请求协议：ARQ
停止等待协议：CSMA/CD
点对点协议：PPP
物理层
中继器
集线器
网线
HUB

15、ARP 协议的工作原理?

网络层的 ARP 协议完成了 IP 地址与物理地址的映射。首先，每台主机都会在自己的 ARP 缓冲区中建立一个 ARP 列表，以表示 IP 地址和 MAC 地址的对应关系。当源主机需要将一个数据包要发送到目的主机时，会首先检查自己 ARP 列表中是否存在该 IP 地址对应的 MAC 地址：如果有，就直接将数据包发送到这个 MAC 地址；如果没有，就向本地网段发起一个 ARP 请求的广播包，查询此目的主机对应的 MAC 地址。

此 ARP 请求数据包里包括源主机的 IP 地址、硬件地址、以及目的主机的 IP 地址。网络中所有的主机收到这个 ARP 请求后，会检查数据包中的目的 IP 是否和自己的 IP 地址一致。如果不相同就忽略此数据包；如果相同，该主机首先将发送端的 MAC 地址和 IP 地址添加到自己的 ARP 列表中，如果 ARP 表中已经存在该 IP 的信息，则将其覆盖，然后给源主机发送一个 ARP 响应数据包，告诉对方自己是它需要查找的 MAC 地址；源主机收到这个 ARP 响应数据包后，将得到的目的主机的 IP 地址和 MAC 地址添加到自己的 ARP 列表中，并利用此信息开始数据的传输。如果源主机一直没有收到 ARP 响应数据包，表示 ARP 查询失败

16、TCP 的主要特点是什么?

1. TCP 是面向连接的。（就好像打电话一样，通话前需要先拨号建立连接，通话结束后要挂机释放连接）；
2. 每一条 TCP 连接只能有两个端点，每一条 TCP 连接只能是点对点的（一对一）；
3. TCP 提供可靠交付的服务。通过 TCP 连接传送的数据，无差错、不丢失、不重复、并且按序到达；
4. TCP 提供全双工通信。TCP 允许通信双方的应用进程在任何时候都能发送数据。TCP 连接的两端都设有发送缓存和接收缓存，用来临时存放双方通信的数据；
5. 面向字节流。TCP 中的“流”（Stream）指的是流入进程或从进程流出的字节序列。“面向字节流”的含义是：虽然应用程序和 TCP 的交互是一次一个数据块（大小不等），但 TCP 把应用程序交下来的数据仅仅看成是一连串的无结构的字节流。

17、UDP 的主要特点是什么?

1. UDP 是无连接的；
2. UDP 使用尽最大努力交付，即不保证可靠交付，因此主机不需要维持复杂的链接状态（这里面有许多参数）；
3. UDP 是面向报文的；
4. UDP 没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如 直播，实时视频会议等）；
5. UDP 支持一对一、一对多、多对一和多对多的交互通信；
6. UDP 的首部开销小，只有 8 个字节，比 TCP 的 20 个字节的首部要短。

18、TCP 和 UDP 分别对应的常见应用层协议有哪些?

1. TCP 对应的应用层协议

FTP：定义了文件传输协议，使用 21 端口。常说某某计算机开了 FTP 服务便是启动了文件传输服务。下载文件，上传主页，都要用到 FTP 服务。

Telnet：它是一种用于远程登陆的端口，用户可以以自己的身份远程连接到计算机上，通过这种端口可以提供一种基于 DOS 模式下的通信服务。如以前的 BBS 是-纯字符界面的，支持 BBS 的服务器将 23 端口打开，对外提供服务。

SMTP：定义了简单邮件传送协议，现在很多邮件服务器都用的是这个协议，用于发送邮件。如常见的免费邮件服务中用的就是这个邮件服务端口，所以在电子邮件设置-中常看到有这么 SMTP 端口设置这个栏，服务器开放的是 25 号端口。

POP3：它是和 SMTP 对应，POP3 用于接收邮件。通常情况下，POP3 协议所用的是 110 端口。也是说，只要你有相应的使用 POP3 协议的程序（例如 Fo-xmail 或 Outlook），就可以不以 Web 方式登陆进邮箱界面，直接用邮件程序就可以收到邮件（如是163 邮箱就没有必要先进入网易网站，再进入自己的邮-箱来收信）。

HTTP：从 Web 服务器传输超文本到本地浏览器的传送协议。

2. UDP 对应的应用层协议

DNS：用于域名解析服务，将域名地址转换为 IP 地址。DNS 用的是 53 号端口。

SNMP：简单网络管理协议，使用 161 号端口，是用来管理网络设备的。由于网络设备很多，无连接的服务就体现出其优势。

TFTP(Trival File Transfer Protocal)：简单文件传输协议，该协议在熟知端口 69 上使用 UDP 服务。

19、为什么 TIME-WAIT 状态必须等待 2MSL 的时间呢?

1、为了保证 A 发送的最后一个 ACK 报文段能够到达 B。这个 ACK 报文段有可能丢失，因而使处在 LAST-ACK 状态的 B 收不到对已发送的 FIN + ACK 报文段的确认。B 会超时重传这个 FIN+ACK 报文段，而 A 就能在 2MSL 时间内（超时 + 1MSL 传输）收到这个重传的 FIN+ACK 报文段。接着 A 重传一次确认，重新启动 2MSL 计时器。最后，A 和 B 都正常进入到 CLOSED 状态。如果 A 在 TIME-WAIT 状态不等待一段时间，而是在发送完 ACK 报文段后立即释放连接，那么就无法收到 B 重传的 FIN + ACK 报文段，因而也不会再发送一次确认报文段，这样，B 就无法按照正常步骤进入 CLOSED 状态。

2、 防止已失效的连接请求报文段出现在本连接中。A 在发送完最后一个 ACK 报文段后，再经过时间 2MSL，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失。这样就可以使下一个连接中不会出现这种旧的连接请求报文段。

20、保活计时器的作用?

除时间等待计时器外，TCP 还有一个保活计时器（keepalive timer）。设想这样的场景：客户已主动与服务器建立了 TCP 连接。但后来客户端的主机突然发生故障。显然，服务器以后就不能再收到客户端发来的数据。因此，应当有措施使服务器不要再白白等待下去。这就需要使用保活计时器了。

服务器每收到一次客户的数据，就重新设置保活计时器，时间的设置通常是两个小时。若两个小时都没有收到客户端的数据，服务端就发送一个探测报文段，以后则每隔 75 秒钟发送一次。若连续发送 10个 探测报文段后仍然无客户端的响应，服务端就认为客户端出了故障，接着就关闭这个连接。

21、TCP 协议是如何保证可靠传输的?

1. 数据包校验：目的是检测数据在传输过程中的任何变化，若校验出包有错，则丢弃报文段并且不给出响应，这时 TCP 发送数据端超时后会重发数据；
2. 对失序数据包重排序：既然 TCP 报文段作为 IP 数据报来传输，而 IP 数据报的到达可能会失序，因此 TCP 报文段的到达也可能会失序。TCP 将对失序数据进行重新排序，然后才交给应用层；
3. 丢弃重复数据：对于重复数据，能够丢弃重复数据；
4. 应答机制：当 TCP 收到发自 TCP 连接另一端的数据，它将发送一个确认。这个确认不是立即发送，通常将推迟几分之一秒；
5. 超时重发：当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段；
6. 流量控制：TCP 连接的每一方都有固定大小的缓冲空间。TCP 的接收端只允许另一端发送接收端缓冲区所能接纳的数据，这可以防止较快主机致使较慢主机的缓冲区溢出，这就是流量控制。TCP 使用的流量控制协议

是可变大小的滑动窗口协议。

22、谈谈你对停止等待协议的理解?

停止等待协议是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组；在停止等待协议中，若接收方收到重复分组，就丢弃该分组，但同时还要发送确认。主要包括以下几种情况：无差错情况、出现差错情况（超时重传）、确认丢失和确认迟到、确认丢失和确认迟到。

23、谈谈你对 ARQ 协议的理解?

自动重传请求 ARQ 协议

停止等待协议中超时重传是指只要超过一段时间仍然没有收到确认，就重传前面发送过的分组（认为刚才发送过的分组丢失了）。因此每发送完一个分组需要设置一个超时计时器，其重传时间应比数据在分组传输的平均往返时间更长一些。这种自动重传方式常称为自动重传请求 ARQ。

连续 ARQ 协议

连续 ARQ 协议可提高信道利用率。发送方维持一个发送窗口，凡位于发送窗口内的分组可以连续发送出去，而不需要等待对方确认。接收方一般采用累计确认，对按序到达的最后一个分组发送确认，表明到这个分组为止的所有分组都已经正确收到了。

24、谈谈你对滑动窗口的了解?

TCP 利用滑动窗口实现流量控制的机制。滑动窗口（Sliding window）是一种流量控制技术。早期的网络通信中，通信双方不会考虑网络的拥挤情况直接发送数据。由于大家不知道网络拥塞状况，同时发送数据，导致中间节点阻塞掉包，谁也发不了数据，所以就有了滑动窗口机制来解决此问题。

TCP 中采用滑动窗口来进行传输控制，滑动窗口的大小意味着接收方还有多大的缓冲区可以用于接收数据。发送方可以通过滑动窗口的大小来确定应该发送多少字节的数据。当滑动窗口为 0 时，发送方一般不能再发送数据报，但有两种情况除外，一种情况是可以发送紧急数据，例如，允许用户终止在远端机上的运行进程。另一种情况是发送方可以发送一个 1 字节的数据报来通知接收方重新声明它希望接收的下一字节及发送方的滑动窗口大小。

25、谈下你对流量控制的理解?

TCP 利用滑动窗口实现流量控制。流量控制是为了控制发送方发送速率，保证接收方来得及接收。接收方发送的确认报文中的窗口字段可以用来控制发送方窗口大小，从而影响发送方的发送速率。将窗口字段设置为 0，则发送方不能发送数据。

26、谈下你对 TCP 拥塞控制的理解？使用了哪些算法?

拥塞控制和流量控制不同，前者是一个全局性的过程，而后者指点对点通信量的控制。在某段时间，若对网络中某一资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏。这种情况就叫拥塞。

拥塞控制就是为了防止过多的数据注入到网络中，这样就可以使网络中的路由器或链路不致于过载。拥塞控制所要做的都有一个前提，就是网络能够承受现有的网络负荷。拥塞控制是一个全局性的过程，涉及到所有的主机，所有的路由器，以及与降低网络传输性能有关的所有因素。相反，流量控制往往是点对点通信量的控制，是个端到端的问题。流量控制所要做到的就是抑制发送端发送数据的速率，以便使接收端来得及接收。

为了进行拥塞控制，TCP 发送方要维持一个拥塞窗口(cwnd) 的状态变量。拥塞控制窗口的大小取决于网络的拥塞程度，并且动态变化。发送方让自己的发送窗口取为拥塞窗口和接收方的接受窗口中较小的一个。

TCP 的拥塞控制采用了四种算法，即：慢开始、拥塞避免、快重传和快恢复。在网络层也可以使路由器采用适当的分组丢弃策略（如：主动队列管理 AQM），以减少网络拥塞的发生。

慢开始：

慢开始算法的思路是当主机开始发送数据时，如果立即把大量数据字节注入到网络，那么可能会引起网络阻塞，因为现在还不知道网络的符合情况。经验表明，较好的方法是先探测一下，即由小到大逐渐增大发送窗口，也就是由小到大逐渐增大拥塞窗口数值。cwnd 初始值为 1，每经过一个传播轮次，cwnd 加倍。

拥塞避免：

拥塞避免算法的思路是让拥塞窗口 cwnd 缓慢增大，即每经过一个往返时间 RTT 就把发送方的 cwnd 加 1。

快重传与快恢复：

在 TCP/IP 中，快速重传和快恢复（fast retransmit and recovery，FRR）是一种拥塞控制算法，它能快速恢复丢失的数据包。

没有 FRR，如果数据包丢失了，TCP 将会使用定时器来要求传输暂停。在暂停的这段时间内，没有新的或复制的数据包被发送。有了 FRR，如果接收机接收到一个不按顺序的数据段，它会立即给发送机发送一个重复确认。如果发送机接收到三个重复确认，它会假定确认件指出的数据段丢失了，并立即重传这些丢失的数据段。

有了 FRR，就不会因为重传时要求的暂停被耽误。当有单独的数据包丢失时，快速重传和快恢复（FRR）能最有效地工作。当有多个数据信息包在某一段很短的时间内丢失时，它则不能很有效地工作。

27、什么是粘包?

在进行 Java NIO 学习时，可能会发现：如果客户端连续不断的向服务端发送数据包时，服务端接收的数据会出现两个数据包粘在一起的情况。

1. TCP 是基于字节流的，虽然应用层和 TCP 传输层之间的数据交互是大小不等的数据块，但是 TCP 把这些数据块仅仅看成一连串无结构的字节流，没有边界；
2. 从 TCP 的帧结构也可以看出，在 TCP 的首部没有表示数据长度的字段。

基于上面两点，在使用 TCP 传输数据时，才有粘包或者拆包现象发生的可能。一个数据包中包含了发送端发送的两个数据包的信息，这种现象即为粘包。

接收端收到了两个数据包，但是这两个数据包要么是不完整的，要么就是多出来一块，这种情况即发生了拆包和粘包。拆包和粘包的问题导致接收端在处理的时候会非常困难，因为无法区分一个完整的数据包。

28、TCP 黏包是怎么产生的?

发送方产生粘包

采用 TCP 协议传输数据的客户端与服务器经常是保持一个长连接的状态（一次连接发一次数据不存在粘包），双方在连接不断开的情况下，可以一直传输数据。但当发送的数据包过于的小时，那么 TCP 协议默认的会启用 Nagle 算法，将这些较小的数据包进行合并发送（缓冲区数据发送是一个堆压的过程）；这个合并过程就是在发送缓冲区中进行的，也就是说数据发送出来它已经是粘包的状态了。

接收方产生粘包

接收方采用 TCP 协议接收数据时的过程是这样的：数据到接收方，从网络模型的下方传递至传输层，传输层的 TCP 协议处理是将其放置接收缓冲区，然后由应用层来主动获取（C 语言用 recv、read 等函数）；这时会出现一个问题，就是我们在程序中调用的读取数据函数不能及时的把缓冲区中的数据拿出来，而下一个数据又到来并有一部分放入的缓冲区末尾，等我们读取数据时就是一个粘包。（放数据的速度 > 应用层拿数据速度）

29、怎么解决拆包和粘包?

分包机制一般有两个通用的解决方法：

1. 特殊字符控制；
2. 在包头首都添加数据包的长度。

如果使用 netty 的话，就有专门的编码器和解码器解决拆包和粘包问题了。

tips：UDP 没有粘包问题，但是有丢包和乱序。不完整的包是不会有的，收到的都是完全正确的包。传送的数据单位协议是 UDP 报文或用户数据报，发送的时候既不合并，也不拆分。

30、forward 和 redirect 的区别?

Forward 和 Redirect 代表了两种请求转发方式：直接转发和间接转发。

直接转发方式（Forward）：客户端和浏览器只发出一次请求，Servlet、HTML、JSP 或其它信息资源，由第二个信息资源响应该请求，在请求对象 request 中，保存的对象对于每个信息资源是共享的。

间接转发方式（Redirect）：实际是两次 HTTP 请求，服务器端在响应第一次请求的时候，让浏览器再向另外一个 URL 发出请求，从而达到转发的目的。

- 举个通俗的例子：

直接转发就相当于：“A 找 B 借钱，B 说没有，B 去找 C 借，借到借不到都会把消息传递给 A”；

间接转发就相当于："A 找 B 借钱，B 说没有，让 A 去找 C 借"。

31、HTTP 方法有哪些?

客户端发送的 请求报文 第一行为请求行，包含了方法字段。

1. GET：获取资源，当前网络中绝大部分使用的都是 GET；
2. HEAD：获取报文首部，和 GET 方法类似，但是不返回报文实体主体部分；
3. POST：传输实体主体
4. PUT：上传文件，由于自身不带验证机制，任何人都可以上传文件，因此存在安全性问题，一般不使用该方法。
5. PATCH：对资源进行部分修改。PUT 也可以用于修改资源，但是只能完全替代原始资源，PATCH 允许部分修改。
6. OPTIONS：查询指定的 URL 支持的方法；
7. CONNECT：要求在与代理服务器通信时建立隧道。使用 SSL（Secure Sockets Layer，安全套接层）和 TLS（Transport Layer Security，传输层安全）协议把通信内容加密后经网络隧道传输。
8. TRACE：追踪路径。服务器会将通信路径返回给客户端。发送请求时，在 Max-Forwards 首部字段中填入数值，每经过一个服务器就会减 1，当数值为 0 时就停止传输。通常不会使用 TRACE，并且它容易受到 XST 攻击（Cross-Site Tracing，跨站追踪）。

32、在浏览器中输入 URL 地址到显示主页的过程?

1. DNS 解析：浏览器查询 DNS，获取域名对应的 IP 地址：具体过程包括浏览器搜索自身的 DNS 缓存、搜索操作系统的 DNS 缓存、读取本地的 Host 文件和向本地 DNS 服务器进行查询等。对于向本地 DNS 服务器进行查询，如果要查询的域名包含在本地配置区域资源中，则返回解析结果给客户机，完成域名解析(此解析具有权威性)；如果要查询的域名不由本地 DNS 服务器区域解析，但该服务器已缓存了此网址映射关系，则调用这个 IP 地址映射，完成域名解析（此解析不具有权威性）。如果本地域名服务器并未缓存该网址映射关系，那么将根据其设置发起递归查询或者迭代查询；
2. TCP 连接：浏览器获得域名对应的 IP 地址以后，浏览器向服务器请求建立链接，发起三次握手；
3. 发送 HTTP 请求：TCP 连接建立起来后，浏览器向服务器发送 HTTP 请求；
4. 服务器处理请求并返回 HTTP 报文：服务器接收到这个请求，并根据路径参数映射到特定的请求处理器进行处理，并将处理结果及相应的视图返回给浏览器；
5. 浏览器解析渲染页面：浏览器解析并渲染视图，若遇到对 js 文件、css 文件及图片等静态资源的引用，则重复上述步骤并向服务器请求这些资源；浏览器根据其请求到的资源、数据渲染页面，最终向用户呈现一个完整的页面。
6. 连接结束。

33、DNS 的解析过程?

1. 主机向本地域名服务器的查询一般都是采用递归查询。所谓递归查询就是：如果主机所询问的本地域名服务器不知道被查询的域名的 IP 地址，那么本地域名服务器就以 DNS 客户的身份，向根域名服务器继续发出查询请求报文(即替主机继续查询)，而不是让主机自己进行下一步查询。因此，递归查询返回的查询结果或者是所要查询的 IP 地址，或者是报错，表示无法查询到所需的 IP 地址。
2. 本地域名服务器向根域名服务器的查询的迭代查询。迭代查询的特点：当根域名服务器收到本地域名服务器发出的迭代查询请求报文时，要么给出所要查询的 IP 地址，要么告诉本地服务器：“你下一步应当向哪一个域名服务器进行查询”。然后让本地服务器进行后续的查询。根域名服务器通常是把自己知道的顶级域名服务器的 IP 地址告诉本地域名服务器，让本地域名服务器再向顶级域名服务器查询。顶级域名服务器在收到本地域名服务器的查询请求后，要么给出所要查询的 IP 地址，要么告诉本地服务器下一步应当向哪一个权限域名服务器进行查询。最后，本地域名服务器得到了所要解析的 IP 地址或报错，然后把这个结果返回给发起查询的主机。

34、谈谈你对域名缓存的了解?

为了提高 DNS 查询效率，并减轻服务器的负荷和减少因特网上的 DNS 查询报文数量，在域名服务器中广泛使用了高速缓存，用来存放最近查询过的域名以及从何处获得域名映射信息的记录。

由于名字到地址的绑定并不经常改变，为保持高速缓存中的内容正确，域名服务器应为每项内容设置计时器并处理超过合理时间的项（例如：每个项目两天）。当域名服务器已从缓存中删去某项信息后又被请求查询该项信息，就必须重新到授权管理该项的域名服务器绑定信息。当权限服务器回答一个查询请求时，在响应中都指明绑定有效存在的时间值。增加此时间值可减少网络开销，而减少此时间值可提高域名解析的正确性。

不仅在本地域名服务器中需要高速缓存，在主机中也需要。许多主机在启动时从本地服务器下载名字和地址的全部数据库，维护存放自己最近使用的域名的高速缓存，并且只在从缓存中找不到名字时才使用域名服务器。维护本地域名服务器数据库的主机应当定期地检查域名服务器以获取新的映射信息，而且主机必须从缓存中删除无效的项。由于域名改动并不频繁，大多数网点不需花精力就能维护数据库的一致性。

35、谈下你对 HTTP 长连接和短连接的理解？分别应用于哪些场景?

在 HTTP/1.0 中默认使用短连接。也就是说，客户端和服务器每进行一次 HTTP 操作，就建立一次连接，任务结束就中断连接。当客户端浏览器访问的某个 HTML 或其他类型的 Web 页中包含有其他的 Web 资源（如：JavaScript 文件、图像文件、CSS 文件等），每遇到这样一个 Web 资源，浏览器就会重新建立一个 HTTP 会话。

而从 HTTP/1.1 起，默认使用长连接，用以保持连接特性。使用长连接的 HTTP 协议，会在响应头加入这行代码：

```
Connection:keep-alive
```

在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输 HTTP 数据的 TCP 连接不会关闭，客户端再次访问这个服务器时，会继续使用这一条已经建立的连接。

Keep-Alive 不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如：Apache）中设定这个时间。实现长连接需要客户端和服务端都支持长连接。

36、HTTPS 的工作过程?

1、 客户端发送自己支持的加密规则给服务器，代表告诉服务器要进行连接了；

2、 服务器从中选出一套加密算法和 hash 算法以及自己的身份信息（地址等）以证书的形式发送给浏览器，证书中包含服务器信息，加密公钥，证书的办法机构；

3、客户端收到网站的证书之后要做下面的事情：

- 验证证书的合法性；
- 果验证通过证书，浏览器会生成一串随机数，并用证书中的公钥进行加密；
- 用约定好的 hash 算法计算握手消息，然后用生成的密钥进行加密，然后一起发送给服务器。

4、服务器接收到客户端传送来的信息，要做下面的事情：

- 4.1 用私钥解析出密码，用密码解析握手消息，验证 hash 值是否和浏览器发来的一致；
- 4.2 使用密钥加密消息；

5、如果计算法 hash 值一致，握手成功。

37、HTTP 和 HTTPS 的区别?

1. 开销：HTTPS 协议需要到 CA 申请证书，一般免费证书很少，需要交费；
2. 资源消耗：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，需要消耗更多的 CPU 和内存资源；
3. 端口不同：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；
4. 安全性：HTTP 的连接很简单，是无状态的；HTTPS 协议是由 TSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 HTTP 协议安全

38、HTTPS 的优缺点?

优点：

1. 使用 HTTPS 协议可认证用户和服务器，确保数据发送到正确的客户机和服务器；
2. 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，可防止数据在传输过程中不被窃取、改变，确保数据的完整性；

3. HTTPS 是现行架构下最安全的解决方案，虽然不是绝对安全，但它大幅增加了中间人攻击的成本。

缺点：

1. HTTPS 协议握手阶段比较费时，会使页面的加载时间延长近 50%，增加 10% 到 20% 的耗电；
2. HTTPS 连接缓存不如 HTTP 高效，会增加数据开销和功耗，甚至已有的安全措施也会因此而受到影响；
3. SSL 证书需要钱，功能越强大的证书费用越高，个人网站、小网站没有必要一般不会用；
4. SSL 证书通常需要绑定 IP，不能在同一 IP 上绑定多个域名，IPv4 资源不可能支撑这个消耗；
5. HTTPS 协议的加密范围也比较有限，在黑客攻击、拒绝服务攻击、服务器劫持等方面几乎起不到什么作用。最关键的，SSL 证书的信用链体系并不安全，特别是在某些国家可以控制 CA 根证书的情况下，中间人攻击一样可行。

39、什么是数字签名?

为了避免数据在传输过程中被替换，比如黑客修改了你的报文内容，但是你并不知道，所以我们让发送端做一个数字签名，把数据的摘要消息进行一个加密，比如 MD5，得到一个签名，和数据一起发送。然后接收端把数据摘要进行 MD5 加密，如果和签名一样，则说明数据确实是真的。

40、什么是数字证书?

对称加密中，双方使用公钥进行解密。虽然数字签名可以保证数据不被替换，但是数据是由公钥加密的，如果公钥也被替换，则仍然可以伪造数据，因为用户不知道对方提供的公钥其实是假的。所以为了保证发送方的公钥是真的，CA 证书机构会负责颁发一个证书，里面的公钥保证是真的，用户请求服务器时，服务器将证书发给用户，这个证书是经由系统内置证书的备案的。

操作系统

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**Java面试题**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**Java面试题**」，即可获取最新版 PDF。

另外，也建议大家来网站读，因为如果有错误，我会及时在网站更改，PDF 很难及时更新。

在线网站：https://www.iamshuaidi.com

1、简单说下你对并发和并行的理解?

1. 并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔发生；
2. 并行是在不同实体上的多个事件，并发是在同一实体上的多个事件；

2、同步、异步、阻塞、非阻塞的概念

同步：当一个同步调用发出后，调用者要一直等待返回结果。通知后，才能进行后续的执行。

异步：当一个异步过程调用发出后，调用者不能立刻得到返回结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。

阻塞：是指调用结果返回前，当前线程会被挂起，即阻塞。

非阻塞：是指即使调用结果没返回，也不会阻塞当前线程。

3、进程和线程的基本概念

进程：进程是系统进行资源分配和调度的一个独立单位，是系统中的并发执行的单位。

线程：线程是进程的一个实体，也是 CPU 调度和分派的基本单位，它是比进程更小的能独立运行的基本单位，有时又被称为轻权进程或轻量级进程。

4、进程与线程的区别?

1. 进程是资源分配的最小单位，而线程是 CPU 调度的最小单位；
2. 创建进程或撤销进程，系统都要为之分配或回收资源，操作系统开销远大于创建或撤销线程时的开销；
3. 不同进程地址空间相互独立，同一进程内的线程共享同一地址空间。一个进程的线程在另一个进程内是不可见的；
4. 进程间不会相互影响，而一个线程挂掉将可能导致整个进程挂掉；

5、为什么有了进程，还要有线程呢?

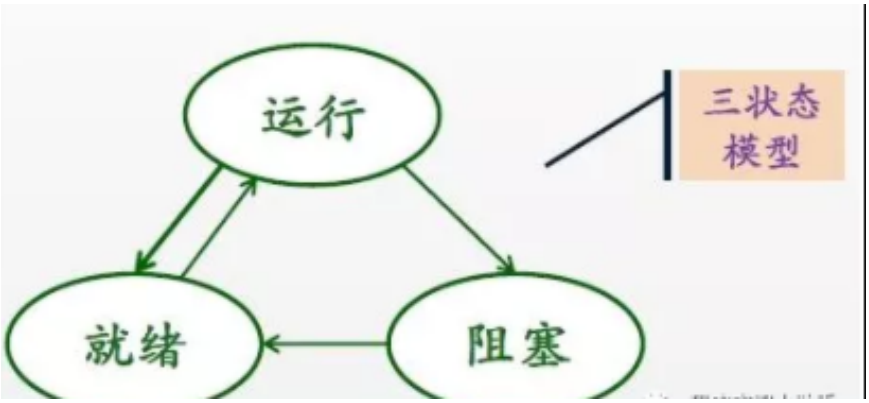
进程可以使多个程序并发执行，以提高资源的利用率和系统的吞吐量，但是其带来了一些缺点：

1. 进程在同一时间只能干一件事情；
2. 进程在执行的过程中如果阻塞，整个进程就会被挂起，即使进程中有些工作不依赖与等待的资源，仍然不会执行。

基于以上的缺点，操作系统引入了比进程粒度更小的线程，作为并发执行的基本单位，从而减少程序在并发执行时所付出的时间和空间开销，提高并发性能。

6、进程的状态转换

进程包括三种状态：就绪态、运行态和阻塞态。



1. 就绪 —> 执行：对就绪状态的进程，当进程调度程序按一种选定的策略从中选中一个就绪进程，为之分配了处理机后，该进程便由就绪状态变为执行状态；
2. 执行 —> 阻塞：正在执行的进程因发生某等待事件而无法执行，则进程由执行状态变为阻塞状态，如进程提出输入/输出请求而变成等待外部设备传输信息的状态，进程申请资源（主存空间或外部设备）得不到满足时变成等待资源状态，进程运行中出现了故障（程序出错或主存储器读写错等）变成等待干预状态等等；
3. 阻塞 —> 就绪：处于阻塞状态的进程，在其等待的事件已经发生，如输入/输出完成，资源得到满足或错误处理完毕时，处于等待状态的进程并不马上转入执行状态，而是先转入就绪状态，然后再由系统进程调度程序在适当的时候将该进程转为执行状态；
4. 执行 —> 就绪：正在执行的进程，因时间片用完而被暂停执行，或在采用抢先式优先级调度算法的系统中,当有更高优先级的进程要运行而被迫让出处理机时，该进程便由执行状态转变为就绪状态。

7、进程间的通信方式有哪些?

进程间通信（IPC，InterProcess Communication）是指在不同进程之间传播或交换信息。IPC 的方式通常有管道（包括无名管道和命名管道）、消息队列、信号量、共享存储、Socket、Streams 等。其中 Socket 和 Streams 支持不同主机上的两个进程 IPC。

管道

1. 它是半双工的，具有固定的读端和写端；
2. 它只能用于父子进程或者兄弟进程之间的进程的通信；
3. 它可以看成是一种特殊的文件，对于它的读写也可以使用普通的 read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

命名管道

1. FIFO 可以在无关的进程之间交换数据，与无名管道不同；
2. FIFO 有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。

消息队列

1. 消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符 ID 来标识；
2. 消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级；
3. 消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除；
4. 消息队列可以实现消息的随机查询，消息不一定要以先进先出的次序读取，也可以按消息的类型读取。

信号量

1. 信号量（semaphore）是一个计数器。用于实现进程间的互斥与同步，而不是用于存储进程间通信数据；
2. 信号量用于进程间同步，若要在进程间传递数据需要结合共享内存；
3. 信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作；

4. 每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数；
5. 支持信号量组。

共享内存

1. 共享内存（Shared Memory），指两个或多个进程共享一个给定的存储区；
2. 共享内存是最快的一种 IPC，因为进程是直接对内存进行存取。

8、进程的调度算法有哪些?

调度算法是指：根据系统的资源分配策略所规定的资源分配算法。常用的调度算法有：先来先服务调度算法、时间片轮转调度法、短作业优先调度算法、最短剩余时间优先、高响应比优先调度算法、优先级调度算法等等。

- **先来先服务调度算法**

先来先服务调度算法是一种最简单的调度算法，也称为先进先出或严格排队方案。当每个进程就绪后，它加入就绪队列。当前正运行的进程停止执行，选择在就绪队列中存在时间最长的进程运行。该算法既可以用于作业调度，也可以用于进程调度。先来先去服务比较适合于常作业（进程），而不利于段作业（进程）。

- **时间片轮转调度算法**

时间片轮转调度算法主要适用于分时系统。在这种算法中，系统将所有就绪进程按到达时间的先后次序排成一个队列，进程调度程序总是选择就绪队列中第一个进程执行，即先来先服务的原则，但仅能运行一个时间片。

- **短作业优先调度算法**

短作业优先调度算法是指对短作业优先调度的算法，从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。 短作业优先调度算法是一个非抢占策略，他的原则是下一次选择预计处理时间最短的进程，因此短进程将会越过长作业，跳至队列头。

- **最短剩余时间优先调度算法**

最短剩余时间是针对最短进程优先增加了抢占机制的版本。在这种情况下，进程调度总是选择预期剩余时间最短的进程。当一个进程加入到就绪队列时，他可能比当前运行的进程具有更短的剩余时间，因此只要新进程就绪，调度程序就能可能抢占当前正在运行的进程。像最短进程优先一样，调度程序正在执行选择函数是必须有关于处理时间的估计，并且存在长进程饥饿的危险。

- **高响应比优先调度算法**

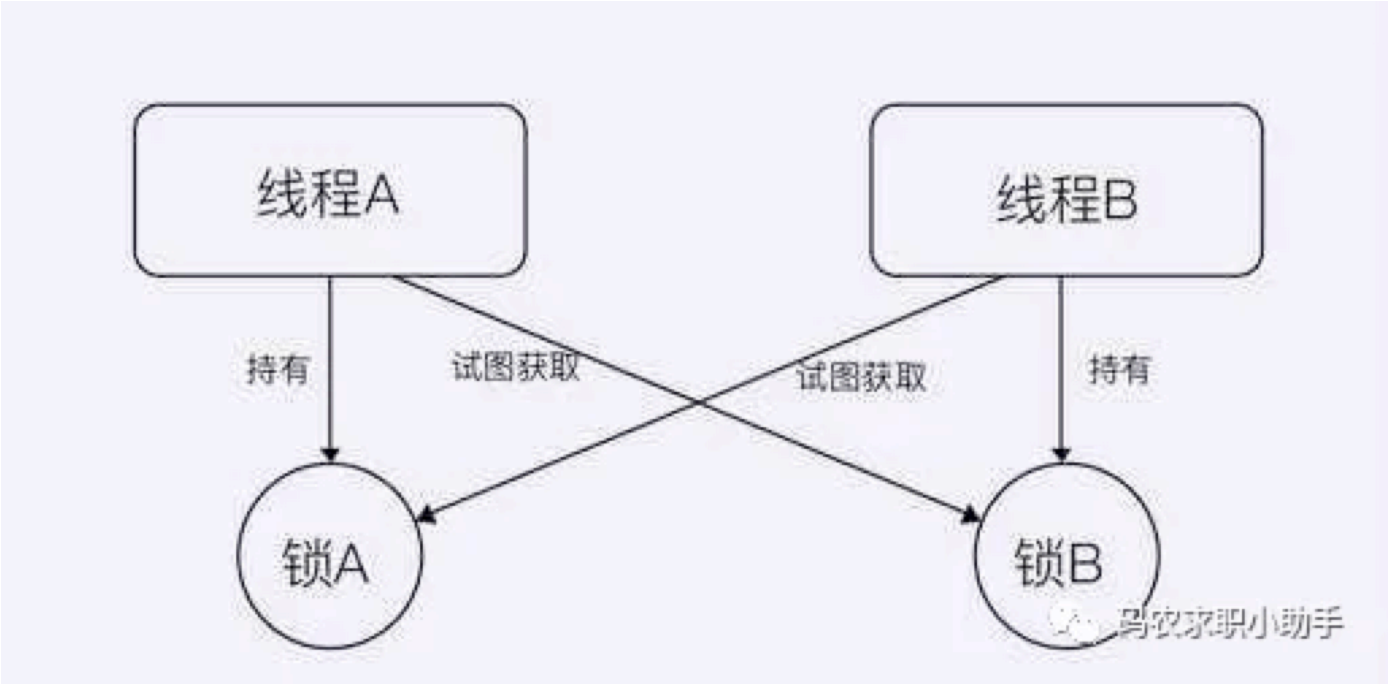
高响应比优先调度算法主要用于作业调度，该算法是对 先来先服务调度算法和短作业优先调度算法的一种综合平衡，同时考虑每个作业的等待时间和估计的运行时间。在每次进行作业调度时，先计算后备作业队列中每个作业的响应比，从中选出响应比最高的作业投入运行。

- **优先级调度算法**

优先级调度算法每次从后备作业队列中选择优先级最高的一个或几个作业，将它们调入内存，分配必要的资源，创建进程并放入就绪队列。在进程调度中，优先级调度算法每次从就绪队列中选择优先级最高的进程，将处理机分配给它，使之投入运行。

9、什么是死锁?

死锁，是指多个进程在运行过程中因争夺资源而造成的一种僵局，当进程处于这种僵持状态时，若无外力作用，它们都将无法再向前推进。 如下图所示：如果此时有一个线程 A，已经持有了锁 A，但是试图获取锁 B，线程 B 持有锁 B，而试图获取锁 A，这种情况下就会产生死锁。



10、产生死锁的原因?

由于系统中存在一些不可剥夺资源，而当两个或两个以上进程占有自身资源，并请求对方资源时，会导致每个进程都无法向前推进，这就是死锁。

- **竞争资源**

例如：系统中只有一台打印机，可供进程 A 使用，假定 A 已占用了打印机，若 B 继续要求打印机打印将被阻塞。

系统中的资源可以分为两类：

1. 可剥夺资源：是指某进程在获得这类资源后，该资源可以再被其他进程或系统剥夺，CPU 和主存均属于可剥夺性资源；
2. 不可剥夺资源，当系统把这类资源分配给某进程后，再不能强行收回，只能在进程用完后自行释放，如磁带机、打印机等。

- **进程推进顺序不当**

例如：进程 A 和 进程 B 互相等待对方的数据。

11、死锁产生的必要条件?

1. 互斥条件：进程要求对所分配的资源进行排它性控制，即在一段时间内某资源仅为一进程所占用。
2. 请求和保持条件：当进程因请求资源而阻塞时，对已获得的资源保持不放。
3. 不剥夺条件：进程已获得的资源在未使用完之前，不能剥夺，只能在使用完时由自己释放。
4. 环路等待条件：在发生死锁时，必然存在一个进程--资源的环形链。

12、解决死锁的基本方法?

1. 预防死锁
2. 避免死锁
3. 检测死锁
4. 解除死锁

13、怎么预防死锁?

1. 破坏请求条件：一次性分配所有资源，这样就不会再有请求了；
2. 破坏请保持条件：只要有一个资源得不到分配，也不给这个进程分配其他的资源：
3. 破坏不可剥夺条件：当某进程获得了部分资源，但得不到其它资源，则释放已占有的资源；
4. 破坏环路等待条件：系统给每类资源赋予一个编号，每一个进程按编号递增的顺序请求资源，释放则相反。

14、怎么避免死锁?

1. 安全状态

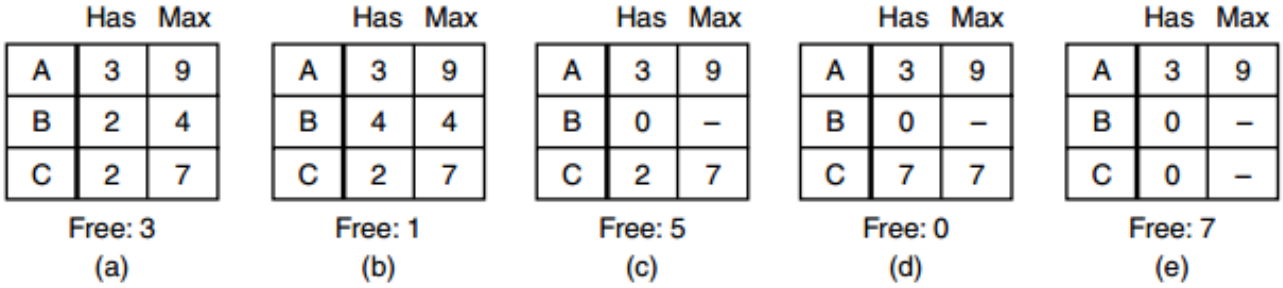


Figure 6-9. Demonstration that the state in (a) is safe.

图 a 的第二列 Has 表示已拥有的资源数，第三列 Max 表示总共需要的资源数，Free 表示还有可以使用的资源数。从图 a 开始出发，先让 B 拥有所需的所有资源（图 b），运行结束后释放 B，此时 Free 变为 5（图 c）；接着以同样的方式运行 C 和 A，使得所有进程都能成功运行，因此可以称图 a 所示的状态时安全的。

定义：如果没有死锁发生，并且即使所有进程突然请求对资源的最大需求，也仍然存在某种调度次序能够使得每一个进程运行完毕，则称该状态是安全的。

安全状态的检测与死锁的检测类似，因为安全状态必须要求不能发生死锁。下面的银行家算法与死锁检测算法非常类似，可以结合着做参考对比。

2. 单个资源的银行家算法

一个小城镇的银行家，他向一群客户分别承诺了一定的贷款额度，算法要做的是判断对请求的满足是否会进入不安全状态，如果是，就拒绝请求；否则予以分配。

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

Figure 6-11. Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

上图 c 为不安全状态，因此算法会拒绝之前的请求，从而避免进入图 c 中的状态。

3. 多个资源的银行家算法

Process	Tape drives	Plotters	Printers	Blu-rays
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Printers	Blu-rays
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still assigned

E = (6342)
P = (5322)
A = (1020)

Figure 6-12. The banker's algorithm with multiple resources.

上图中有五个进程，四个资源。左边的图表示已经分配的资源，右边的图表示还需要分配的资源。最右边的 E、P 以及 A 分别表示：总资源、已分配资源以及可用资源，注意这三个为向量，而不是具体数值，例如 A=(1020)，表示 4 个资源分别还剩下 1/0/2/0。

检查一个状态是否安全的算法如下：

- 查找右边的矩阵是否存在一行小于等于向量 A。如果不存在这样的行，那么系统将会发生死锁，状态是不安全的。
- 假若找到这样一行，将该进程标记为终止，并将其已分配资源加到 A 中。
- 重复以上两步，直到所有进程都标记为终止，则状态时安全的。

如果一个状态不是安全的，需要拒绝进入这个状态。

15、怎么解除死锁?

1. 资源剥夺：挂起某些死锁进程，并抢占它的资源，将这些资源分配给其他死锁进程（但应该防止被挂起的进程长时间得不到资源）；
2. 撤销进程：强制撤销部分、甚至全部死锁进程并剥夺这些进程的资源（撤销的原则可以按进程优先级和撤销进程代价的高低进行）；
3. 进程回退：让一个或多个进程回退到足以避免死锁的地步。进程回退时自愿释放资源而不是被剥夺。要求系统保持进程的历史信息，设置还原点。

16、什么是缓冲区溢出？有什么危害?

缓冲区为暂时置放输出或输入资料的内存。缓冲区溢出是指当计算机向缓冲区填充数据时超出了缓冲区本身的容量，溢出的数据覆盖在合法数据上。造成缓冲区溢出的主要原因是程序中没有仔细检查用户输入是否合理。计算机中，缓冲区溢出会造成的危害主要有以下两点：程序崩溃导致拒绝服务和跳转并且执行一段恶意代码。

17、分页与分段的区别?

1. 段是信息的逻辑单位，它是根据用户的需要划分的，因此段对用户是可见的 ；页是信息的物理单位，是为了管理主存的方便而划分的，对用户是透明的；
2. 段的大小不固定，有它所完成的功能决定；页大大小固定，由系统决定；
3. 段向用户提供二维地址空间；页向用户提供的是一维地址空间；
4. 段是信息的逻辑单位，便于存储保护和信息的共享，页的保护和共享受到限制。

18、物理地址、逻辑地址、虚拟内存的概念

1. 物理地址：它是地址转换的最终地址，进程在运行时执行指令和访问数据最后都要通过物理地址从主存中存取，是内存单元真正的地址。
2. 逻辑地址：是指计算机用户看到的地址。例如：当创建一个长度为 100 的整型数组时，操作系统返回一个逻辑上的连续空间：指针指向数组第一个元素的内存地址。由于整型元素的大小为 4 个字节，故第二个元素的地址时起始地址加 4，以此类推。事实上，逻辑地址并不一定是元素存储的真实地址，即数组元素的物理地址（在内存条中所处的位置），并非是连续的，只是操作系统通过地址映射，将逻辑地址映射成连续的，这样更符合人们的直观思维。
3. 虚拟内存：是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。

19、页面置换算法有哪些?

请求调页，也称按需调页，即对不在内存中的“页”，当进程执行时要用时才调入，否则有可能到程序结束时也不会调入。而内存中给页面留的位置是有限的，在内存中以帧为单位放置页面。为了防止请求调页的过程出现过多的内存页面错误（即需要的页面当前不在内存中，需要从硬盘中读数据，也即需要做页面的替换）而使得程序执行效率下降，我们需要设计一些页面置换算法，页面按照这些算法进行相互替换时，可以尽量达到较低的错误率。常用的页面置换算法如下：

- **先进先出置换算法（FIFO）**

先进先出，即淘汰最早调入的页面。

- **最佳置换算法（OPT）**

选未来最远将使用的页淘汰，是一种最优的方案，可以证明缺页数最小。

- **最近最久未使用（LRU）算法**

即选择最近最久未使用的页面予以淘汰

- **时钟（Clock）置换算法**

时钟置换算法也叫最近未用算法 NRU（Not RecentlyUsed）。该算法为每个页面设置一位访问位，将内存中的所有页面都通过链接指针链成一个循环队列。

20、谈谈你对动态链接库和静态链接库的理解?

静态链接就是在编译链接时直接将需要的执行代码拷贝到调用处，优点就是在程序发布的时候就不需要的依赖库，也就是不再需要带着库一块发布，程序可以独立执行，但是体积可能会相对大一些。

动态链接就是在编译的时候不直接拷贝可执行代码，而是通过记录一系列符号和参数，在程序运行或加载时将这些信息传递给操作系统，操作系统负责将需要的动态库加载到内存中，然后程序在运行到指定的代码时，去共享执行内存中已经加载的动态库可执行代码，最终达到运行时连接的目的。优点是多个程序可以共享同一段代码，而不需要在磁盘上存储多个拷贝，缺点是由于是运行时加载，可能会影响程序的前期执行性能

21、外中断和异常有什么区别?

外中断是指由 CPU 执行指令以外的事件引起，如 I/O 完成中断，表示设备输入/输出处理已经完成，处理器能够发送下一个输入/输出请求。此外还有时钟中断、控制台中断等。

而异常时由 CPU 执行指令的内部事件引起，如非法操作码、地址越界、算术溢出等。

22、一个程序从开始运行到结束的完整过程，你能说出来多少?

四个过程：

（1）预编译 主要处理源代码文件中的以“#”开头的预编译指令。处理规则见下

1、删除所有的#define，展开所有的宏定义。

2、处理所有的条件预编译指令，如“#if”、“#endif”、“#ifdef”、“#elif”和“#else”。

3、处理“#include”预编译指令，将文件内容替换到它的位置，这个过程是递归进行的，文件中包含其他 文件。

4、删除所有的注释，“//”和“/**/”。

5、保留所有的#pragma 编译器指令，编译器需要用到他们，如：#pragma once 是为了防止有文件被重 复引用。

6、添加行号和文件标识，便于编译时编译器产生调试用的行号信息，和编译时产生编译错误或警告是 能够显示行号。

（2）编译 把预编译之后生成的xxx.i或xxx.ii文件，进行一系列词法分析、语法分析、语义分析及优化后，生成相应的汇编代码文件。

1、词法分析：利用类似于“有限状态机”的算法，将源代码程序输入到扫描机中，将其中的字符序列分割成一系列的记号。

2、语法分析：语法分析器对由扫描器产生的记号，进行语法分析，产生语法树。由语法分析器输出的语法树是一种以表达式为节点的树。

3、语义分析：语法分析器只是完成了对表达式语法层面的分析，语义分析器则对表达式是否有意义进行判断，其分析的语义是静态语义——在编译期能分期的语义，相对应的动态语义是在运行期才能确定的语义。

4、优化：源代码级别的一个优化过程。

5、目标代码生成：由代码生成器将中间代码转换成目标机器代码，生成一系列的代码序列——汇编语言表示。

6、目标代码优化：目标代码优化器对上述的目标机器代码进行优化：寻找合适的寻址方式、使用位移来替代乘法运算、删除多余的指令等。

（3）汇编

将汇编代码转变成机器可以执行的指令(机器码文件)。 汇编器的汇编过程相对于编译器来说更简单，没有复杂的语法，也没有语义，更不需要做指令优化，只是根据汇编指令和机器指令的对照表一一翻译过来，汇编过程有汇编器as完成。

经汇编之后，产生目标文件(与可执行文件格式几乎一样)xxx.o(Linux下)、xxx.obj(Windows下)。

（4）链接

将不同的源文件产生的目标文件进行链接，从而形成一个可以执行的程序。链接分为静态链接和动态链接：

1、静态链接： 函数和数据被编译进一个二进制文件。在使用静态库的情况下，在编译链接可执行文件时，链接器从库中复制这些函数和数据并把它们和应用程序的其它模块组合起来创建最终的可执行文件。 空间浪费：因为每个可执行程序中对所有需要的目标文件都要有一份副本，所以如果多个程序对同一个目标文件都有依赖，会出现同一个目标文件都在内存存在多个副本； 更新困难：每当库函数的代码修改了，这个时候就需要重新进行编译链接形成可执行程序。

运行速度快：但是静态链接的优点就是，在可执行程序中已经具备了所有执行程序所需要的任何东西，在执行的时候运行速度快。

2、动态链接： 动态链接的基本思想是把程序按照模块拆分成各个相对独立部分，在程序运行时才将它们链接在一起形成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。

共享库：就是即使需要每个程序都依赖同一个库，但是该库不会像静态链接那样在内存中存在多份副本，而是这多个程序在执行时共享同一份副本；

更新方便：更新时只需要替换原来的目标文件，而无需将所有的程序再重新链接一遍。当程序下一次运行时，新版本的目标文件会被自动加载到内存并且链接起来，程序就完成了升级的目标。

性能损耗：因为把链接推迟到了程序运行时，所以每次执行程序都需要进行链接，所以性能会有一定损失。

23、介绍一下几种典型的锁?

读写锁

- 多个读者可以同时进行读
- 写者必须互斥（只允许一个写者写，也不能读者写者同时进行）
- 写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）

互斥锁

一次只能一个线程拥有互斥锁，其他线程只有等待

互斥锁是在抢锁失败的情况下主动放弃CPU进入睡眠状态直到锁的状态改变时再唤醒，而操作系统负责线程调度，为了实现锁的状态发生改变时唤醒阻塞的线程或者进程，需要把锁交给操作系统管理，所以互斥锁在加锁操作时涉及上下文的切换。互斥锁实际的效率还是可以让人接受的，加锁的时间大概100ns左右，而实际上互斥锁的一种可能的实现是先自旋一段时间，当自旋的时间超过阀值之后再将线程投入睡眠中，因此在并发运算中使用互斥锁（每次占用锁的时间很短）的效果可能不亚于使用自旋锁

条件变量

互斥锁一个明显的缺点是他只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，他常和互斥锁一起使用，以免出现竞态条件。当条件不满足时，线程往往解开相应的互斥锁并阻塞线程然后等待条件发生变化。一旦其他的某个线程改变了条件变量，他将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。总的来说**互斥锁是线程间互斥的机制，条件变量则是同步机制。**

自旋锁

如果进线程无法取得锁，进线程不会立刻放弃CPU时间片，而是一直循环尝试获取锁，直到获取为止。如果别的线程长时期占有锁，那么自旋就是在浪费CPU做无用功，但是自旋锁一般应用于加锁时间很短的场景，这个时候效率比较高。

24、什么是用户态和内核态

用户态和内核态是操作系统的两种运行状态。

- `内核态`：处于内核态的 CPU 可以访问任意的数据，包括外围设备，比如网卡、硬盘等，处于内核态的 CPU 可以从一个程序切换到另外一个程序，并且占用 CPU 不会发生抢占情况，一般处于特权级 0 的状态我们称之为内核态。
- `用户态`：处于用户态的 CPU 只能受限的访问内存，并且不允许访问外围设备，用户态下的 CPU 不允许独占，也就是说 CPU 能够被其他程序获取。

> 那么为什么要有用户态和内核态呢?

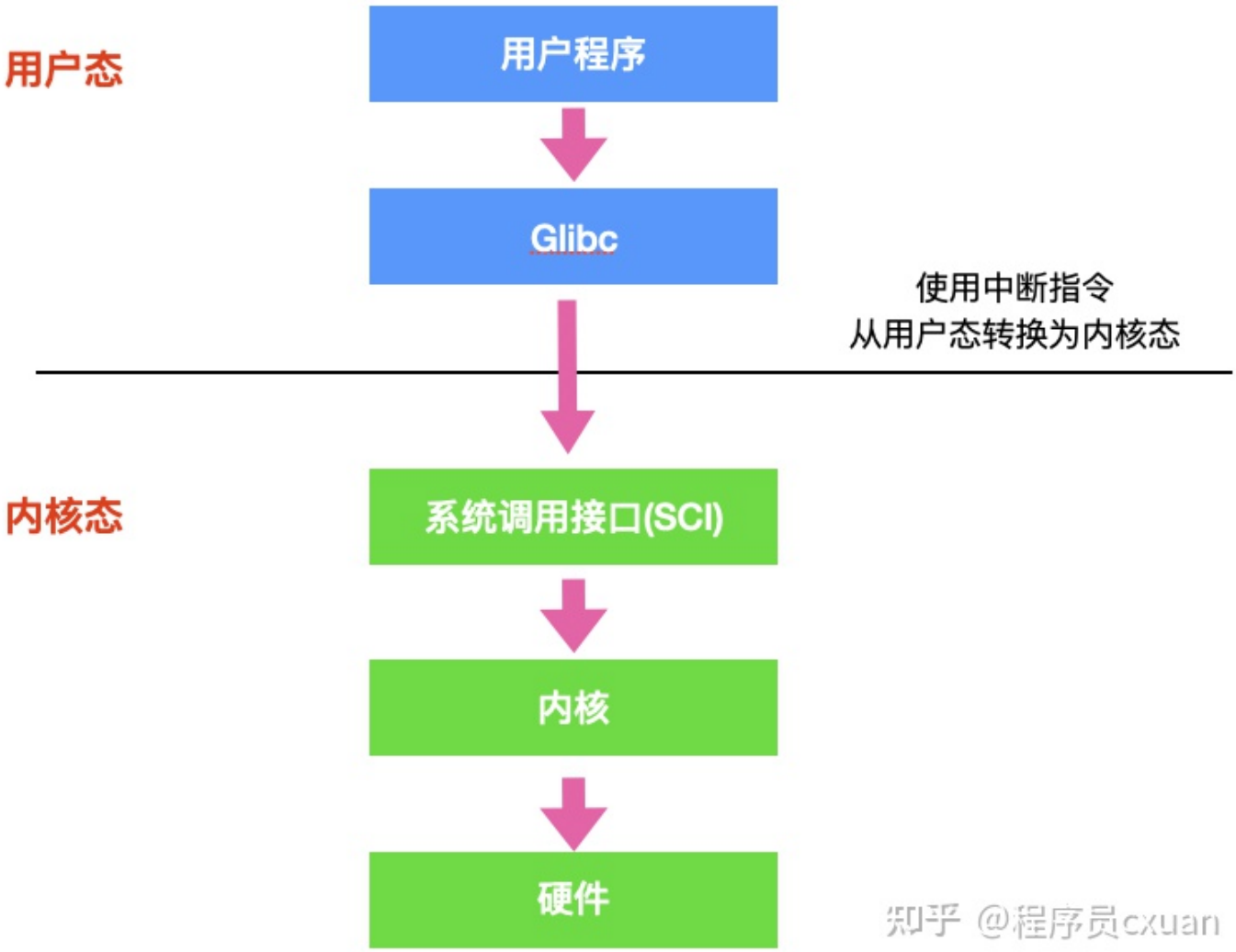
这个主要是访问能力的限制的考量，计算机中有一些比较危险的操作，比如设置时钟、内存清理，这些都需要在内核态下完成，如果随意进行这些操作，那你的系统得崩溃多少次。

25、用户态和内核态是如何切换的?

所有的用户进程都是运行在用户态的，但是我们上面也说了，用户程序的访问能力有限，一些比较重要的比如从硬盘读取数据，从键盘获取数据的操作则是内核态才能做的事情，而这些数据却又对用户程序来说非常重要。所以就涉及到两种模式下的转换，即**用户态 -> 内核态 -> 用户态**，而唯一能够做这些操作的只有 `系统调用`，而能够执行系统调用的就只有 `操作系统`。

一般用户态 -> 内核态的转换我们都称之为 trap 进内核，也被称之为 `陷阱指令(trap instruction)`。

他们的工作流程如下：



- 首先用户程序会调用 `glibc` 库，glibc 是一个标准库，同时也是一套核心库，库中定义了很多关键 API。
- glibc 库知道针对不同体系结构调用 `系统调用` 的正确方法，它会根据体系结构应用程序的二进制接口设置用户进程传递的参数，来准备系统调用。
- 然后，glibc 库调用 `软件中断指令(SWI)` ，这个指令通过更新 `CPSR` 寄存器将模式改为超级用户模式，然后跳转到地址 `0x08` 处。
- 到目前为止，整个过程仍处于用户态下，在执行 SWI 指令后，允许进程执行内核代码，MMU 现在允许内核虚拟内存访问
- 从地址 0x08 开始，进程执行加载并跳转到中断处理程序，这个程序就是 ARM 中的 `vector_swi()` 。
- 在 vector_swi() 处，从 SWI 指令中提取系统调用号 SCNO，然后使用 SCNO 作为系统调用表 `sys_call_table` 的索引，调转到系统调用函数。
- 执行系统调用完成后，将还原用户模式寄存器，然后再以用户模式执行。

26、进程终止的方式

进程的终止

进程在创建之后，它就开始运行并做完成任务。然而，没有什么事儿是永不停歇的，包括进程也一样。进程早晚会发生终止，但是通常是由于以下情况触发的

- `正常退出(自愿的)`
- `错误退出(自愿的)`
- `严重错误(非自愿的)`

- `被其他进程杀死(非自愿的)`

正常退出

多数进程是由于完成了工作而终止。当编译器完成了所给定程序的编译之后，编译器会执行一个系统调用告诉操作系统它完成了工作。这个调用在 UNIX 中是 `exit` ，在 Windows 中是 `ExitProcess` 。面向屏幕中的软件也支持自愿终止操作。字处理软件、Internet 浏览器和类似的程序中总有一个供用户点击的图标或菜单项，用来通知进程删除它锁打开的任何临时文件，然后终止。

错误退出

进程发生终止的第二个原因是发现严重错误，例如，如果用户执行如下命令

```
cc foo.c
```

为了能够编译 foo.c 但是该文件不存在，于是编译器就会发出声明并退出。在给出了错误参数时，面向屏幕的交互式进程通常并不会直接退出，因为这从用户的角度来说并不合理，用户需要知道发生了什么并想要进行重试，所以这时候应用程序通常会弹出一个对话框告知用户发生了系统错误，是需要重试还是退出。

严重错误

进程终止的第三个原因是由进程引起的错误，通常是由于程序中的错误所导致的。例如，执行了一条非法指令，引用不存在的内存，或者除数是 0 等。在有些系统比如 UNIX 中，进程可以通知操作系统，它希望自行处理某种类型的错误，在这类错误中，进程会收到信号（中断），而不是在这类错误出现时直接终止进程。

被其他进程杀死

第四个终止进程的原因是，某个进程执行系统调用告诉操作系统杀死某个进程。在 UNIX 中，这个系统调用是 kill。在 Win32 中对应的函数是 `TerminateProcess`（注意不是系统调用）。

27、 守护进程、僵尸进程和孤儿进程

守护进程

指在后台运行的，没有控制终端与之相连的进程。它独立于控制终端，周期性地执行某种任务。Linux的大多数服务器就是用守护进程的方式实现的，如web服务器进程http等

创建守护进程要点：

（1）让程序在后台执行。方法是调用fork（）产生一个子进程，然后使父进程退出。

（2）调用setsid（）创建一个新对话期。控制终端、登录会话和进程组通常是从父进程继承下来的，守护进程要摆脱它们，不受它们的影响，方法是调用setsid（）使进程成为一个会话组长。setsid（）调用成功后，进程成为新的会话组长和进程组长，并与原来的登录会话、进程组和控制终端脱离。

（3）禁止进程重新打开控制终端。经过以上步骤，进程已经成为一个无终端的会话组长，但是它可以重新申请打开一个终端。为了避免这种情况发生，可以通过使进程不再是会话组长来实现。再一次通过fork（）创建新的子进程，使调用fork的进程退出。

（4）关闭不再需要的文件描述符。子进程从父进程继承打开的文件描述符。如不关闭，将会浪费系统资源，造成进程所在的文件系统无法卸下以及引起无法预料的错误。首先获得最高文件描述符值，然后用一个循环程序，关闭0到最高文件描述符值的所有文件描述符。

（5）将当前目录更改为根目录。

（6）子进程从父进程继承的文件创建屏蔽字可能会拒绝某些许可权。为防止这一点，使用unmask（0）将屏蔽字清零。

（7）处理SIGCHLD信号。对于服务器进程，在请求到来时往往生成子进程处理请求。如果子进程等待父进程捕获状态，则子进程将成为僵尸进程（zombie），从而占用系统资源。如果父进程等待子进程结束，将增加父进程的负担，影响服务器进程的并发性能。在Linux下可以简单地将SIGCHLD信号的操作设为SIG_IGN。这样，子进程结束时不会产生僵尸进程。

孤儿进程

如果父进程先退出，子进程还没退出，那么子进程的父进程将变为init进程。（注：任何一个进程都必须有父进程）。

一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。

僵尸进程

如果子进程先退出，父进程还没退出，那么子进程必须等到父进程捕获到了子进程的退出状态才真正结束，否则这个时候子进程就成为僵尸进程。

设置**僵尸进程的目**的是维护子进程的信息，以便父进程在以后某个时候获取。这些信息至少包括进程ID，进程的终止状态，以及该进程使用的CPU时间，所以当终止子进程的父进程调用wait或waitpid时就可以得到这些信息。如果一个进程终止，而该进程有子进程处于僵尸状态，那么它的所有僵尸子进程的父进程ID将被重置为1（init进程）。继承这些子进程的init进程将清理它们（也就是说init进程将wait它们，从而去除它们的僵尸状态）。

28、如何避免僵尸进程?

- 通过signal(SIGCHLD, SIG_IGN)通知内核对子进程的结束不关心，由内核回收。如果不想让父进程挂起，可以在父进程中加入一条语句：signal(SIGCHLD,SIG_IGN);表示父进程忽略SIGCHLD信号，该信号是子进程退出的时候向父进程发送的。
- 父进程调用wait/waitpid等函数等待子进程结束，如果尚无子进程退出wait会导致父进程阻塞。waitpid可以通过传递WNOHANG使父进程不阻塞立即返回。
- 如果父进程很忙可以用signal注册信号处理函数，在信号处理函数调用wait/waitpid等待子进程退出。
- 通过两次调用fork。父进程首先调用fork创建一个子进程然后waitpid等待子进程退出，子进程再fork一个孙进程后退出。这样子进程退出后会被父进程等待回收，而对于孙子进程其父进程已经退出所以孙进程成为一个孤儿进程，孤儿进程由init进程接管，孙进程结束后，init会等待回收。

第一种方法忽略SIGCHLD信号，这常用于并发服务器的性能的一个技巧因为并发服务器常常fork很多子进程，子进程终结之后需要服务器进程去wait清理资源。如果将此信号的处理方式设为忽略，可让内核把僵尸子进程转交给init进程去处理，省去了大量僵尸进程占用系统资源。

29、常见内存分配内存错误

（1）内存分配未成功，却使用了它。

编程新手常犯这种错误，因为他们没有意识到内存分配会不成功。常用解决办法是，在使用内存之前检查指针是否为NULL。如果指针p是函数的参数，那么在函数的入口处用assert(p!=NULL)进行检查。如果是用malloc或new来申请内存，应该用if(p==NULL) 或if(p!=NULL)进行防错处理。

（2）内存分配虽然成功，但是尚未初始化就引用它。

犯这种错误主要有两个起因：一是没有初始化的观念；二是误以为内存的缺省初值全为零，导致引用初值错误（例如数组）。内存的缺省初值究竟是什么并没有统一的标准，尽管有些时候为零值，我们宁可信其无不可信其有。所以无论用何种方式创建数组，都别忘了赋初值，即便是赋零值也不可省略，不要嫌麻烦。

（3）内存分配成功并且已经初始化，但操作越过了内存的边界。

例如在使用数组时经常发生下标“多1”或者“少1”的操作。特别是在for循环语句中，循环次数很容易搞错，导致数组操作越界。

（4）忘记了释放内存，造成内存泄露。

含有这种错误的函数每被调用一次就丢失一块内存。刚开始时系统的内存充足，你看不到错误。终有一次程序突然挂掉，系统出现提示：内存耗尽。动态内存的申请与释放必须配对，程序中malloc与free的使用次数一定要相同，否则肯定有错误（new/delete同理）。

（5）释放了内存却继续使用它。常见于以下有三种情况：

- 程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存，此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。
- 函数的return语句写错了，注意不要返回指向“栈内存”的“指针”或者“引用”，因为该内存在函数体结束时被自动销毁。
- 使用free或delete释放了内存后，没有将指针设置为NULL。导致产生“野指针”。

30、内存交换中，被换出的进程保存在哪里?

保存在磁盘中，也就是外存中。具有对换功能的操作系统中，通常把磁盘空间分为文件区和对换区两部分。文件区主要用于存放文件，主要追求存储空间的利用率，因此对文件区空间的管理采用离散分配方式;对换区空间只占磁盘空间的小部分，被换出的进程数据就存放在对换区。由于对换的速度直接影响到系统的整体速度，因此对换区空间的管理主要追求换入换出速度，因此通常对换区采用连续分配方式(学过文件管理章节后即可理解)。总之，对换区的I/O速度比文件区的更快。

31、原子操作的是如何实现的

处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。首先处理器会自动保证基本的内存操作的原子性。处理器保证从系统内存中读取或者写入一个字节是原子的，意思是当一个处理器读取一个字节时，其他处理器不能访问这个字节的内存地址。Pentium 6和最新的处理器能自动保证单处理器对同一个缓存行里进行16/32/64位的操作是原子的，但是复杂的内存操作处理器是不能自动保证其原子性的，比如跨总线宽度、跨多个缓存行和跨页表的访问。但是，处理器提供总线锁定和缓存锁定两个机制来保证复杂内存操作的原子性。

（1）使用总线锁保证原子性 第一个机制是通过总线锁保证原子性。如果多个处理器同时对共享变量进行读改写操作（i++就是经典的读改写操作），那么共享变量就会被多个处理器同时进行操作，这样读改写操作就不是原子的，操作完之后共享变量的值会和期望的不一致。举个例子，如果i=1，我们进行两次i++操作，我们期望的结果是3，但是有可能结果是2，如图下图所示。

```
CPU1    CPU2
 i=1     i=1
 i+1     i+1
 i=2     i=2Copy to clipboardErrorCopied
```

原因可能是多个处理器同时从各自的缓存中读取变量i，分别进行加1操作，然后分别写入系统内存中。那么，想要保证读改写共享变量的操作是原子的，就必须保证CPU1读改写共享变量的时候，CPU2不能操作缓存了该共享变量内存地址的缓存。

处理器使用总线锁就是来解决这个问题的。**所谓总线锁就是使用处理器提供的一个LOCK＃信号，当一个处理器在总线上输出此信号时，其他处理器的请求将被阻塞住，那么该处理器可以独占共享内存。**

（2）使用缓存锁保证原子性 第二个机制是通过缓存锁定来保证原子性。在同一时刻，我们只需保证对某个内存地址的操作是原子性即可，但**总线锁定把CPU和内存之间的通信锁住了**，这使得锁定期间，其他处理器不能操作其他内存地址的数据，所以总线锁定的开销比较大，目前处理器在某些场合下使用缓存锁定代替总线锁定来进行优化。

频繁使用的内存会缓存在处理器的L1、L2和L3高速缓存里，那么原子操作就可以直接在处理器内部缓存中进行，并不需要声明总线锁，在Pentium 6和目前的处理器中可以使用“缓存锁定”的方式来实现复杂的原子性。

所谓“缓存锁定”是指内存区域如果被缓存在处理器的缓存行中，并且在Lock操作期间被锁定，那么当它执行锁操作回写到内存时，处理器不在总线上声言LOCK＃信号，而是修改内部的内存地址，并允许它的缓存一致性机制来保证操作的原子性，因为**缓存一致性机制会阻止同时修改由两个以上处理器缓存的内存区域数据，当其他处理器回写已被锁定的缓存行的数据时，会使缓存行无效，在如上图所示的例子中，当CPU1修改缓存行中的i时使用了缓存锁定，那么CPU2就不能使用同时缓存i的缓存行。**

但是有两种情况下处理器不会使用缓存锁定。 第一种情况是：当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行（cache line）时，则处理器会调用总线锁定。 第二种情况是：有些处理器不支持缓存锁定。对于Intel 486和Pentium处理器，就算锁定的内存区域在处理器的缓存行中也会调用总线锁定。

32、抖动你知道是什么吗？它也叫颠簸现象

刚刚换出的页面马上又要换入内存，刚刚换入的页面马上又要换出外存，这种频繁的页面调度行为称为抖动，或颠簸。产生抖动的主要原因是进程频繁访问的页面数目高于可用的物理块数(分配给进程的物理块不够)

为进程分配的物理块太少，会使进程发生抖动现象。为进程分配的物理块太多，又会降低系统整体的并发度，降低某些资源的利用率 为了研究为应该为每个进程分配多少个物理块，Denning 提出了进程工作集” 的概念

MySQL

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**Java面试题**」，即可获取最新版的 PDF 哦，扫码直达

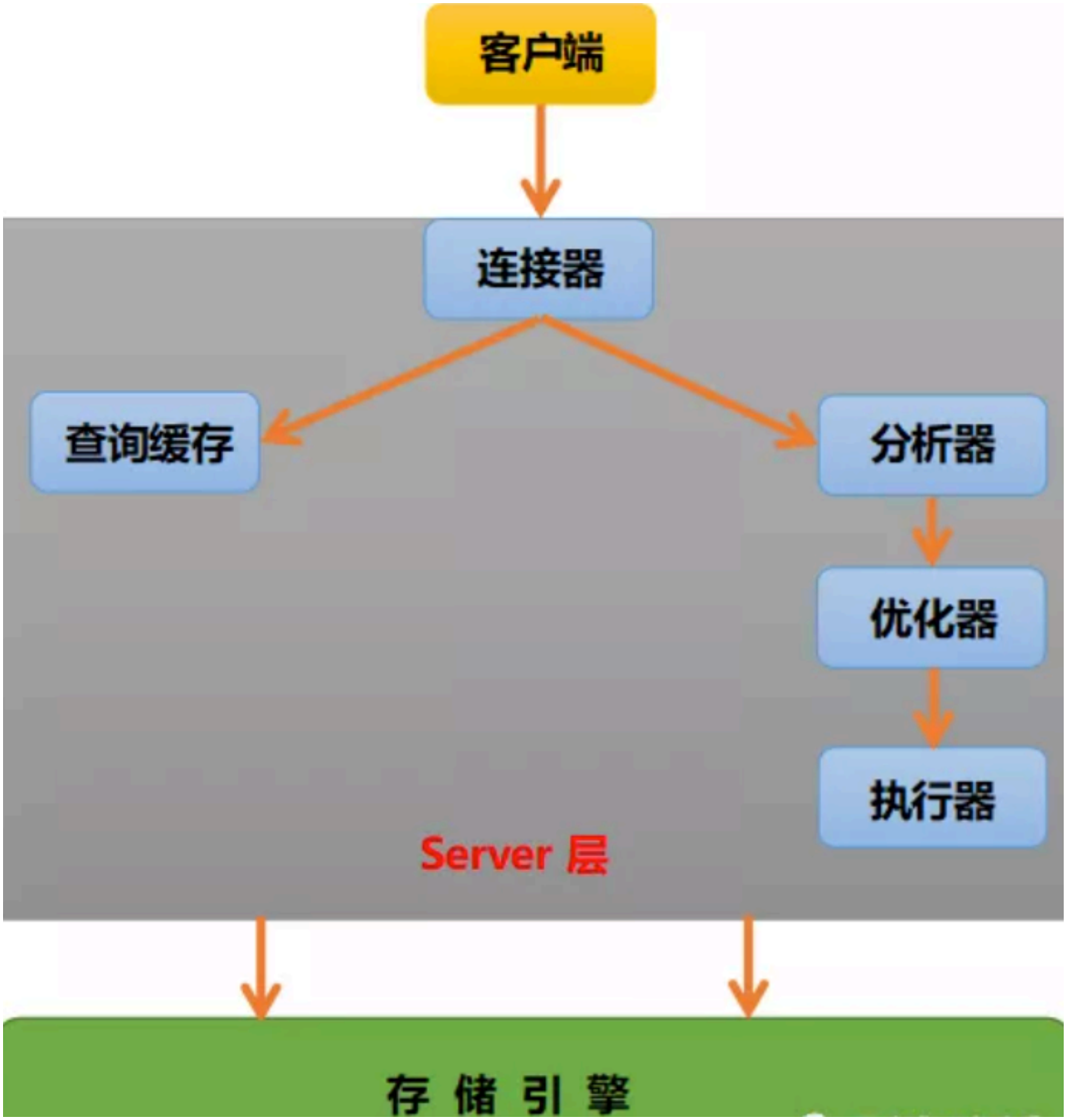
关注后，回复「**Java面试题**」，即可获取最新版 PDF。

另外，也建议大家来网站读，因为如果有错误，我会及时在网站更改，PDF 很难及时更新。

在线网站：https://www.iamshuaidi.com

1、请说下你对 MySQL 架构的了解?

- 先看下 MySQL 的基本架构图：



大体来说，MySQL 可以分为 Server 层和存储引擎两部分。

Server 层包括：连接器、查询缓存、分析器、优化器、执行器等，涵盖了 MySQL 的大多数核心服务功能，以及所有的内置函数（如：日期、时间、数学和加密函数等），所有跨存储引擎的功能都在这一层实现，比如：存储过程、触发器、视图等等。

存储引擎层负责：数据的存储和提取。其架构是插件式的，支持 InnoDB、MyISAM 等多个存储引擎。从 MySQL5.5.5 版本开始默认的是InnoDB，但是在建表时可以通过 engine = MyISAM 来指定存储引擎。不同存储引擎的表数据存取方式不同，支持的功能也不同。

从上图中可以看出，不同的存储引擎共用一个 Server 层，也就是从连接器到执行器的部分。

2、一条 SQL 语句在数据库框架中的执行流程?

1. 应用程序把查询 SQL 语句发送给服务器端执行；
2. 查询缓存，如果查询缓存是打开的，服务器在接收到查询请求后，并不会直接去数据库查询，而是在数据库的查询缓存中找是否有相对应的查询数据，如果存在，则直接返回给客户端。只有缓存不存在时，才会进行下面的操作；
3. 查询优化处理，生成执行计划。这个阶段主要包括解析 SQL、预处理、优化 SQL 执行计划；
4. MySQL 根据相应的执行计划完成整个查询；
5. 将查询结果返回给客户端。

详细过程可以看这篇博客https://blog.csdn.net/pcwl1206/article/details/86137408

3、数据库的三范式是什么?

1. 第一范式：强调的是列的原子性，即数据库表的每一列都是不可分割的原子数据项；
2. 第二范式：要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性；
3. 第三范式：任何非主属性不依赖于其它非主属性。

4、char 和 varchar 的区别?

char(n)： 固定长度类型，比如：订阅 char(10)，当你输入"abc"三个字符的时候，它们占的空间还是 10 个字节，其他 7 个是空字节。char 优点：效率高；缺点：占用空间；适用场景：存储密码的 md5 值，固定长度的，使用 char 非常合适。

varchar(n)： 可变长度，存储的值是每个值占用的字节再加上一个用来记录其长度的字节的长度。

所以，从空间上考虑 varcahr 比较合适；从效率上考虑 char 比较合适，二者使用需要权衡。

5、varchar(10) 和 varchar(20) 的区别?

varchar(10) 中 10 的涵义最多存放 10 个字符，varchar(10) 和 varchar(20) 存储 hello 所占空间一样，但后者在排序时会消耗更多内存，因为 order by col 采用 fixed_length 计算 col 长度。

6、谈谈你对索引的理解?

索引的出现是为了提高数据的查询效率，就像书的目录一样。一本500页的书，如果你想快速找到其中的某一个知识点，在不借助目录的情况下，那我估计你可得找一会儿。同样，对于数据库的表而言，索引其实就是它的“目录”。

同样索引也会带来很多负面影响：创建索引和维护索引需要耗费时间，这个时间随着数据量的增加而增加；索引需要占用物理空间，不光是表需要占用数据空间，每个索引也需要占用物理空间；当对表进行增、删、改、的时候索引也要动态维护，这样就降低了数据的维护速度。

建立索引的原则：

1. 在最频繁使用的、用以缩小查询范围的字段上建立索引；
2. 在频繁使用的、需要排序的字段上建立索引。

不适合建立索引的情况：

1. 对于查询中很少涉及的列或者重复值比较多的列，不宜建立索引；
2. 对于一些特殊的数据类型，不宜建立索引，比如：文本字段（text）等。

7、索引的底层使用的是什么数据结构?

索引的数据结构和具体存储引擎的实现有关,，在MySQL中使用较多的索引有 Hash 索引、B+树索引等。而我们经常使用的 InnoDB 存储引擎的默认索引实现为 B+ 树索引。

8、谈谈你对 B+ 树的理解?

1. B+ 树是基于 B 树和叶子节点顺序访问指针进行实现，它具有 B 树的平衡性，并且通过顺序访问指针来提高区间查询的性能。
2. 在 B+ 树中，一个节点中的 key 从左到右非递减排列，如果某个指针的左右相邻 key 分别是 key i 和 key i+1，且不为 null，则该指针指向节点的所有 key 大于等于 key i 且小于等于 key i+1。
3. 进行查找操作时，首先在根节点进行二分查找，找到一个 key 所在的指针，然后递归地在指针所指向的节点进行查找。直到查找到叶子节点，然后在叶子节点上进行二分查找，找出 key 所对应的 data。
4. 插入、删除操作会破坏平衡树的平衡性，因此在插入删除操作之后，需要对树进行一个分裂、合并、旋转等操作来维护平衡性。

9、为什么 InnoDB 存储引擎选用 B+ 树而不是 B 树呢?

用 B+ 树不用 B 树考虑的是 IO 对性能的影响，B 树的每个节点都存储数据，而 B+ 树只有叶子节点才存储数据，所以查找相同数据量的情况下，B 树的高度更高，IO 更频繁。数据库索引是存储在磁盘上的，当数据量大时，就不能把整个索引全部加载到内存了，只能逐一加载每一个磁盘页（对应索引树的节点）。

10、谈谈你对聚簇索引的理解?

聚簇索引是对磁盘上实际数据重新组织以按指定的一个或多个列的值排序的算法。特点是存储数据的顺序和索引顺序一致。一般情况下主键会默认创建聚簇索引，且一张表只允许存在一个聚簇索引。

聚簇索引和非聚簇索引的区别：

聚簇索引的叶子节点就是数据节点，而非聚簇索引的叶子节点仍然是索引节点，只不过有指向对应数据块的指针。

11、谈谈你对哈希索引的理解?

哈希索引能以 O(1) 时间进行查找，但是失去了有序性。无法用于排序与分组、只支持精确查找，无法用于部分查找和范围查找。

InnoDB 存储引擎有一个特殊的功能叫“自适应哈希索引”，当某个索引值被使用的非常频繁时，会在 B+ 树索引之上再创建一个哈希索引，这样就让 B+Tree 索引具有哈希索引的一些优点，比如：快速的哈希查找。

12、谈谈你对覆盖索引的认识?

如果一个索引包含了满足查询语句中字段与条件的数据就叫做覆盖索引。具有以下优点：

1. 索引通常远小于数据行的大小，只读取索引能大大减少数据访问量。
2. 一些存储引擎（例如：MyISAM）在内存中只缓存索引，而数据依赖于操作系统来缓存。因此，只访问索引可以不使用系统调用（通常比较费时）。
3. 对于 InnoDB 引擎，若辅助索引能够覆盖查询，则无需访问主索引。

13、索引的分类?

- **从数据结构角度**

1. 树索引 (O(log(n)))
2. Hash 索引

从物理存储角度

1. 聚集索引（clustered index）
2. 非聚集索引（non-clustered index）

从逻辑角度

1. 普通索引
2. 唯一索引
3. 主键索引
4. 联合索引
5. 全文索引

13、谈谈你对最左前缀原则的理解?

MySQL 使用联合索引时，需要满足最左前缀原则。下面举例对其进行说明：

```
1. 一个 2 列的索引 (name, age)，对 (name)、(name, age) 上建立了索引；
2. 一个 3 列的索引 (name, age, sex)，对 (name)、(name, age)、(name, age, sex) 上建立了索引。
```

1、 B+ 树的数据项是复合的数据结构，比如：(name, age, sex) 的时候，B+ 树是按照从左到右的顺序来建立搜索树的，比如：当(小明, 22, 男)这样的数据来检索的时候，B+ 树会优先比较 name 来确定下一步的所搜方向，如果 name 相同再依次比较 age 和 sex，最后得到检索的数据。但当 (22, 男) 这样没有 name 的数据来的时候，B+ 树就不知道第一步该查哪个节点，因为建立搜索树的时候 name 就是第一个比较因子，必须要先根据 name 来搜索才能知道下一步去哪里查询。

2、 当 (小明, 男) 这样的数据来检索时，B+ 树可以用 name 来指定搜索方向，但下一个字段 age 的缺失，所以只能把名字等于小明的数据都找到，然后再匹配性别是男的数据了， 这个是非常重要的性质，即索引的最左匹配特性。

关于最左前缀的补充：

1. 最左前缀匹配原则会一直向右匹配直到遇到范围查询（>、<、between、like）就停止匹配，比如：a = 1 and b = 2 and c > 3 and d = 4 如果建立 (a, b, c, d) 顺序的索引，d 是用不到索引的。如果建立 (a, b, d, c) 的索引则都可以用到，a、b、d 的顺序可以任意调整。
2. = 和 in 可以乱序，比如：a = 1 and b = 2 and c = 3 建立 (a, b ,c) 索引可以任意顺序，MySQL 的优化器会优化成索引可以识别的形式。

14、怎么知道创建的索引有没有被使用到？或者说怎么才可以知道这条语句运行很慢的原因？

使用 Explain 命令来查看语句的执行计划，MySQL 在执行某个语句之前，会将该语句过一遍查询优化器，之后会拿到对语句的分析，也就是执行计划，其中包含了许多信息。可以通过其中和索引有关的信息来分析是否命中了索引，例如：possilbe_key、key、key_len 等字段，分别说明了此语句可能会使用的索引、实际使用的索引以及使用的索引长度。

16、什么情况下索引会失效？即查询不走索引？

下面列举几种不走索引的 SQL 语句：

1、索引列参与表达式计算：

```
SELECT 'sname' FROM 'stu' WHERE 'age' + 10 = 30;
```

2、 函数运算：

```
SELECT 'sname' FROM 'stu' WHERE LEFT('date',4) < 1990;
```

3、%词语%--模糊查询：

```
SELECT * FROM 'manong' WHERE `uname` LIKE '%码农%' -- 走索引

SELECT * FROM 'manong' WHERE `uname` LIKE "%码农%" -- 不走索引
```

4、 字符串与数字比较不走索引：

```
CREATE TABLE 'a' ('a' char(10));
EXPLAIN SELECT * FROM 'a' WHERE 'a'="1" -- 走索引
EXPLAIN SELECT * FROM 'a'WHERE 'a'=1 -- 不走索引，同样也是使用了函数运算
```

5、 查询条件中有 or ，即使其中有条件带索引也不会使用。换言之，就是要求使用的所有字段，都必须建立索引：

```
select * from dept where dname='xxx' or loc='xx' or deptno = 45;
```

6、正则表达式不使用索引。

7、 MySQL 内部优化器会对 SQL 语句进行优化，如果优化器估计使用全表扫描要比使用索引快，则不使用索引。

16、查询性能的优化方法？

减少请求的数据量

1. 只返回必要的列：最好不要使用 SELECT * 语句。
2. 只返回必要的行：使用 LIMIT 语句来限制返回的数据。

3. 缓存重复查询的数据：使用缓存可以避免在数据库中进行查询，特别在要查询的数据经常被重复查询时，缓存带来的查询性能提升将会是非常明显的。

减少服务器端扫描的行数

1. 最有效的方式是使用索引来覆盖查询。

17、InnoDB 和 MyISAM 的比较?

1. 事务：MyISAM不支持事务，InnoDB支持事务；
2. 全文索引：MyISAM 支持全文索引，InnoDB 5.6 之前不支持全文索引；
3. 关于 count()：*MyISAM会直接存储总行数，InnoDB 则不会，需要按行扫描。意思就是对于 select count()* from table; 如果数据量大，MyISAM 会瞬间返回，而 InnoDB 则会一行行扫描；
4. 外键：MyISAM 不支持外键，InnoDB 支持外键；
5. 锁：MyISAM 只支持表锁，InnoDB 可以支持行锁。

18、谈谈你对水平切分和垂直切分的理解?

- **水平切分**

水平切分是将同一个表中的记录拆分到多个结构相同的表中。当一个表的数据不断增多时，水平切分是必然的选择，它可以将数据分布到集群的不同节点上，从而缓存单个数据库的压力。

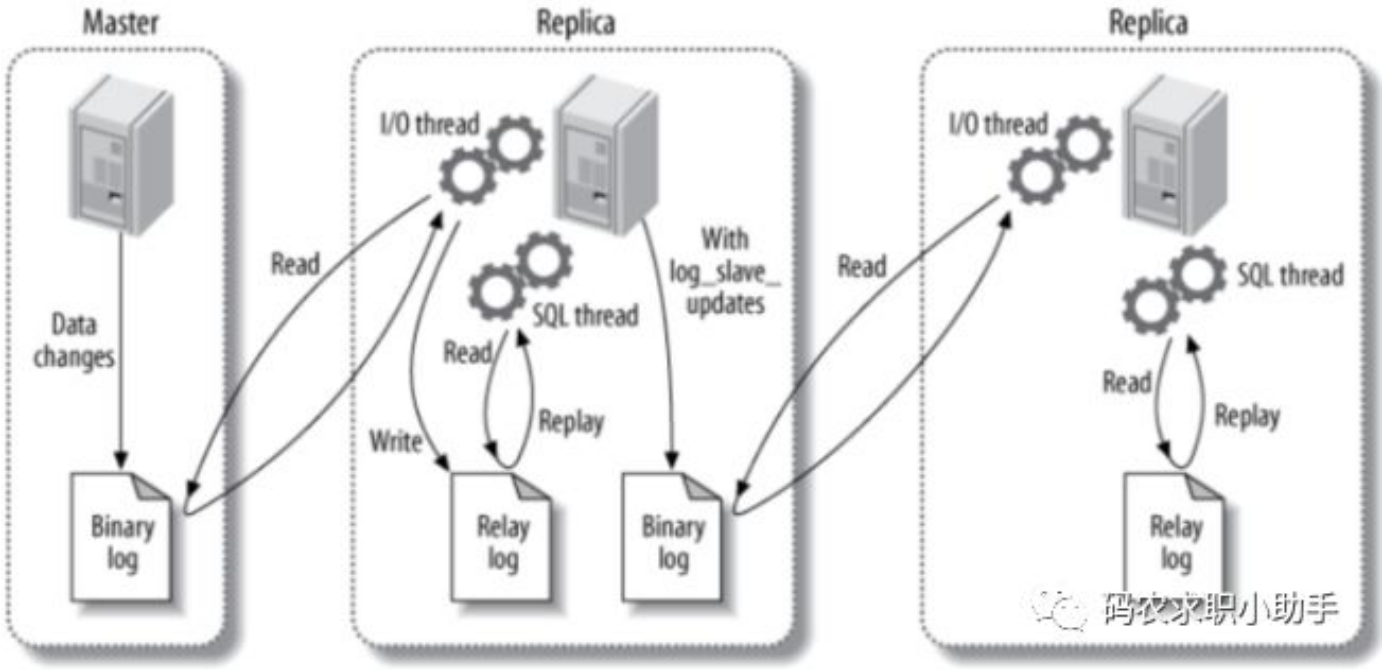
- **垂直切分**

垂直切分是将一张表按列切分成多个表，通常是按照列的关系密集程度进行切分，也可以利用垂直切分将经常被使用的列和不经常被使用的列切分到不同的表中。例如：将原来的电商数据库垂直切分成商品数据库、用户数据库等。

19、主从复制中涉及到哪三个线程?

主要涉及三个线程：binlog 线程、I/O 线程和 SQL 线程。

1. binlog 线程 ： 负责将主服务器上的数据更改写入二进制日志（Binary log）中。
2. I/O 线程 ： 负责从主服务器上读取二进制日志，并写入从服务器的重放日志（Relay log）中。
3. SQL 线程 ： 负责读取重放日志并重放其中的 SQL 语句。



20、主从同步的延迟原因及解决办法?

主从同步的延迟的原因：

假如一个服务器开放 N 个连接给客户端，这样有会有大并发的更新操作, 但是从服务器的里面读取 binlog 的线程仅有一个， 当某个 SQL 在从服务器上执行的时间稍长或者由于某个 SQL 要进行锁表就会导致主服务器的 SQL 大量积压，未被同步到从服务器里。这就导致了主从不一致， 也就是主从延迟。

主从同步延迟的解决办法：

实际上主从同步延迟根本没有什么一招制敌的办法， 因为所有的 SQL 必须都要在从服务器里面执行一遍，但是主服务器如果不断的有更新操作源源不断的写入，那么一旦有延迟产生，那么延迟加重的可能性就会原来越大。当然我们可以做一些缓解的措施。

1. 我们知道因为主服务器要负责更新操作， 它对安全性的要求比从服务器高，所有有些设置可以修改，比如 sync_binlog=1，innodb_flush_log_at_trx_commit = 1 之类的设置，而 slave 则不需要这么高的数据安全，完全可以将 sync_binlog 设置为 0 或者关闭 binlog、innodb_flushlog、innodb_flush_log_at_trx_commit 也可以设置为 0 来提高 SQL 的执行效率。
2. 增加从服务器，这个目的还是分散读的压力， 从而降低服务器负载。

21、谈谈你对数据库读写分离的理解?

读写分离常用代理方式来实现，代理服务器接收应用层传来的读写请求，然后决定转发到哪个服务器。主服务器处理写操作以及实时性要求比较高的读操作，而从服务器处理读操作。

读写分离能提高性能的原因在于：

1. 主从服务器负责各自的读和写，极大程度缓解了锁的争用；
2. 从服务器可以使用 MyISAM，提升查询性能以及节约系统开销；
3. 增加冗余，提高可用性。

22、请你描述下事务的特性?

1. 原子性：事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；
2. 一致性：执行事务前后，数据库从一个一致性状态转换到另一个一致性状态。
3. 隔离性：并发访问数据库时，一个用户的事物不被其他事务所干扰，各并发事务之间数据库是独立的；
4. 持久性：一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库 发生故障也不应该对其有任何影响。

23、谈谈你对事务隔离级别的理解?

1. READ_UNCOMMITTED（未提交读）: 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读；
2. READ_COMMITTED（提交读）: 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生；
3. REPEATABLE_READ（可重复读）: 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生；
4. SERIALIZABLE（串行化）: 最高的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

24、解释下什么叫脏读、不可重复读和幻读?

- **脏读:**

表示一个事务能够读取另一个事务中还未提交的数据。比如：某个事务尝试插入记录 A，此时该事务还未提交，然后另一个事务尝试读取到了记录 A。

- **不可重复读：**

是指在一个事务内，多次读同一数据。

- **幻读:**

指同一个事务内多次查询返回的结果集不一样。比如同一个事务 A 第一次查询时候有 n 条记录，但是第二次同等条件下查询却有 n+1 条记录，这就好像产生了幻觉。发生幻读的原因也是另外一个事务新增或者删除或者修改了第一个事务结果集里面的数据，同一个记录的数据内容被修改了，所有数据行的记录就变多或者变少了。

25、MySQL 默认的隔离级别是什么?

MySQL默认采用的 REPEATABLE_READ隔离级别。

Oracle 默认采用的 READ_COMMITTED 隔离级别。

26、谈谈你对MVCC 的了解?

数据库并发场景:

1. 读-读：不存在任何问题，也不需要并发控制；
2. 读-写：有线程安全问题，可能会造成事务隔离性问题，可能遇到脏读，幻读，不可重复读；
3. 写-写：有线程安全问题，可能会存在更新丢失问题。

多版本并发控制（MVCC）是一种用来解决读-写冲突的无锁并发控制，也就是为事务分配单向增长的时间戳，为每个修改保存一个版本，版本与事务时间戳关联，读操作只读该事务开始前的数据库的快照。

MVCC 可以为数据库解决以下问题：

1. 在并发读写数据库时，可以做到在读操作时不用阻塞写操作，写操作也不用阻塞读操作，提高了数据库并发读写的性能；
2. 同时还可以解决脏读，幻读，不可重复读等事务隔离问题，但不能解决更新丢失问题。

27、说一下 MySQL 的行锁和表锁?

MyISAM 只支持表锁，InnoDB 支持表锁和行锁，默认为行锁。

表级锁：开销小，加锁快，不会出现死锁。锁定粒度大，发生锁冲突的概率最高，并发量最低。

行级锁：开销大，加锁慢，会出现死锁。锁力度小，发生锁冲突的概率小，并发度最高。

28、InnoDB 存储引擎的锁的算法有哪些?

1. Record lock：单个行记录上的锁；
2. Gap lock：间隙锁，锁定一个范围，不包括记录本身；
3. Next-key lock：record+gap 锁定一个范围，包含记录本身。

29、MySQL 问题排查都有哪些手段?

1. 使用 show processlist 命令查看当前所有连接信息；
2. 使用 Explain 命令查询 SQL 语句执行计划；
3. 开启慢查询日志，查看慢查询的 SQL。

30、MySQL 数据库 CPU 飙升到 500% 的话他怎么处理?

1. 列出所有进程 show processlist，观察所有进程，多秒没有状态变化的(干掉)；
2. 查看超时日志或者错误日志 (一般会是查询以及大批量的插入会导致 CPU与 I/O 上涨，当然不排除网络状态突然断了，导致一个请求服务器只接受到一半。

Redis

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**Java面试题**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**Java面试题**」，即可获取最新版 PDF。

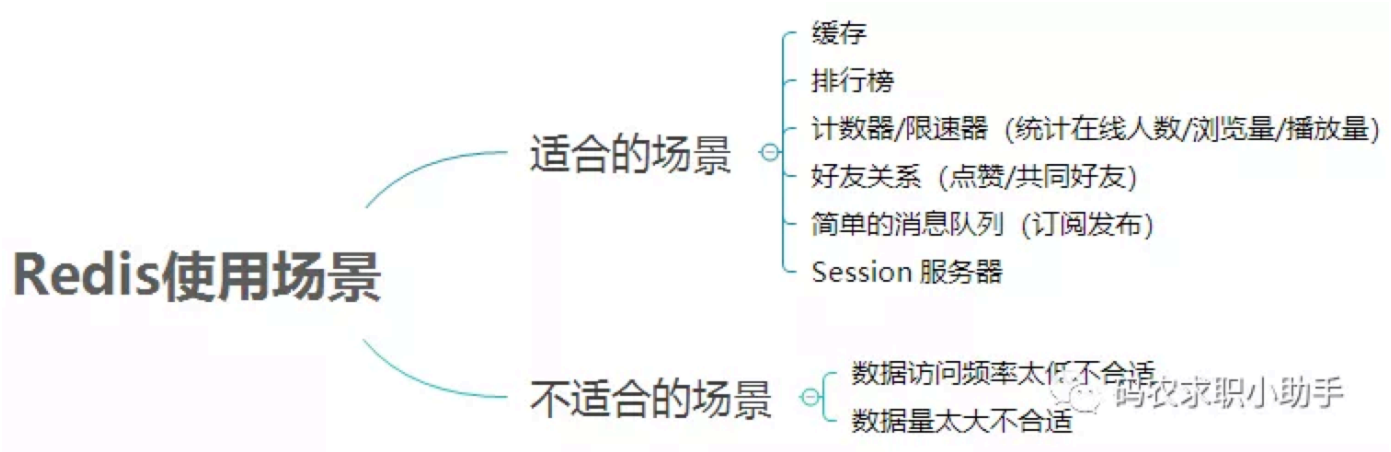
另外，也建议大家来网站读，因为如果有错误，我会及时在网站更改，PDF 很难及时更新。

在线网站：https://www.iamshuaidi.com

1、谈下你对 Redis 的了解?

Redis（全称：Remote Dictionary Server 远程字典服务）是一个开源的使用 ANSI C 语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value 数据库，并提供多种语言的 API。

2、Redis 一般都有哪些使用场景?



Redis 适合的场景

1. 缓存：减轻 MySQL 的查询压力，提升系统性能；
2. 排行榜：利用 Redis 的 SortSet（有序集合）实现；
3. 计算器/限速器：利用 Redis 中原子性的自增操作，我们可以统计类似用户点赞数、用户访问数等。这类操作如果用 MySQL，频繁的读写会带来相当大的压力；限速器比较典型的使用场景是限制某个用户访问某个 API 的频率，常用的有抢购时，防止用户疯狂点击带来不必要的压力；
4. 好友关系：利用集合的一些命令，比如求交集、并集、差集等。可以方便解决一些共同好友、共同爱好之类的功能；
5. 消息队列：除了 Redis 自身的发布/订阅模式，我们也可以利用 List 来实现一个队列机制，比如：到货通知、

邮件发送之类的需求，不需要高可靠，但是会带来非常大的 DB 压力，完全可以用 List 来完成异步解耦；
6. Session 共享：Session 是保存在服务器的文件中，如果是集群服务，同一个用户过来可能落在不同机器上，这就会导致用户频繁登陆；采用 Redis 保存 Session 后，无论用户落在那台机器上都能够获取到对应的 Session 信息。

Redis 不适合的场景

数据量太大、数据访问频率非常低的业务都不适合使用 Redis，数据太大会增加成本，访问频率太低，保存在内存中纯属浪费资源。

3、Redis 有哪些常见的功能?

1. 数据缓存功能
2. 分布式锁的功能
3. 支持数据持久化
4. 支持事务
5. 支持消息队列

4、Redis 支持的数据类型有哪些?

- **1. string 字符串**

字符串类型是 Redis 最基础的数据结构，首先键是字符串类型，而且其他几种结构都是在字符串类型基础上构建的。字符串类型实际上可以是字符串：简单的字符串、XML、JSON；数字：整数、浮点数；二进制：图片、音频、视频。

使用场景：缓存、计数器、共享 Session、限速。

- **2. Hash（哈希）**

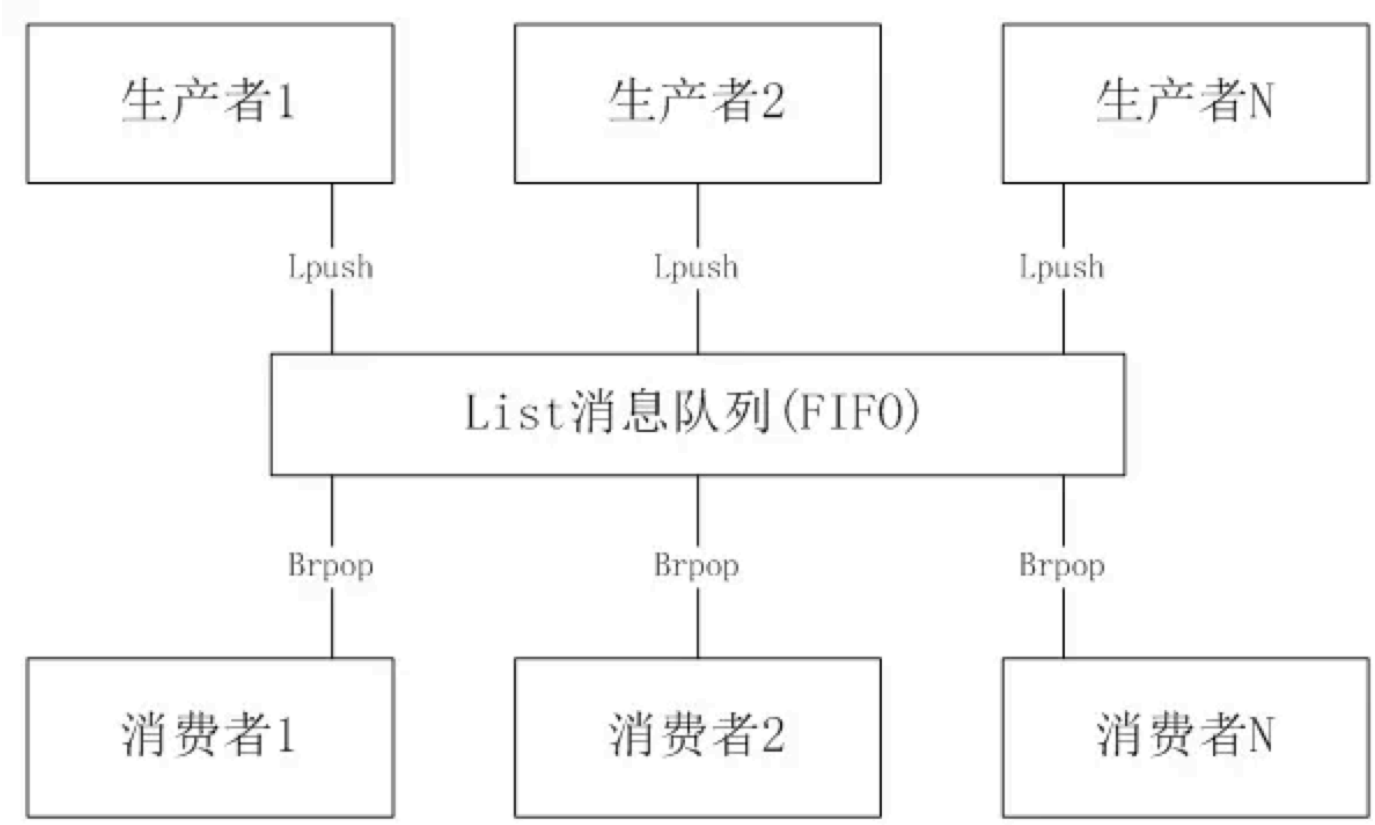
在 Redis中哈希类型是指键本身是一种键值对结构，如 value={{field1,value1},.....{fieldN,valueN}}

使用场景：哈希结构相对于字符串序列化缓存信息更加直观，并且在更新操作上更加便捷。所以常常用于用户信息等管理，但是哈希类型和关系型数据库有所不同，哈希类型是稀疏的，而关系型数据库是完全结构化的，关系型数据库可以做复杂的关系查询，而 Redis 去模拟关系型复杂查询开发困难且维护成本高。

- **3. List（列表）**

列表类型是用来储存多个有序的字符串，列表中的每个字符串成为元素，一个列表最多可以储存 2 ^ 32 - 1 个元素，在 Redis 中，可以队列表两端插入和弹出，还可以获取指定范围的元素列表、获取指定索引下的元素等，列表是一种比较灵活的数据结构，它可以充当栈和队列的角色。

使用场景：Redis 的 lpush + brpop 命令组合即可实现阻塞队列，生产者客户端是用 lpush 从列表左侧插入元素，多个消费者客户端使用 brpop 命令阻塞式的“抢”列表尾部的元素，多个客户端保证了消费的负载均衡和高可用性。



- **4. Set（集合）**

集合类型也是用来保存多个字符串的元素，但和列表不同的是集合中不允许有重复的元素，并且集合中的元素是无序的，不能通过索引下标获取元素，Redis 除了支持集合内的增删改查，同时还支持多个集合取交集、并集、差集。合理的使用好集合类型，能在实际开发中解决很多实际问题。

使用场景：如：一个用户对娱乐、体育比较感兴趣，另一个可能对新闻感兴趣，这些兴趣就是标签，有了这些数据就可以得到同一标签的人，以及用户的共同爱好的标签，这些数据对于用户体验以及曾强用户粘度比较重要。

- **5. zset（sorted set：有序集合）**

有序集合和集合有着必然的联系，它保留了集合不能有重复成员的特性，但不同得是，有序集合中的元素是可以排序的，但是它和列表的使用索引下标作为排序依据不同的是：它给每个元素设置一个分数，作为排序的依据。

使用场景：排行榜是有序集合经典的使用场景。例如：视频网站需要对用户上传的文件做排行榜，榜单维护可能是多方面：按照时间、按照播放量、按照获得的赞数等。

5、Redis 为什么这么快?

1. 完全基于内存，绝大部分请求是纯粹的内存操作，非常快速；
2. 数据结构简单，对数据操作也简单；
3. 采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；
4. 使用多路 I/O 复用模型，非阻塞 IO。

6、什么是缓存穿透？怎么解决？

缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时需要从数据库查询，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，造成缓存穿透。

解决办法：

1、缓存空对象：如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

缓存空对象带来的问题：

1. 空值做了缓存，意味着缓存中存了更多的键，需要更多的内存空间，比较有效的方法是针对这类数据设置一个较短的过期时间，让其自动剔除。
2. 缓存和存储的数据会有一段时间窗口的不一致，可能会对业务有一定影响。例如：过期时间设置为 5分钟，如果此时存储添加了这个数据，那此段时间就会出现缓存和存储数据的不一致，此时可以利用消息系统或者其他方式清除掉缓存层中的空对象。

2、布隆过滤器：将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力。

7、什么是缓存雪崩？该如何解决？

如果缓存集中在一段时间内失效，发生大量的缓存穿透，所有的查询都落在数据库上，造成了缓存雪崩。

解决办法：

1. 加锁排队：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个 key 只允许一个线程查询数据和写缓存，其他线程等待；
2. 数据预热：可以通过缓存 reload 机制，预先去更新缓存，再即将发生大并发访问前手动触发加载缓存不同的 key，设置不同的过期时间，让缓存失效的时间点尽量均匀；
3. 做二级缓存，或者双缓存策略：Cache1 为原始缓存，Cache2 为拷贝缓存，Cache1 失效时，可以访问 Cache2，Cache1 缓存失效时间设置为短期，Cache2 设置为长期。
4. 在缓存的时候给过期时间加上一个随机值，这样就会大幅度的减少缓存在同一时间过期。

8、 怎么保证缓存和数据库数据的一致性？

1. 从理论上说，只要我们设置了合理的键的过期时间，我们就能保证缓存和数据库的数据最终是一致的。因为只要缓存数据过期了，就会被删除。随后读的时候，因为缓存里没有，就可以查数据库的数据，然后将数据库查出来的数据写入到缓存中。除了设置过期时间，我们还需要做更多的措施来尽量避免数据库与缓存处于不一致的情况发生。
2. 新增、更改、删除数据库操作时同步更新 Redis，可以使用事物机制来保证数据的一致性。

一般有如下四种方案，详情看这里：

1. 先更新数据库，后更新缓存
2. 先更新缓存，后更新数据库
3. 先删除缓存，后更新数据库
4. 先更新数据库，后删除缓存

第一种和第二种方案，没有人使用的，因为第一种方案存在问题是：并发更新数据库场景下，会将脏数据刷到缓存。

第二种方案存在的问题是：如果先更新缓存成功，但是数据库更新失败，则肯定会造成数据不一致。

目前主要用第三和第四种方案，详情看这里：

双写一致性方案一：先删除缓存，后更新数据库

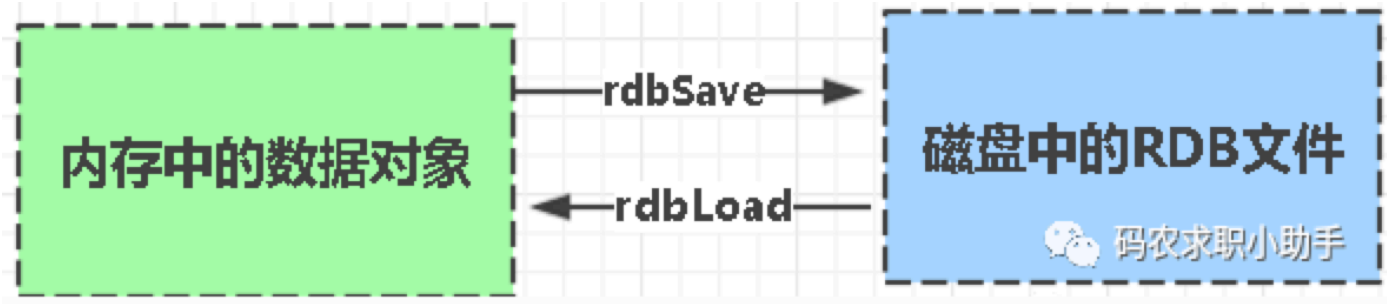
双写一致性方案二：先更新数据库，后删除缓存

9、Redis 持久化有几种方式?

持久化就是把内存的数据写到磁盘中去，防止服务宕机了内存数据丢失。Redis 提供了两种持久化方式：RDB（默认） 和 AOF。

RDB

RDB 是 Redis DataBase 的缩写。按照一定的时间周期策略把内存的数据以快照的形式保存到硬盘的二进制文件。即 Snapshot 快照存储，对应产生的数据文件为 dump.rdb，通过配置文件中的 save 参数来定义快照的周期。核心函数：rdbSave（生成 RDB 文件）和 rdbLoad（从文件加载内存）两个函数。



AOF

AOF 是 Append-only file 的缩写。Redis会将每一个收到的写命令都通过 Write 函数追加到文件最后，类似于 MySQL 的 binlog。当 Redis 重启是会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。每当执行服务器（定时）任务或者函数时，flushAppendOnlyFile 函数都会被调用， 这个函数执行以下两个工作：

- WRITE：根据条件，将 aof_buf 中的缓存写入到 AOF 文件；
- SAVE：根据条件，调用 fsync 或 fdatasync 函数，将 AOF 文件保存到磁盘中。

RDB 和 AOF 的区别：

1. AOF 文件比 RDB 更新频率高，优先使用 AOF 还原数据；
2. AOF比 RDB 更安全也更大；
3. RDB 性能比 AOF 好；

4. 如果两个都配了优先加载 AOF。

10、Redis 怎么实现分布式锁?

Redis 为单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对 Redis 的连接并不存在竞争关系。Redis 中可以使用 SETNX 命令实现分布式锁。一般使用 setnx(set if not exists) 指令，只允许被一个程序占有，使用完调用 del 释放锁。

11、Redis 内存淘汰策略有哪些?

1. volatile-lru：从已设置过期时间的数据集（server. db[i]. expires）中挑选最近最少使用的数据淘汰；
2. volatile-ttl：从已设置过期时间的数据集（server. db[i]. expires）中挑选将要过期的数据淘汰。
3. volatile-random：从已设置过期时间的数据集（server. db[i]. expires）中任意选择数据淘汰。
4. allkeys-lru：从数据集（server. db[i]. dict）中挑选最近最少使用的数据淘汰。
5. allkeys-random：从数据集（server. db[i]. dict）中任意选择数据淘汰。
6. no-enviction（驱逐）：禁止驱逐数据。

12、Redis 常见性能问题和解决方案?

1. Master 最好不要做任何持久化工作，如 RDB 内存快照和 AOF 日志文件。如果数据比较重要，某个 Slave 开启 AOF 备份数据，策略设置为每秒同步一次；
2. 为了主从复制的速度和连接的稳定性， Master 和 Slave 最好在同一个局域网内；
3. 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3...

13、Redis的过期键的删除策略

我们都知道，Redis是key-value数据库，我们可以设置Redis中缓存的key的过期时间。Redis的过期策略就是指当Redis中缓存的key过期了，Redis如何处理。

过期策略通常有以下三种：

- 定时过期：每个设置过期时间的key都需要创建一个定时器，到过期时间就会立即清除。该策略可以立即清除过期的数据，对内存很友好；但是会占用大量的CPU资源去处理过期的数据，从而影响缓存的响应时间和吞吐量。
- 惰性过期：只有当访问一个key时，才会判断该key是否已过期，过期则清除。该策略可以最大化地节省CPU资源，却对内存非常不友好。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。
- 定期过期：每隔一定的时间，会扫描一定数量的数据库的expires字典中一定数量的key，并清除其中已过期的key。该策略是前两者的一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。(expires字典会保存所有设置了过期时间的key的过期时间数据，其中，key是指向键空间中的某个键的指针，value是该键的毫秒精度的UNIX时间戳表示的过期时间。键空间是指该Redis集群中保存的所有键。)

Redis中同时使用了惰性过期和定期过期两种过期策略。

14、我们知道通过expire来设置key 的过期时间，那么对过期的数据怎么处理呢?

除了缓存服务器自带的缓存失效策略之外（Redis默认的有6中策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

1. 定时去清理过期的缓存；
2. 当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的key是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，大家可以根据自己的应用场景来权衡。

15、Hash 冲突怎么办?

Redis 通过**链式哈希**解决冲突：**也就是同一个 桶里面的元素使用链表保存**。但是当链表过长就会导致查找性能变差可能，所以 Redis 为了追求快，使用了两个全局哈希表。用于 rehash 操作，增加现有的哈希桶数量，减少哈希冲突。

开始默认使用 「hash 表 1 」保存键值对数据，「hash 表 2」 此刻没有分配空间。当数据越来越多触发 rehash 操作，则执行以下操作：

1. 给 「hash 表 2 」分配更大的空间；
2. 将 「hash 表 1 」的数据重新映射拷贝到 「hash 表 2」 中；
3. 释放 「hash 表 1」 的空间。

值得注意的是，将 hash 表 1 的数据重新映射到 hash 表 2 的过程中并不是一次性的，这样会造成 Redis 阻塞，无法提供服务。

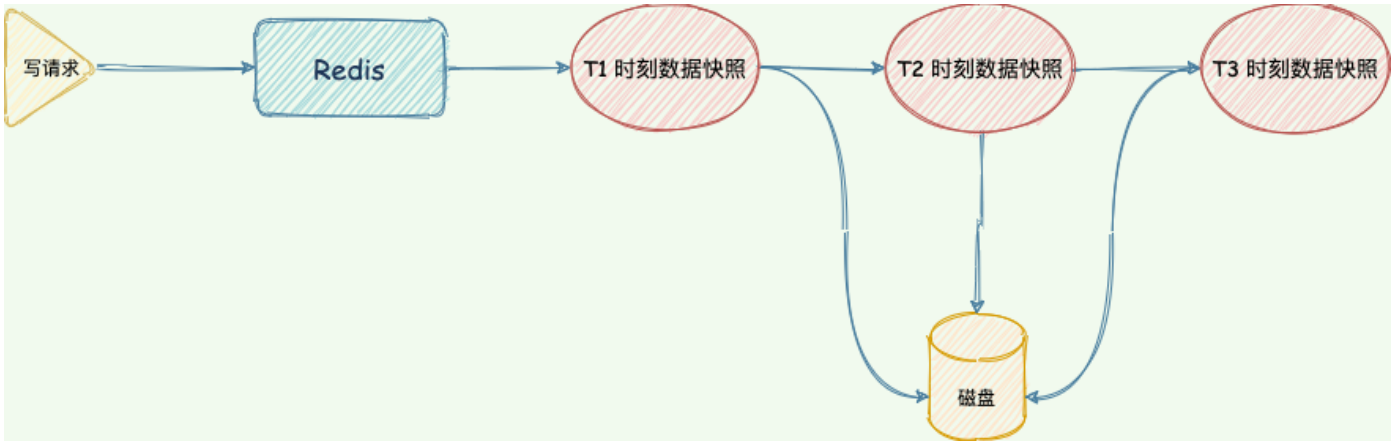
而是采用了**渐进式 rehash**，每次处理客户端请求的时候，先从「 hash 表 1」 中第一个索引开始，将这个位置的所有数据拷贝到 「hash 表 2」 中，就这样将 rehash 分散到多次请求过程中，避免耗时阻塞。

16、什么是 RDB 内存快照?

在 Redis 执行「写」指令过程中，内存数据会一直变化。所谓的内存快照，指的就是 Redis 内存中的数据在某一刻的状态数据。

好比时间定格在某一刻，当我们拍照的，通过照片就能把某一刻的瞬间画面完全记录下来。

Redis 跟这个类似，就是把某一刻的数据以文件的形式拍下来，写到磁盘上。这个快照文件叫做 **RDB 文件，RDB 就是 Redis DataBase 的缩写。**



在做数据恢复时，直接将 RDB 文件读入内存完成恢复。

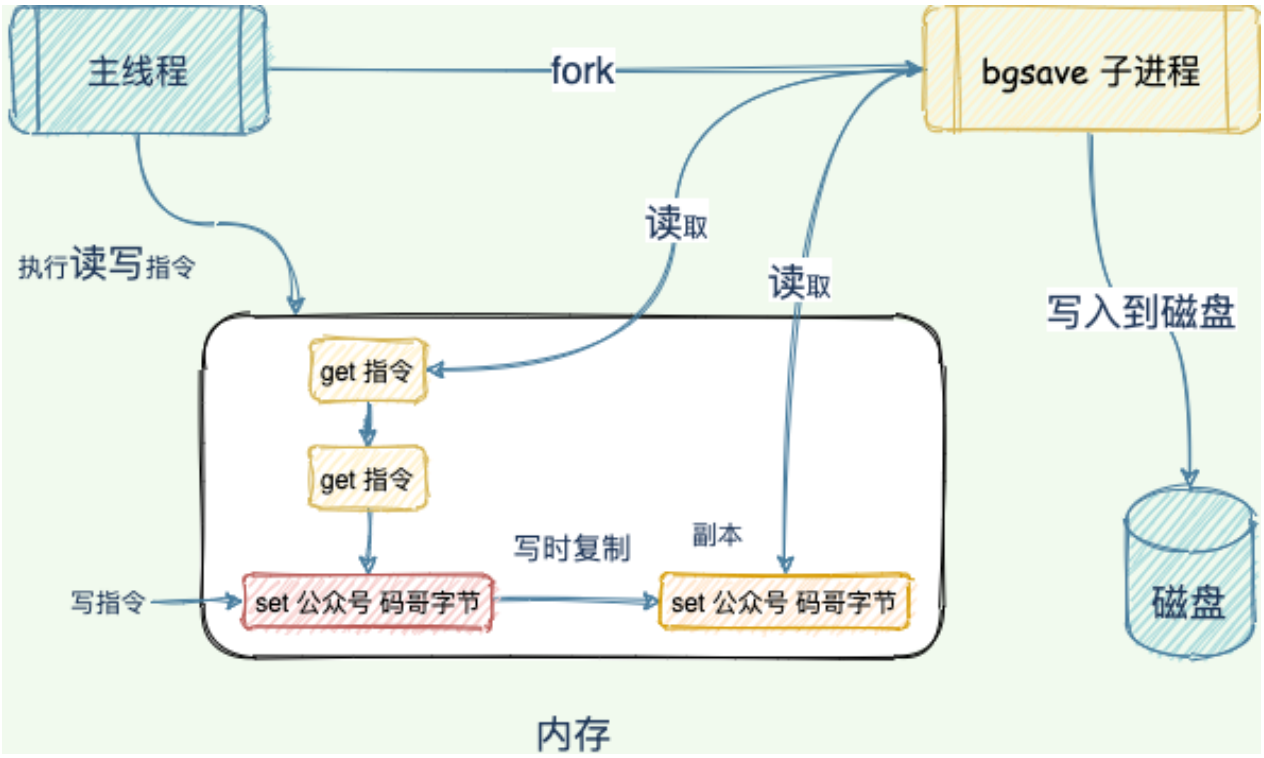
17、在生成 RDB 期间，Redis 可以同时处理写请求么?

可以的，Redis 使用操作系统的多进程**写时复制技术 COW(Copy On Write)** 来实现快照持久化，保证数据一致性。

Redis 在持久化时会调用 glibc 的函数`fork`产生一个子进程，快照持久化完全交给子进程来处理，父进程继续处理客户端请求。

当主线程执行写指令修改数据的时候，这个数据就会复制一份副本，`bgsave`子进程读取这个副本数据写到 RDB 文件。

这既保证了快照的完整性，也允许主线程同时对数据进行修改，避免了对正常业务的影响。



18、如何实现数据尽可能少丢失又能兼顾性能呢?

重启 Redis 时，我们很少使用 rdb 来恢复内存状态，因为会丢失大量数据。我们通常使用 AOF 日志重放，但是重放 AOF 日志性能相对 rdb 来说要慢很多，这样在 Redis 实例很大的情况下，启动需要花费很长的时间。

Redis 4.0 为了解决这个问题，带来了一个新的持久化选项——**混合持久化**。将 rdb 文件的内容和增量的 AOF 日志文件存在一起。这里的 AOF 日志不再是全量的日志，而是**自持久化开始到持久化结束的这段时间发生的增量 AOF 日志**，通常这部分 AOF 日志很小。

于是**在 Redis 重启的时候，可以先加载 rdb 的内容，然后再重放增量 AOF 日志就可以完全替代之前的 AOF 全量文件重放，重启效率因此大幅得到提升**。

19、Redis如何做内存优化?

1、**控制key的数量**：当使用Redis存储大量数据时，通常会存在大量键，过多的键同样会消耗大量内存。Redis本质是一个数据结构服务器，它为我们提供多种数据结构，如hash，list，set，zset 等结构。使用Redis时不要进入一个误区，大量使用get/set这样的API，把Redis当成Memcached使用。对于存储相同的数据内容利用Redis的数据结构降低外层键的数量，也可以节省大量内存。

2、**缩减键值对象**，降低Redis内存使用最直接的方式就是缩减键（key）和值（value）的长度。

- key长度：如在设计键时，在完整描述业务情况下，键值越短越好。
- value长度：值对象缩减比较复杂，常见需求是把业务对象序列化成二进制数组放入Redis。首先应该在业务上精简业务对象，去掉不必要的属性避免存储无效数据。其次在序列化工具选择上，应该选择更高效的序列化工具来降低字节数组大小。

3、**编码优化**。Redis对外提供了string,list,hash,set,zet等类型，但是Redis内部针对不同类型存在编码的概念，所谓编码就是具体使用哪种底层数据结构来实现。编码不同将直接影响数据的内存占用和读写效率。可参考文章：https://cloud.tencent.com/developer/article/1162213

20、Redis线程模型

Redis的线程模型包括Redis 6.0之前和Redis 6.0。

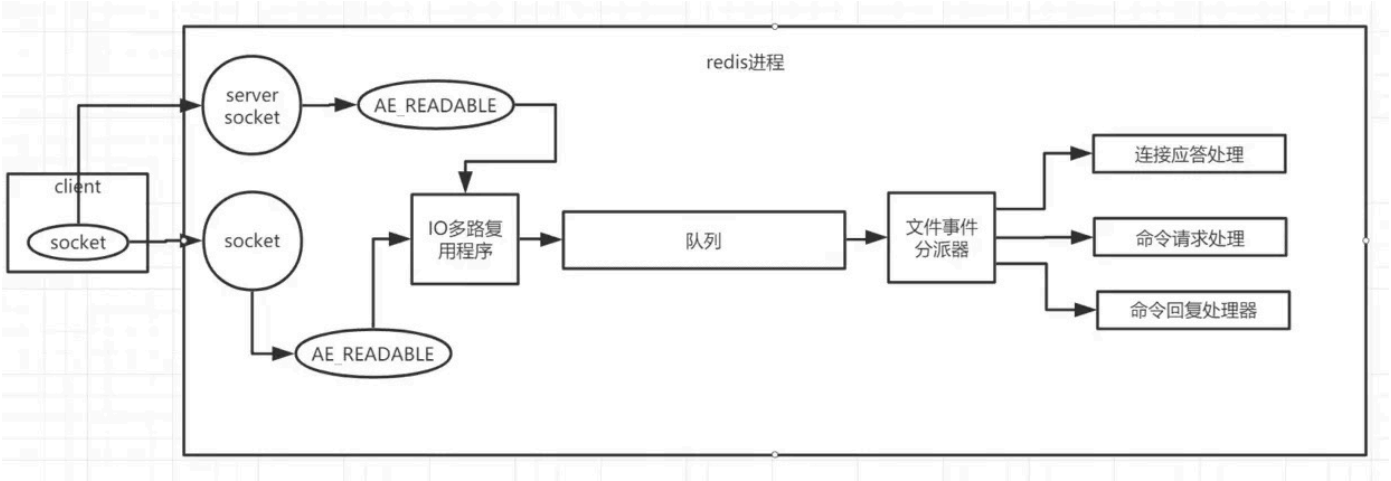
下面介绍的是Redis 6.0之前。

Redis 是基于 reactor 模式开发了网络事件处理器，这个处理器叫做文件事件处理器（file event handler）。由于这个文件事件处理器是单线程的，所以 Redis 才叫做单线程的模型。采用 IO 多路复用机制同时监听多个 Socket，根据 socket 上的事件来选择对应的事件处理器来处理这个事件。

> IO多路复用是 IO 模型的一种，有时也称为异步阻塞 IO，是基于经典的 Reactor 设计模式设计的。多路指的是多个 Socket 连接，复用指的是复用一个线程。多路复用主要有三种技术：Select，Poll，Epoll。

> Epoll 是最新的也是目前最好的多路复用技术。

模型如下图：



文件事件处理器的结构包含了四个部分：

1、多个 Socket。Socket 会产生 AE_READABLE 和 AE_WRITABLE 事件：

- 当 socket 变得可读时或者有新的可以应答的 socket 出现时，socket 就会产生一个 AE_READABLE 事件
- 当 socket 变得可写时，socket 就会产生一个 AE_WRITABLE 事件。

2、IO 多路复用程序

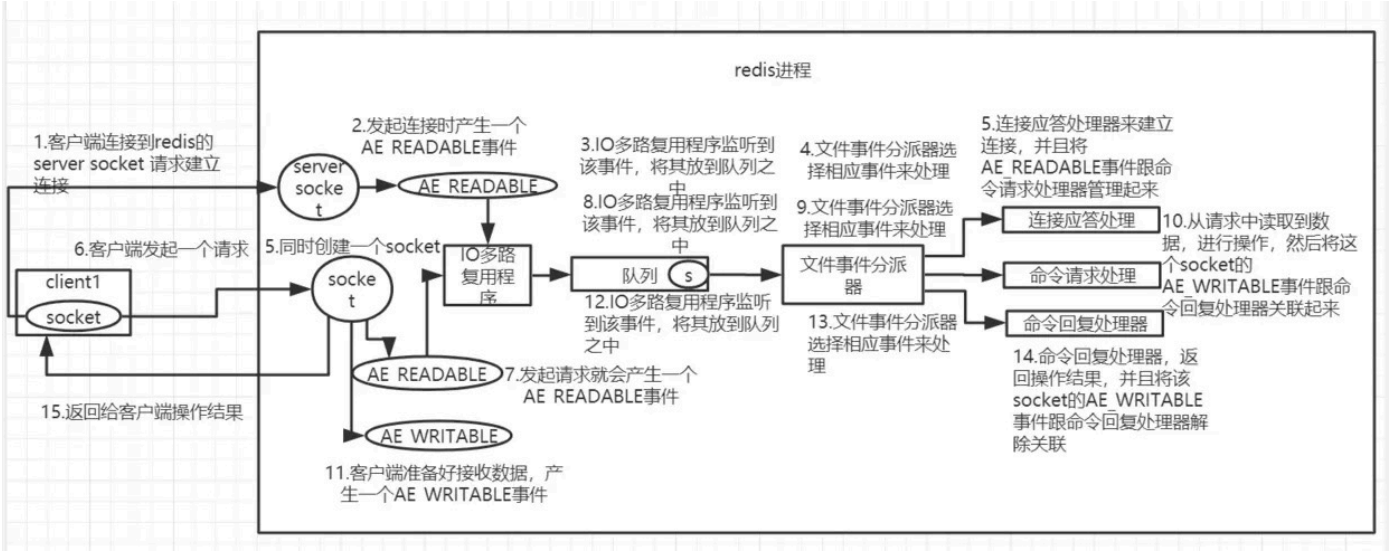
3、文件事件分派器

4、事件处理器。事件处理器包括：连接应答处理器、命令请求处理器、命令回复处理器，每个处理器对应不同的 socket 事件：

- 如果是客户端要连接 Redis，那么会为 socket 关联连接应答处理器
- 如果是客户端要写数据到 Redis（读、写请求命令），那么会为 socket 关联命令请求处理器
- 如果是客户端要从 Redis 读数据，那么会为 socket 关联命令回复处理器

多个 socket 会产生不同的事件，不同的事件对应着不同的操作，IO 多路复用程序监听着这些 Socket，当这些 Socket 产生了事件，IO 多路复用程序会将这些事件放到一个队列中，通过这个队列，以有序、同步、每次一个事件的方式向文件时间分派器中传送。当事件处理器处理完一个事件后，IO 多路复用程序才会继续向文件分派器传送下一个事件。

下图是客户端与 Redis 通信的一次完整的流程：



1. Redis 启动初始化的时候，Redis 会将连接应答处理器与 AE_READABLE 事件关联起来。
2. 如果一个客户端跟 Redis 发起连接，此时 Redis 会产生一个 AE_READABLE 事件，由于开始之初 AE_READABLE 是与连接应答处理器关联，所以由连接应答处理器来处理该事件，这时连接应答处理器会与客户端建立连接，创建客户端响应的 socket，同时将这个 socket 的 AE_READABLE 事件与命令请求处理器关联起来。
3. 如果这个时间客户端向 Redis 发送一个命令（set k1 v1），这时 socket 会产生一个 AE_READABLE 事件，IO 多路复用程序会将该事件压入队列中，此时事件分派器从队列中取得该事件，由于该 socket 的 AE_READABLE 事件已经和命令请求处理器关联了，因此事件分派器会将该事件交给命令请求处理器处理，命令请求处理器读取事件中的命令并完成。操作完成后，Redis 会将该 socket 的 AE_WRITABLE 事件与命令回复处理器关联。
4. 如果客户端已经准备好接受数据后，Redis 中的该 socket 会产生一个 AE_WRITABLE 事件，同样会压入队列然后被事件派发器取出交给相对应的命令回复处理器，由该命令回复处理器将准备好的响应数据写入 socket 中，供客户端读取。
5. 命令回复处理器写完后，就会删除该 socket 的 AE_WRITABLE 事件与命令回复处理器的关联关系。

21、Redis事务及其相关面试题

什么是事务?

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

Redis事务的概念

Redis 事务的本质是通过MULTI、EXEC、WATCH等一组命令的集合。事务支持一次执行多个命令，一个事务中所有命令都会被序列化。在事务执行过程，会按照顺序串行化执行队列中的命令，其他客户端提交的命令请求不会插入到事务执行命令序列中。

总结说：redis事务就是一次性、顺序性、排他性的执行一个队列中的一系列命令。

Redis事务的三个阶段

1. 事务开始 MULTI
2. 命令入队
3. 事务执行 EXEC

事务执行过程中，如果服务端收到有EXEC、DISCARD、WATCH、MULTI之外的请求，将会把请求放入队列中排

事务管理（ACID）概述

- 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。

- 一致性（Consistency）

事务前后数据的完整性必须保持一致。

- 隔离性（Isolation）

多个事务并发执行时，一个事务的执行不应影响其他事务的执行

- 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，接下来即使数据库发生故障也不应该对其有任何影响

Redis的事务总是具有ACID中的一致性和隔离性，其他特性是不支持的。当服务器运行在*AOF*持久化模式下，并且appendfsync选项的值为always时，事务也具有耐久性。

Redis事务支持隔离性吗

Redis 是单进程程序，并且它保证在执行事务时，不会对事务进行中断，事务可以运行直到执行完所有事务队列中的命令为止。因此，Redis 的事务是总是带有隔离性的。

Redis事务保证原子性吗，支持回滚吗

Redis中，单条命令是原子性执行的，但事务不保证原子性，且没有回滚。事务中任意命令执行失败，其余的命令仍会被执行。

Redis事务其他实现

- 基于Lua脚本，Redis可以保证脚本内的命令一次性、按顺序地执行， 其同时也不提供事务运行错误的回滚，执行过程中如果部分命令运行错误，剩下的命令还是会继续运行完
- 基于中间标记变量，通过另外的标记变量来标识事务是否执行完成，读取数据时先读取该标记变量判断是否事务执行完成。但这样会需要额外写代码实现，比较繁琐

22、Redis是单线程的，如何提高多核CPU的利用率?

可以在同一个服务器部署多个Redis的实例，并把他们当作不同的服务器来使用，在某些时候，无论如何一个服务器是不够的， 所以，如果你想使用多个CPU，你可以考虑一下分片（shard）。

23、为什么要做Redis分区?

分区可以让Redis管理更大的内存，Redis将可以使用所有机器的内存。如果没有分区，你最多只能使用一台机器的内存。分区使Redis的计算能力通过简单地增加计算机得到成倍提升，Redis的网络带宽也会随着计算机和网卡的增加而成倍增长。

24、你知道有哪些Redis分区实现方案?

- 客户端分区就是在客户端就已经决定数据会被存储到哪个redis节点或者从哪个redis节点读取。大多数客户端已经实现了客户端分区。
- 代理分区 意味着客户端将请求发送给代理，然后代理决定去哪个节点写数据或者读数据。代理根据分区规则决定请求哪些Redis实例，然后根据Redis的响应结果返回给客户端。redis和memcached的一种代理实现就是Twemproxy
- 查询路由(Query routing) 的意思是客户端随机地请求任意一个redis实例，然后由Redis将请求转发给正确的Redis节点。Redis Cluster实现了一种混合形式的查询路由，但并不是直接将请求从一个redis节点转发到另一个redis节点，而是在客户端的帮助下直接redirected到正确的redis节点。

25、Redis分区有什么缺点?

- 涉及多个key的操作通常不会被支持。例如你不能对两个集合求交集，因为他们可能被存储到不同的Redis实例（实际上这种情况也有办法，但是不能直接使用交集指令）。
- 同时操作多个key,则不能使用Redis事务.
- 分区使用的粒度是key，不能使用一个非常长的排序key存储一个数据集（The partitioning granularity is the key, so it is not possible to shard a dataset with a single huge key like a very big sorted set）
- 当使用分区的时候，数据处理会非常复杂，例如为了备份你必须从不同的Redis实例和主机同时收集RDB / AOF文件。
- 分区时动态扩容或缩容可能非常复杂。Redis集群在运行时增加或者删除Redis节点，能做到最大程度对用户透明地数据再平衡，但其他一些客户端分区或者代理分区方法则不支持这种特性。然而，有一种预分片的技术也可以较好的解决这个问题。

26、如何解决 Redis 的并发竞争 Key 问题

所谓 Redis 的并发竞争 Key 的问题也就是多个系统同时对一个 key 进行操作，但是最后执行的顺序和我们期望的顺序不同，这样也就导致了结果的不同！

推荐一种方案：分布式锁（zookeeper 和 redis 都可以实现分布式锁）。（如果不存在 Redis 的并发竞争 Key 问题，不要使用分布式锁，这样会影响性能）

基于zookeeper临时有序节点可以实现的分布式锁。大致思想为：每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。完成业务流程后，删除对应的子节点释放锁。

在实践中，当然是从以可靠性为主。所以首推Zookeeper。

参考：https://www.jianshu.com/p/8bddd381de06

27、分布式Redis是前期做还是后期规模上来了再做好？为什么？

既然Redis是如此的轻量（单实例只使用1M内存），为防止以后的扩容，最好的办法就是一开始就启动较多实例。即便你只有一台服务器，你也可以一开始就让Redis以分布式的方式运行，使用分区，在同一台服务器上启动多个实例。

一开始就多设置几个Redis实例，例如32或者64个实例，对大多数用户来说这操作起来可能比较麻烦，但是从长久来看做这点牺牲是值得的。

这样的话，当你的数据不断增长，需要更多的Redis服务器时，你需要做的就是仅仅将Redis实例从一台服务迁移到另外一台服务器而已（而不用考虑重新分区的问题）。一旦你添加了另一台服务器，你需要将你一半的Redis实例从第一台机器迁移到第二台机器。

28、Redis相比Memcached有哪些优势?

数据类型：Memcached所有的值均是简单的字符串，Redis支持更为丰富的数据类型，支持string(字符串)，list(列表)，Set(集合)、Sorted Set(有序集合)、Hash(哈希)等。

持久化：Redis支持数据落地持久化存储，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。 memcache不支持数据持久存储 。

集群模式：Redis提供主从同步机制，以及 Cluster集群部署能力，能够提供高可用服务。Memcached没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据

性能对比：Redis的速度比Memcached快很多。

网络IO模型：Redis使用单线程的多路 IO 复用模型，Memcached使用多线程的非阻塞IO模式。

Redis支持服务器端的数据操作：Redis相比Memcached来说，拥有更多的数据结构和并支持更丰富的数据操作，通常在Memcached里，你需要将数据拿到客户端来进行类似的修改再set回去。

这大大增加了网络IO的次数和数据体积。在Redis中，这些复杂的操作通常和一般的GET/SET一样高效。所以，如果需要缓存能够支持更复杂的结构和操作，那么Redis会是不错的选择。

29、为什么要用 Redis 而不用 map/guava 做缓存?

缓存分为本地缓存和分布式缓存。以java为例，使用自带的map或者guava实现的是本地缓存，最主要的特点是轻量以及快速，生命周期随着jvm的销毁而结束，并且在多实例的情况下，每个实例都需要各自保存一份缓存，缓存不具有一致性。

使用Redis或memcached之类的称为分布式缓存，在多实例的情况下，各实例共用一份缓存数据，缓存具有一致性。缺点是需要保持Redis或memcached服务的高可用，整个程序架构上较为复杂。

对比:

1、Redis 可以用几十 G 内存来做缓存，Map 不行，一般 JVM 也就分几个 G 数据就够大了；

2、Redis 的缓存可以持久化，Map 是内存对象，程序一重启数据就没了；

3、Redis 可以实现分布式的缓存，Map 只能存在创建它的程序里；

4、Redis 可以处理每秒百万级的并发，是专业的缓存服务，Map 只是一个普通的对象；

5、Redis 缓存有过期机制，Map 本身无此功能；Redis 有丰富的 API，Map 就简单太多了；

6、Redis可单独部署，多个项目之间可以空想，本地内存无法共享；

7、Redis有专门的管理工具可以查看缓存数据。

30、如何选择合适的持久化方式

1、如果是数据不那么敏感，且可以从其他地方重新生成补回的，那么可以关闭持久化。

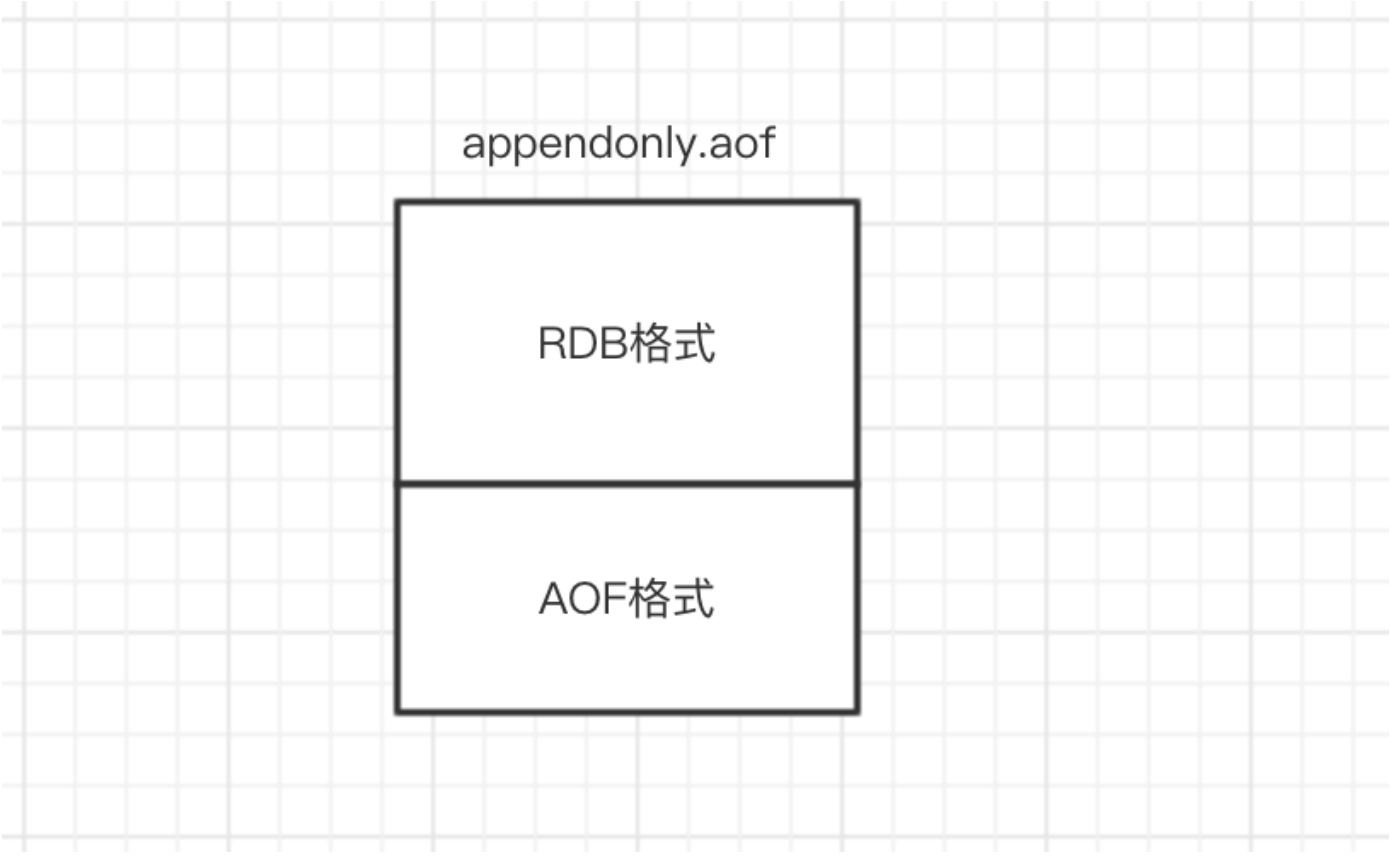
2、如果是数据比较重要，不想再从其他地方获取，且可以承受数分钟的数据丢失，比如缓存等，那么可以只使用RDB。

3、如果是用做内存数据库，要使用Redis的持久化，建议是RDB和AOF都开启，或者定期执行bgsave做快照备份，RDB方式更适合做数据的备份，AOF可以保证数据的不丢失。

补充：Redis4.0 对于持久化机制的优化

Redis4.0相对与3.X版本其中一个比较大的变化是4.0添加了新的混合持久化方式。

简单的说：新的AOF文件前半段是RDB格式的全量数据后半段是AOF格式的增量数据，如下图：



优势：混合持久化结合了RDB持久化 和 AOF 持久化的优点， 由于绝大部分都是RDB格式，加载速度快，同时结合AOF，增量的数据以AOF方式保存了，数据更少的丢失。

劣势：兼容性差，一旦开启了混合持久化，在4.0之前版本都不识别该aof文件，同时由于前部分是RDB格式，阅读性较差。

31、Redis key的过期时间和永久有效分别怎么设置?

通过expire或pexpire命令，客户端可以以秒或毫秒的精度为数据库中的某个键设置生存时间。

与expire和pexpire命令类似，客户端可以通过expireat和pexpireat命令，以秒或毫秒精度给数据库中的某个键设置过期时间，可以理解为：让某个键在某个时间点过期。

32、双写一致性方案一：先删除缓存，后更新数据库

该方案也会出问题，此时来了两个请求，请求 A（更新操作） 和请求 B（查询操作）

1. 请求A进行写操作，删除缓存
2. 请求B查询发现缓存不存在
3. 请求B去数据库查询得到旧值
4. 请求B将旧值写入缓存
5. 请求A将新值写入数据库。

上述情况就会导致不一致的情形出现。而且，如果不采用给缓存设置过期时间策略，该数据永远都是脏数据

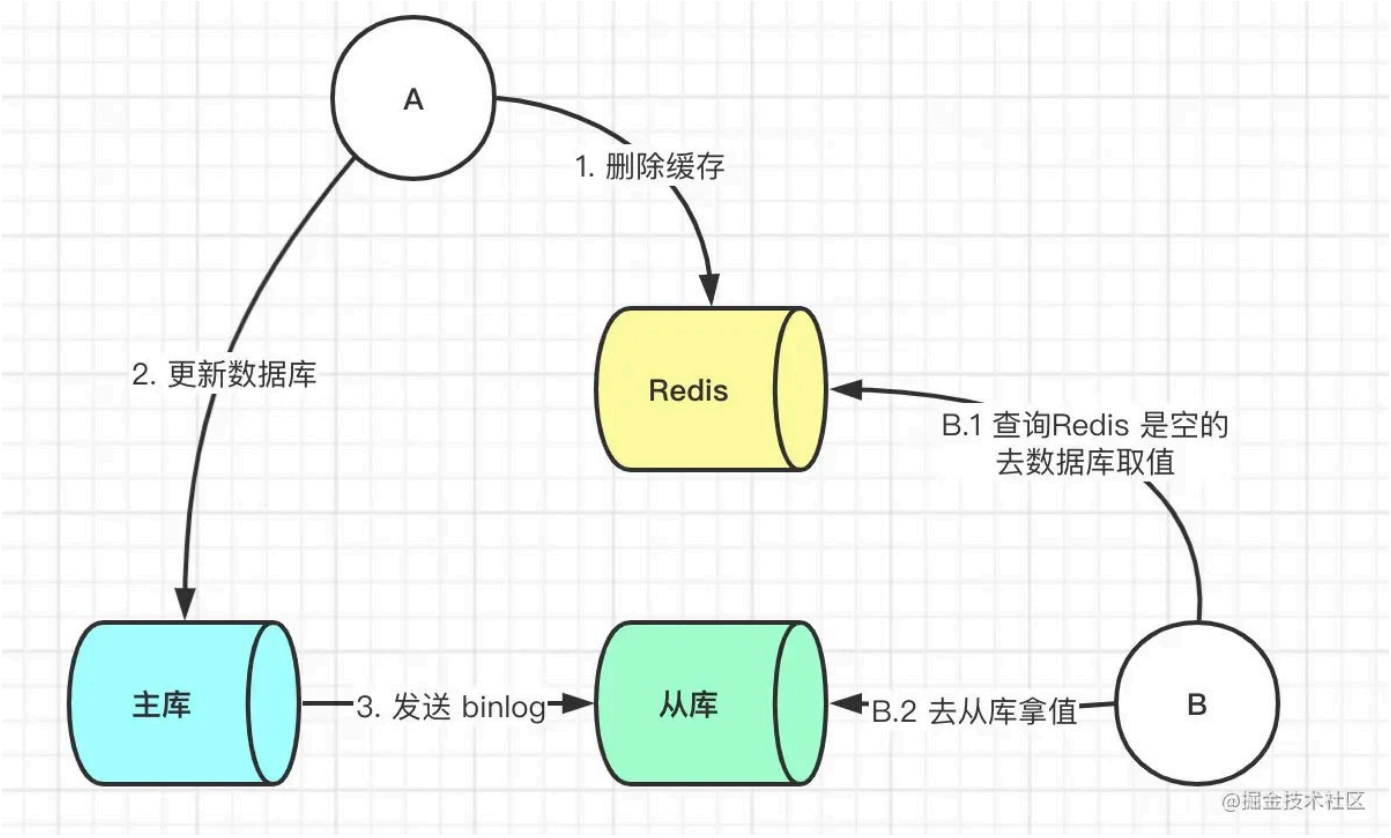
答案一：延时双删 最简单的解决办法延时双删

使用伪代码如下：

```
public void write(String key,Object data){
    Redis.delKey(key);
      db.updateData(data);
      Thread.sleep(1000);
      Redis.delKey(key);
  }
```

转化为中文描述就是 （1）先淘汰缓存 （2）再写数据库（这两步和原来一样） （3）休眠1秒，再次淘汰缓存，这么做，可以将1秒内所造成的缓存脏数据，再次删除。确保读请求结束，写请求可以删除读请求造成的缓存脏数据。自行评估自己的项目的读数据业务逻辑的耗时，写数据的休眠时间则在读数据业务逻辑的耗时基础上，加几百ms即可。

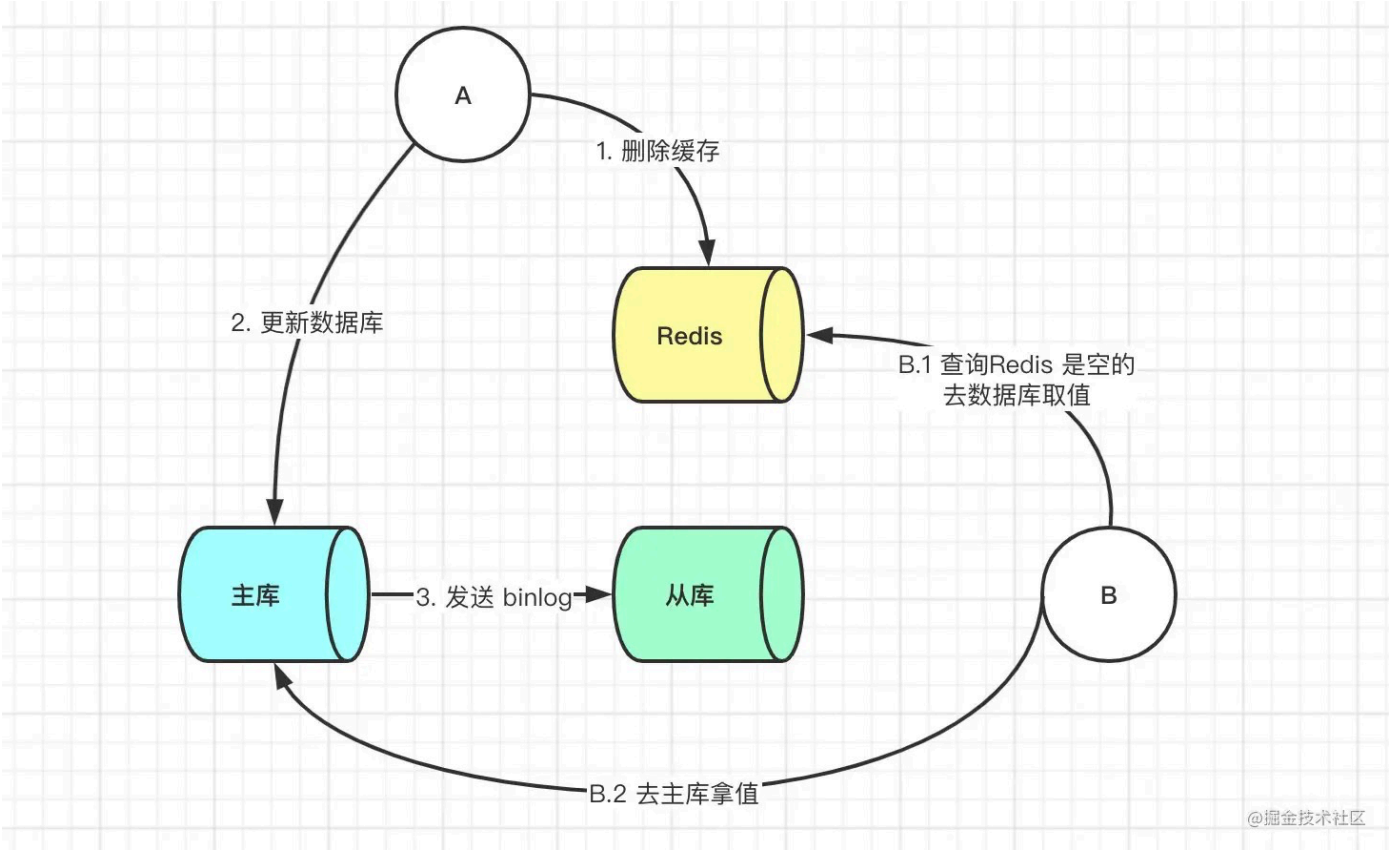
如果使用的是 Mysql 的读写分离的架构的话，那么其实主从同步之间也会有时间差。



此时来了两个请求，请求 A（更新操作） 和请求 B（查询操作）

1. 请求 A 更新操作，删除了 Redis
2. 请求主库进行更新操作，主库与从库进行同步数据的操作
3. 请 B 查询操作，发现 Redis 中没有数据
4. 去从库中拿去数据
5. 此时同步数据还未完成，拿到的数据是旧数据

此时的解决办法就是如果是对 Redis 进行填充数据的查询数据库操作，那么就强制将其指向主库进行查询。



答案二： 更新与读取操作进行异步串行化

采用更新与读取操作进行异步串行化

1、异步串行化

我在系统内部维护n个内存队列，更新数据的时候，根据数据的唯一标识，将该操作路由之后，发送到其中一个jvm内部的内存队列中（对同一数据的请求发送到同一个队列）。读取数据的时候，如果发现数据不在缓存中，并且此时队列里有更新库存的操作，那么将重新读取数据+更新缓存的操作，根据唯一标识路由之后，也将发送到同一个jvm内部的内存队列中。然后每个队列对应一个工作线程，每个工作线程串行地拿到对应的操作，然后一条一条的执行。

这样的话，一个数据变更的操作，先执行删除缓存，然后再去更新数据库，但是还没完成更新的时候，如果此时一个读请求过来，读到了空的缓存，那么可以先将缓存更新的请求发送到队列中，此时会在队列中积压，排在刚才更新库的操作之后，然后同步等待缓存更新完成，再读库。

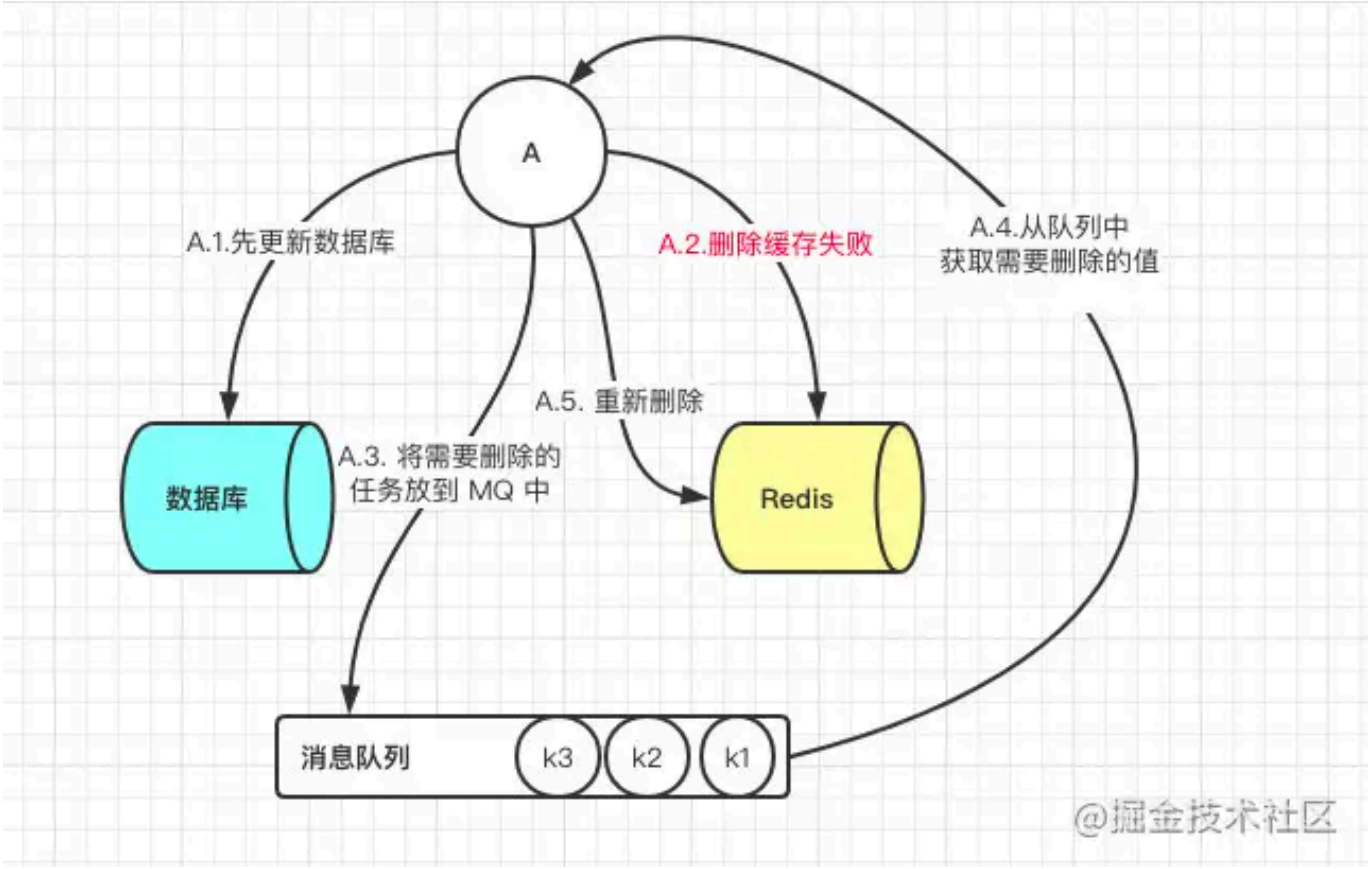
2、读操作去重

多个读库更新缓存的请求串在同一个队列中是没意义的，因此可以做过滤，如果发现队列中已经有了该数据的更新缓存的请求了，那么就不用再放进去了，直接等待前面的更新操作请求完成即可，待那个队列对应的工作线程完成了上一个操作（数据库的修改）之后，才会去执行下一个操作（读库更新缓存），此时会从数据库中读取最新的值，然后写入缓存中。

如果请求还在等待时间范围内，不断轮询发现可以取到值了，那么就直接返回；如果请求等待的时间超过一定时长，那么这一次直接从数据库中读取当前的旧值。（返回旧值不是又导致缓存和数据库不一致了么？那至少可以减少这个情况发生，因为等待超时也不是每次都是，几率很小吧。这里我想的是，如果超时了就直接读旧值，这时候仅仅是读库后返回而不放缓存）

33、双写一致性方案二：先更新数据库，后删除缓存

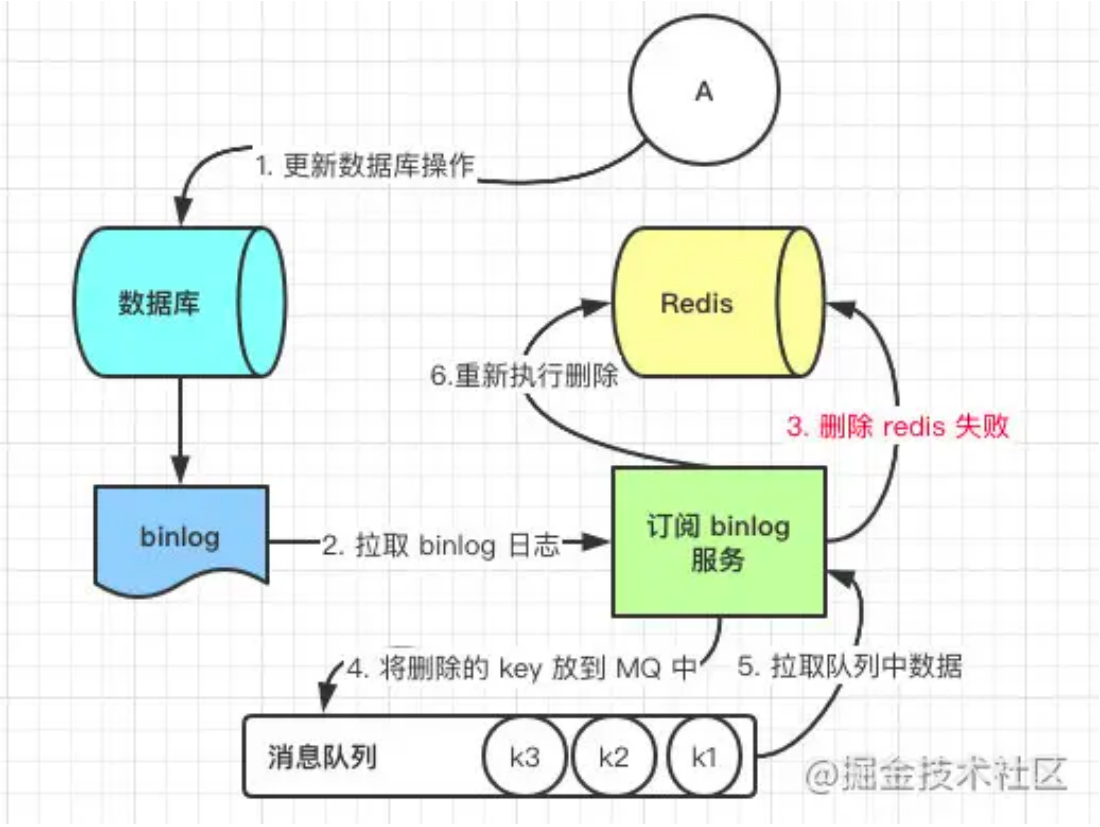
这一种情况也会出现问题，比如更新数据库成功了，但是在删除缓存的阶段出错了没有删除成功，那么此时再读取缓存的时候每次都是错误的数据了。



此时解决方案就是利用消息队列进行删除的补偿。具体的业务逻辑用语言描述如下：

1. 请求 A 先对数据库进行更新操作
2. 在对 Redis 进行删除操作的时候发现报错，删除失败
3. 此时将Redis 的 key 作为消息体发送到消息队列中
4. 系统接收到消息队列发送的消息后再次对 Redis 进行删除操作

但是这个方案会有一个缺点就是会对业务代码造成大量的侵入，深深的耦合在一起，所以这时会有一个优化的方案，我们知道对 Mysql 数据库更新操作后再 binlog 日志中我们都能够找到相应的操作，那么我们可以订阅 Mysql 数据库的 binlog 日志对缓存进行操作。



34、什么是缓存预热?

缓存预热是指系统上线后，提前将相关的缓存数据加载到缓存系统。避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题，用户直接查询事先被预热的缓存数据。

如果不进行预热，那么Redis初始状态数据为空，系统上线初期，对于高并发的流量，都会访问到数据库中， 对数据库造成流量的压力。

缓存预热解决方案：

- 数据量不大的时候，工程启动的时候进行加载缓存动作；
- 数据量大的时候，设置一个定时任务脚本，进行缓存的刷新；
- 数据量太大的时候，优先保证热点数据进行提前加载到缓存。

35、什么是缓存降级?

缓存降级是指缓存失效或缓存服务器挂掉的情况下，不去访问数据库，直接返回默认数据或访问服务的内存数据。降级一般是有损的操作，所以尽量减少降级对于业务的影响程度。

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级；比如可以参考日志级别设置预案：

一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；

警告：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警；

错误：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；

严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

36、Redis真的是单线程?

讨论 这个问题前，先看下 Redis的版本中两个重要的节点：

1. Redisv4.0（引入多线程处理异步任务）
2. Redis 6.0（在网络模型中实现多线程 I/O ）

所以，网络上说的Redis是单线程，通常是指在Redis 6.0之前，其核心网络模型使用的是单线程。

且Redis6.0引入多线程I/O，只是用来处理网络数据的读写和协议的解析，而执行命令依旧是单线程。

Redis在 v4.0 版本的时候就已经引入了的多线程来做一些异步操作，此举主要针对的是那些非常耗时的命令，通过将这些命令的执行进行异步化，避免阻塞单线程的事件循环。

在 Redisv4.0 之后增加了一些的非阻塞命令如 UNLINK、FLUSHALL ASYNC、FLUSHDB ASYNC。

37、Redis 6.0为何引入多线程?

很简单，就是 Redis的网络 I/O 瓶颈已经越来越明显了。

随着互联网的飞速发展，互联网业务系统所要处理的线上流量越来越大，Redis的单线程模式会导致系统消耗很多 CPU 时间在网络 I/O 上从而降低吞吐量，要提升 Redis的性能有两个方向：

- 优化网络 I/O 模块
- 提高机器内存读写的速度

后者依赖于硬件的发展，暂时无解。所以只能从前者下手，网络 I/O 的优化又可以分为两个方向：

- 零拷贝技术或者 DPDK 技术
- 利用多核优势

零拷贝技术有其局限性，无法完全适配 Redis这一类复杂的网络 I/O 场景，更多网络 I/O 对 CPU 时间的消耗和 Linux 零拷贝技术。而 DPDK 技术通过旁路网卡 I/O 绕过内核协议栈的方式又太过于复杂以及需要内核甚至是硬件的支持。

总结起来，Redis支持多线程主要就是两个原因：

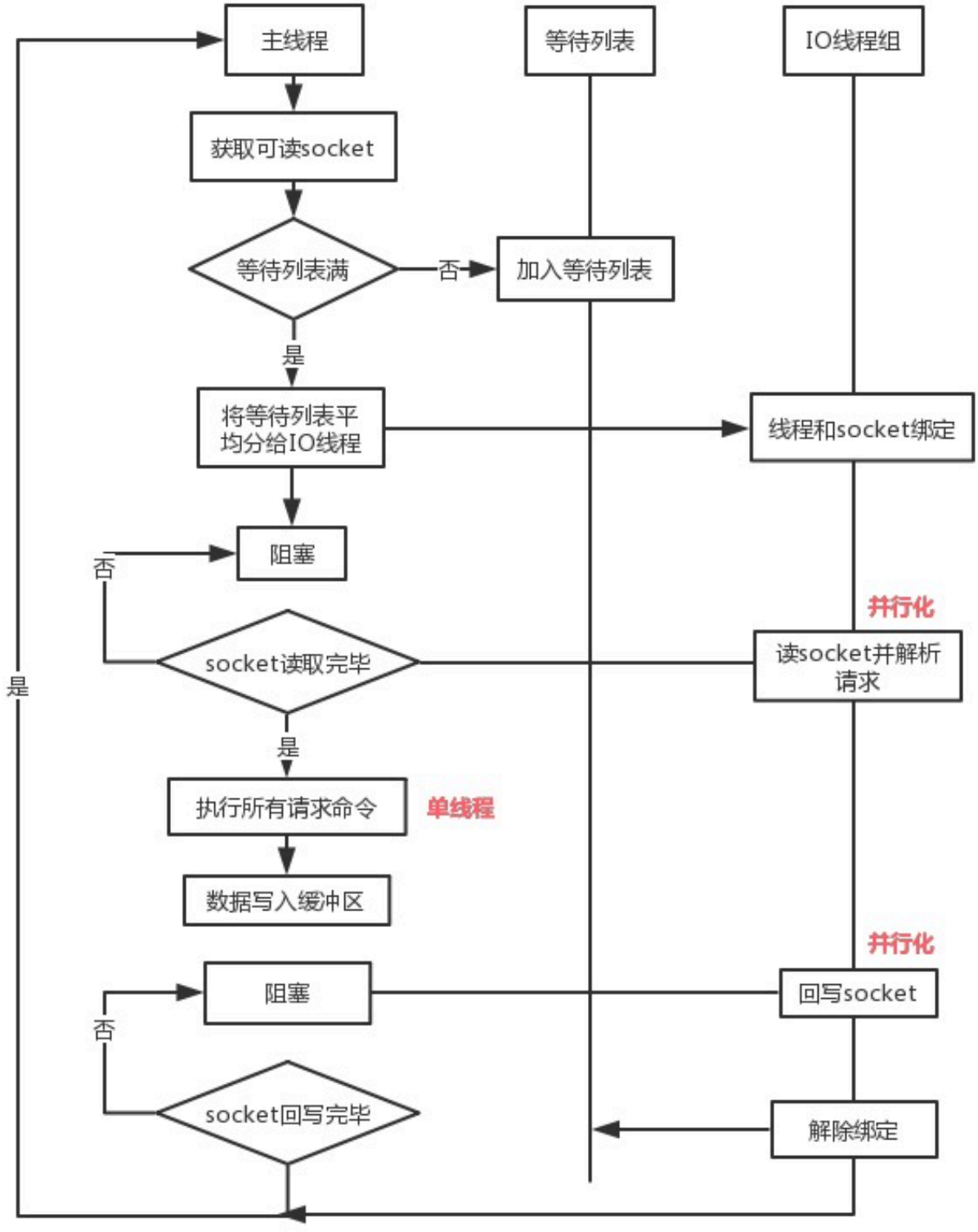
- 可以充分利用服务器 CPU 资源，目前主线程只能利用一个核
- 多线程任务可以分摊 Redis 同步 IO 读写负荷

38、Redis 6.0 多线程的实现机制?

流程简述如下：

- 主线程负责接收建立连接请求，获取 Socket 放入全局等待读处理队列。
- 主线程处理完读事件之后，通过 RR（Round Robin）将这些连接分配给这些 IO 线程。
- 主线程阻塞等待 IO 线程读取 Socket 完毕。

- 主线程通过单线程的方式执行请求命令，请求数据读取并解析完成，但并不执行。
- 主线程阻塞等待 IO 线程将数据回写 Socket 完毕。



该设计有如下特点：

- IO 线程要么同时在读 Socket，要么同时在写，不会同时读或写。
- IO 线程只负责读写 Socket 解析命令，不负责命令处理。

39、Redis 6.0 采用多线程后，性能的提升效果如何?

Redis 作者 antirez 在 RedisConf 2019 分享时曾提到：Redis 6 引入的多线程 IO 特性对性能提升至少是一倍以上。

国内也有大牛曾使用 unstable 版本在阿里云 esc 进行过测试，GET/SET 命令在 4 线程 IO 时性能相比单线程是几乎是翻倍了。

40、Redis 6.0开启多线程后，是否会存在线程并发安全问题?

从实现机制可以看出，Redis 的多线程部分只是用来处理网络数据的读写和协议解析，执行命令仍然是单线程顺序执行。

所以我们不需要去考虑控制 Key、Lua、事务，LPUSH/LPOP 等等的并发及线程安全问题。

41、Redis 6.0 与 Memcached 多线程模型的对比

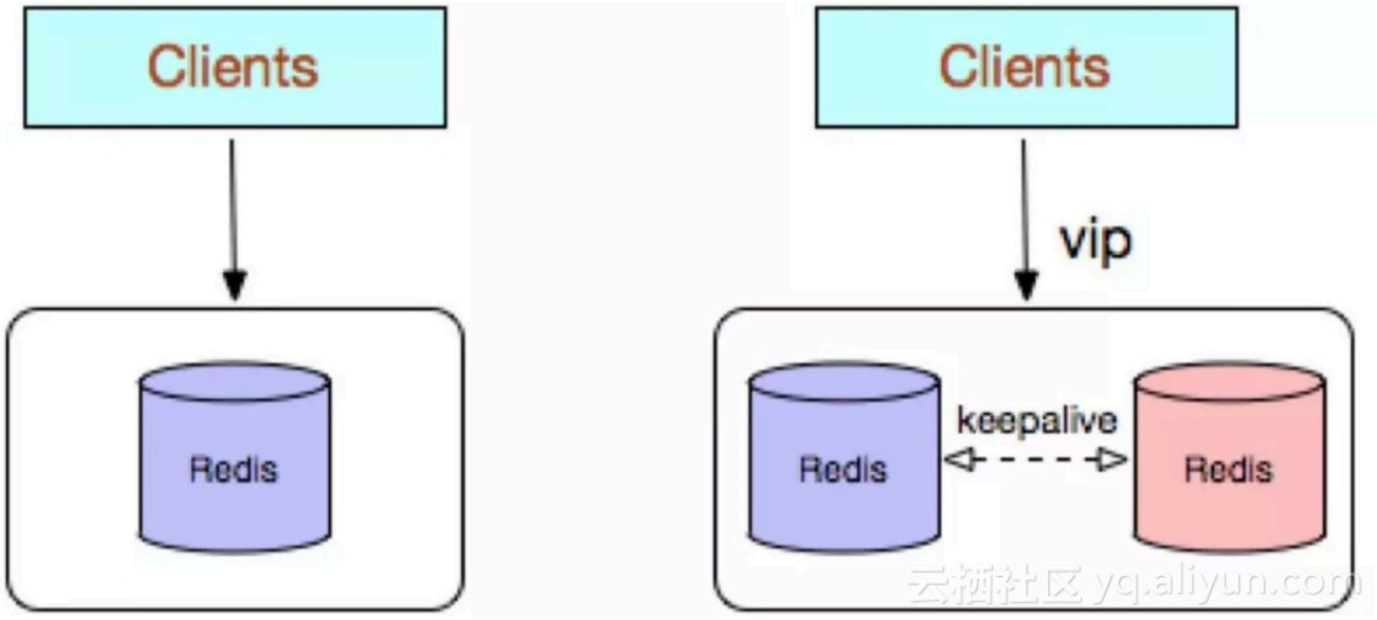
相同点：都采用了 Master 线程 -Worker 线程的模型。

不同点：Memcached 执行主逻辑也是在 Worker 线程里，模型更加简单，实现了真正的线程隔离，符合我们对线程隔离的常规理解。

而 Redis 把处理逻辑交还给 Master 线程，虽然一定程度上增加了模型复杂度，但也解决了线程并发安全等问题。

42、介绍下Redis单副本

Redis单副本，采用单个Redis节点部署架构，没有备用节点实时同步数据，不提供数据持久化和备份策略，适用于数据可靠性要求不高的纯缓存业务场景。



优点：

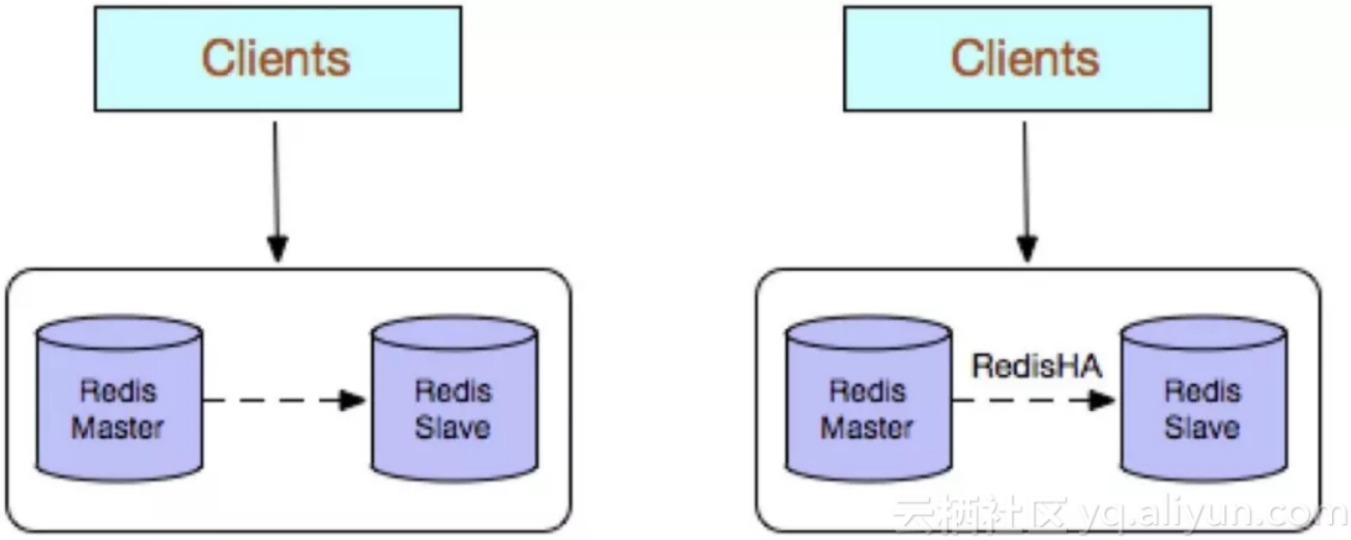
- 架构简单，部署方便；
- 高性价比：缓存使用时无需备用节点（单实例可用性可以用supervisor或crontab保证），当然为了满足业务的高可用性，也可以牺牲一个备用节点，但同时刻只有一个实例对外提供服务；
- 高性能。

缺点：

- 不保证数据的可靠性；
- 在缓存使用，进程重启后，数据丢失，即使有备用的节点解决高可用性，但是仍然不能解决缓存预热问题，因此不适用于数据可靠性要求高的业务；
- 高性能受限于单核CPU的处理能力（Redis是单线程机制），CPU为主要瓶颈，所以适合操作命令简单，排序、计算较少的场景。也可以考虑用Memcached替代。

43、介绍下Redis多副本（主从）

Redis多副本，采用主从（replication）部署结构，相较于单副本而言最大的特点就是主从实例间数据实时同步，并且提供数据持久化和备份策略。主从实例部署在不同的物理服务器上，根据公司的基础环境配置，可以实现同时对外提供服务和读写分离策略。



优点：

- 高可靠性：一方面，采用双机主备架构，能够在主库出现故障时自动进行主备切换，从库提升为主库提供服务，保证服务平稳运行；另一方面，开启数据持久化功能和配置合理的备份策略，能有效的解决数据误操作和数据异常丢失的问题；
- 读写分离策略：从节点可以扩展主库节点的读能力，有效应对大并发量的读操作。

缺点：

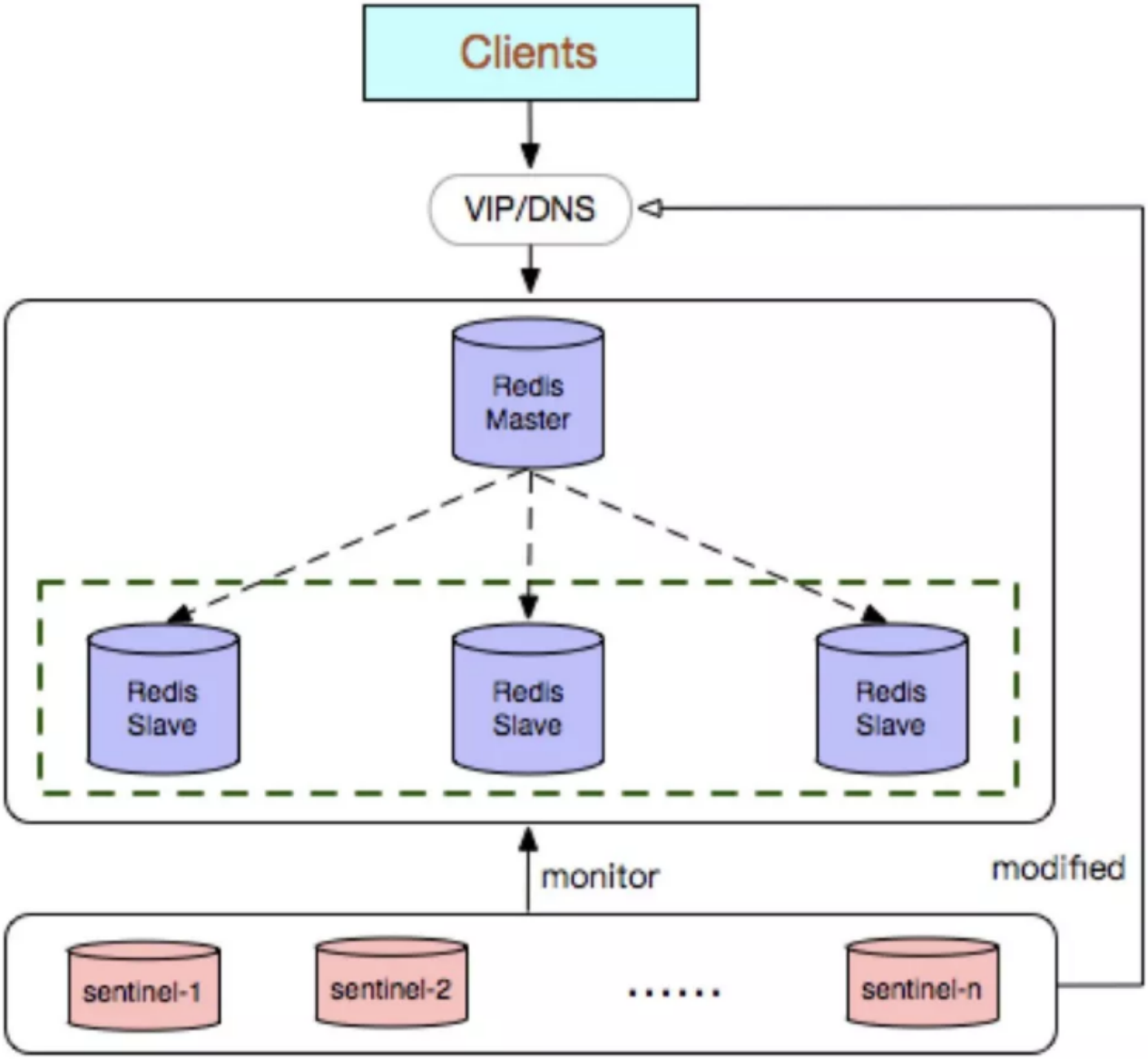
- 故障恢复复杂，如果没有RedisHA系统（需要开发），当主库节点出现故障时，需要手动将一个从节点晋升为主节点，同时需要通知业务方变更配置，并且需要让其它从库节点去复制新主库节点，整个过程需要人为干预，比较繁琐；
- 主库的写能力受到单机的限制，可以考虑分片；
- 主库的存储能力受到单机的限制，可以考虑Pika；
- 原生复制的弊端在早期的版本中也会比较突出，如：Redis复制中断后，Slave会发起psync，此时如果同步不成功，则会进行全量同步，主库执行全量备份的同时可能会造成毫秒或秒级的卡顿；又由于COW机制，导致极端情况下的主库内存溢出，程序异常退出或宕机；主库节点生成备份文件导致服务器磁盘IO和CPU（压缩）资源消耗；发送数GB大小的备份文件导致服务器出口带宽暴增，阻塞请求，建议升级到最新版本。

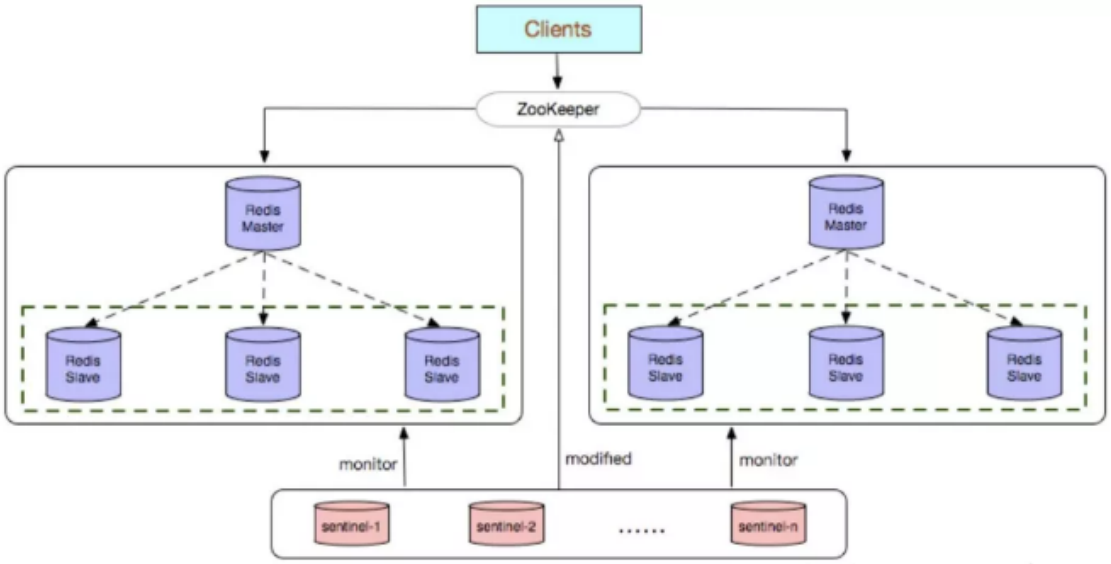
44、介绍下Redis Sentinel（哨兵）

> 主从模式下，当主服务器宕机后，需要手动把一台从服务器切换为主服务器，这就需要人工干预，费事费力，还会造成一段时间内服务不可用。这种方式并不推荐，实际生产中，我们优先考虑哨兵模式。这种模式下，master 宕机，哨兵会自动选举 master 并将其他的 slave 指向新的 master。

Redis Sentinel是社区版本推出的原生高可用解决方案，其部署架构主要包括两部分：Redis Sentinel集群和Redis数据集群。

其中Redis Sentinel集群是由若干Sentinel节点组成的分布式集群，可以实现故障发现、故障自动转移、配置中心和客户端通知。Redis Sentinel的节点数量要满足2n+1（n>=1）的奇数个。





优点：

- Redis Sentinel集群部署简单；
- 能够解决Redis主从模式下的高可用切换问题；
- 很方便实现Redis数据节点的线形扩展，轻松突破Redis自身单线程瓶颈，可极大满足Redis大容量或高性能的业务需求；
- 可以实现一套Sentinel监控一组Redis数据节点或多组数据节点。

缺点：

- 部署相对Redis主从模式要复杂一些，原理理解更繁琐；
- 资源浪费，Redis数据节点中slave节点作为备份节点不提供服务；
- Redis Sentinel主要是针对Redis数据节点中的主节点的高可用切换，对Redis的数据节点做失败判定分为主观下线和客观下线两种，对于Redis的从节点有对节点做主观下线操作，并不执行故障转移。
- 不能解决读写分离问题，实现起来相对复杂。

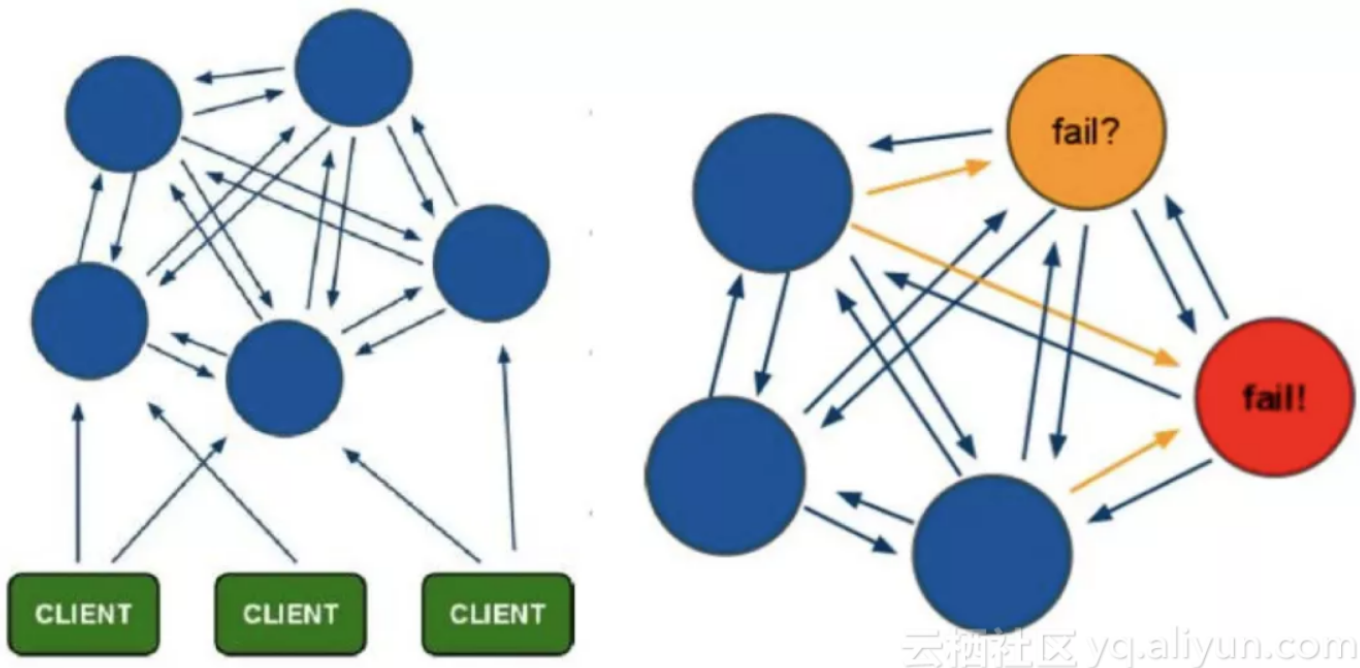
45、介绍下Redis Cluster

> Redis 的哨兵模式基本已经可以实现高可用，读写分离 ，但是在这种模式下每台 Redis 服务器都存储相同的数据，很浪费内存，所以在 Redis3.0 上加入了 Cluster 集群模式，实现了 Redis 的分布式存储，对数据进行分片，也就是说每台 Redis 节点上存储不同的内容。

Redis Cluster是社区版推出的Redis分布式集群解决方案，主要解决Redis分布式方面的需求，比如，当遇到单机内存，并发和流量等瓶颈的时候，Redis Cluster能起到很好的负载均衡的目的。

Redis Cluster集群节点最小配置6个节点以上（3主3从），其中主节点提供读写操作，从节点作为备用节点，不提供请求，只作为故障转移使用。

Redis Cluster采用虚拟槽分区，所有的键根据哈希函数映射到0～16383个整数槽内，每个节点负责维护一部分槽以及槽所印映射的键值数据。



优点：

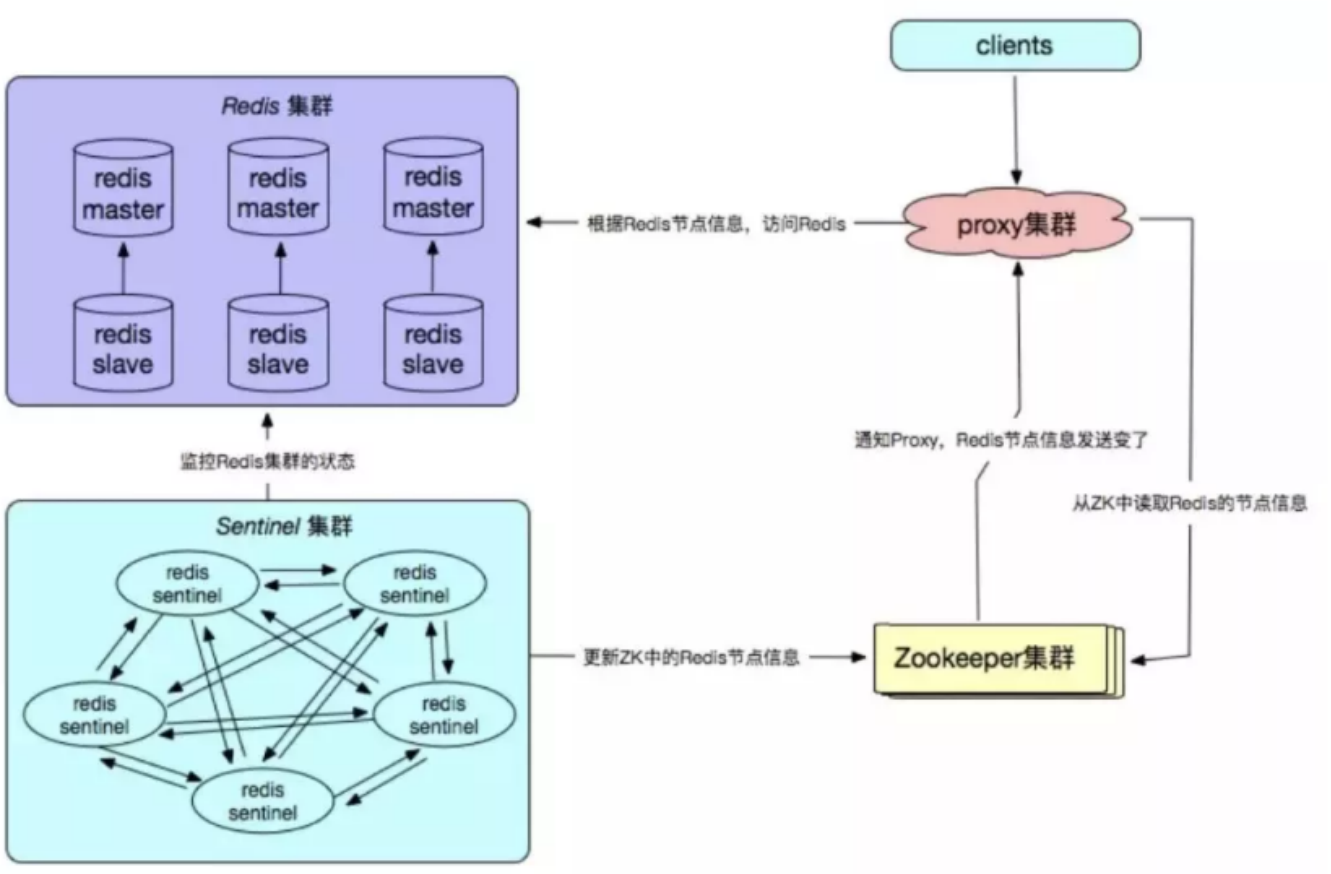
- 无中心架构；
- 数据按照slot存储分布在多个节点，节点间数据共享，可动态调整数据分布；
- 可扩展性：可线性扩展到1000多个节点，节点可动态添加或删除；
- 高可用性：部分节点不可用时，集群仍可用。通过增加Slave做standby数据副本，能够实现故障自动failover，节点之间通过gossip协议交换状态信息，用投票机制完成Slave到Master的角色提升；
- 降低运维成本，提高系统的扩展性和可用性。

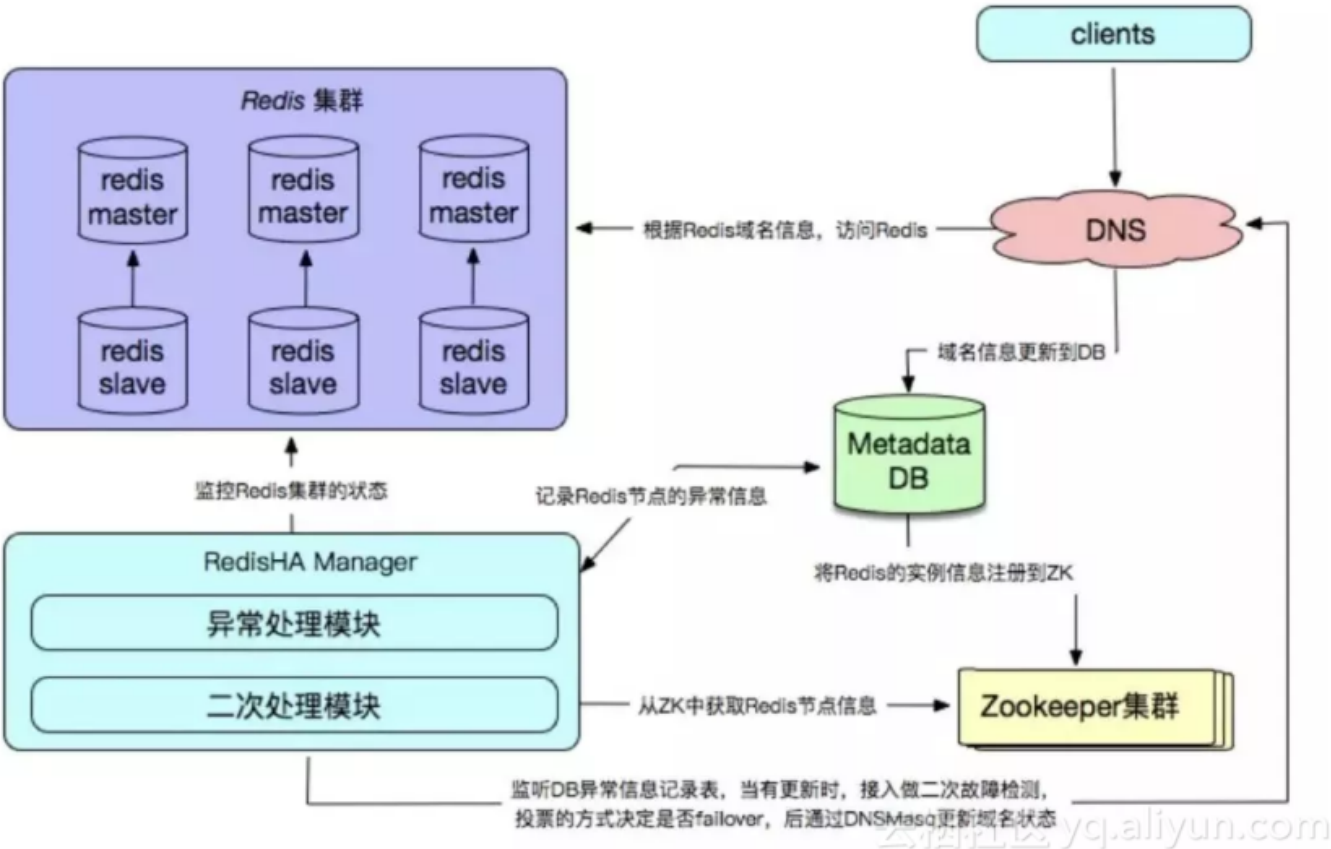
缺点：

- Client实现复杂，驱动要求实现Smart Client，缓存slots mapping信息并及时更新，提高了开发难度，客户端的不成熟影响业务的稳定性。目前仅JedisCluster相对成熟，异常处理部分还不完善，比如常见的“max redirect exception”。
- 节点会因为某些原因发生阻塞（阻塞时间大于clutser-node-timeout），被判断下线，这种failover是没有必要的。
- 数据通过异步复制，不保证数据的强一致性。
- 多个业务使用同一套集群时，无法根据统计区分冷热数据，资源隔离性较差，容易出现相互影响的情况。
- Slave在集群中充当“冷备”，不能缓解读压力，当然可以通过SDK的合理设计来提高Slave资源的利用率。
- Key批量操作限制，如使用mset、mget目前只支持具有相同slot值的Key执行批量操作。对于映射为不同slot值的Key由于Keys不支持跨slot查询，所以执行mset、mget、sunion等操作支持不友好。
- Key事务操作支持有限，只支持多key在同一节点上的事务操作，当多个Key分布于不同的节点上时无法使用事务功能。
- Key作为数据分区的最小粒度，不能将一个很大的键值对象如hash、list等映射到不同的节点。 不支持多数据库空间，单机下的Redis可以支持到16个数据库，集群模式下只能使用1个数据库空间，即db 0。
- 复制结构只支持一层，从节点只能复制主节点，不支持嵌套树状复制结构。
- 避免产生hot-key，导致主库节点成为系统的短板。
- 避免产生big-key，导致网卡撑爆、慢查询等。
- 重试时间应该大于cluster-node-time时间。
- Redis Cluster不建议使用pipeline和multi-keys操作，减少max redirect产生的场景。

46、介绍下Redis自研

Redis自研的高可用解决方案，主要体现在配置中心、故障探测和failover的处理机制上，通常需要根据企业业务的实际线上环境来定制化。


clients
Redis 集群
redis master
redis master
redis master
redis slave
redis slave
redis slave
根据Redis节点信息，访问Redis
proxy集群
通知Proxy，Redis节点信息发送变了
从ZK中读取Redis的节点信息
监控Redis集群的状态
Sentinel 集群
redis sentinel
redis sentinel
redis sentinel
redis sentinel
redis sentinel
更新ZK中的Redis节点信息
Zookeeper集群


clients
Redis 集群
redis master
redis master
redis master
redis slave
redis slave
redis slave
根据Redis域名信息，访问Redis
DNS
域名信息更新到DB
Metadata DB
监控Redis集群的状态
记录Redis节点的异常信息
RedisHA Manager
异常处理模块
二次处理模块
从ZK中获取Redis节点信息
Zookeeper集群
将Redis的实例信息注册到ZK
监听DB异常信息记录表，当有更新时，接入做二次故障检测，
投票的方式决定是否failover，后通过DNSMasq更新域名状态

优点：

- 高可靠性、高可用性；

- 自主可控性高；
- 贴切业务实际需求，可缩性好，兼容性好。

缺点：

- 实现复杂，开发成本高；
- 需要建立配套的周边设施，如监控，域名服务，存储元数据信息的数据库等；
- 维护成本高。

47、Redis高可用方案具体怎么实施?

使用官方推荐的哨兵(sentinel)机制就能实现，当主节点出现故障时，由Sentinel自动完成故障发现和转移，并通知应用方，实现高可用性。它有四个主要功能：

- 集群监控，负责监控Redis master和slave进程是否正常工作。
- 消息通知，如果某个Redis实例有故障，那么哨兵负责发送消息作为报警通知给管理员。
- 故障转移，如果master node挂掉了，会自动转移到slave node上。
- 配置中心，如果故障转移发生了，通知client客户端新的master地址。

48、了解主从复制的原理吗?

1、主从架构的核心原理

当启动一个slave node的时候，它会发送一个PSYNC命令给master node

如果这是slave node重新连接master node，那么master node仅仅会复制给slave部分缺少的数据; 否则如果是slave node第一次连接master node，那么会触发一次full resynchronization

开始full resynchronization的时候，master会启动一个后台线程，开始生成一份RDB快照文件，同时还会将从客户端收到的所有写命令缓存在内存中。RDB文件生成完毕之后，master会将这个RDB发送给slave，slave会先写入本地磁盘，然后再从本地磁盘加载到内存中。然后master会将内存中缓存的写命令发送给slave，slave也会同步这些数据。

slave node如果跟master node有网络故障，断开了连接，会自动重连。master如果发现有多个slave node都来重新连接，仅仅会启动一个rdb save操作，用一份数据服务所有slave node。

2、主从复制的断点续传

从Redis 2.8开始，就支持主从复制的断点续传，如果主从复制过程中，网络连接断掉了，那么可以接着上次复制的地方，继续复制下去，而不是从头开始复制一份

master node会在内存中常见一个backlog，master和slave都会保存一个replica offset还有一个master id，offset就是保存在backlog中的。如果master和slave网络连接断掉了，slave会让master从上次的replica offset开始继续复制

但是如果没有找到对应的offset，那么就会执行一次resynchronization

3、无磁盘化复制

master在内存中直接创建rdb，然后发送给slave，不会在自己本地落地磁盘了

repl-diskless-sync repl-diskless-sync-delay，等待一定时长再开始复制，因为要等更多slave重新连接过来

4、过期key处理

slave不会过期key，只会等待master过期key。如果master过期了一个key，或者通过LRU淘汰了一个key，那么会模拟一条del命令发送给slave。

49、由于主从延迟导致读取到过期数据怎么处理?

1、通过scan命令扫库：当Redis中的key被scan的时候，相当于访问了该key，同样也会做过期检测，充分发挥Redis惰性删除的策略。这个方法能大大降低了脏数据读取的概率，但缺点也比较明显，会造成一定的数据库压力，否则影响线上业务的效率。

2、Redis加入了一个新特性来解决主从不一致导致读取到过期数据问题，增加了key是否过期以及对主从库的判断，如果key已过期，当前访问的master则返回null；当前访问的是从库，且执行的是只读命令也返回null。

50、主从复制的过程中如果因为网络原因停止复制了会怎么样?

如果出现网络故障断开连接了，会自动重连的，从Redis 2.8开始，就支持主从复制的断点续传，可以接着上次复制的地方，继续复制下去，而不是从头开始复制一份。

master如果发现有多个slave node都来重新连接，仅仅会启动一个rdb save操作，用一份数据服务所有slave node。

master node会在内存中创建一个backlog，master和slave都会保存一个replica offset，还有一个master id，offset就是保存在backlog中的。如果master和slave网络连接断掉了，slave会让master从上次的replica offset开始继续复制。

但是如果没有找到对应的offset，那么就会执行一次resynchronization全量复制。

51、Redis主从架构数据会丢失吗，为什么?

有两种数据丢失的情况：

1、异步复制导致的数据丢失：因为master -> slave的复制是异步的，所以可能有部分数据还没复制到slave，master就宕机了，此时这些部分数据就丢失了。

2、脑裂导致的数据丢失：某个master所在机器突然脱离了正常的网络，跟其他slave机器不能连接，但是实际上master还运行着，此时哨兵可能就会认为master宕机了，然后开启选举，将其他slave切换成了master。这个时候，集群里就会有两个master，也就是所谓的脑裂。此时虽然某个slave被切换成了master，但是可能client还没来得及切换到新的master，还继续写向旧master的数据可能也丢失了。因此旧master再次恢复的时候，会被作为一个slave挂到新的master上去，自己的数据会清空，重新从新的master复制数据。

52、如何解决主从架构数据丢失的问题?

数据丢失的问题是不可避免的，但是我们可以尽量减少。

在Redis的配置文件里设置参数

```
min-slaves-to-write 1
min-slaves-max-lag 10
min-slaves-to-write默认情况下是0，min-slaves-max-lag默认情况下是10。
```

上面的配置的意思是要求至少有1个slave，数据复制和同步的延迟不能超过10秒。如果说一旦所有的slave，数据复制和同步的延迟都超过了10秒钟，那么这个时候，master就不会再接收任何请求了。

减小min-slaves-max-lag参数的值，这样就可以避免在发生故障时大量的数据丢失，一旦发现延迟超过了该值就不会往master中写入数据。

那么对于client，我们可以采取降级措施，将数据暂时写入本地缓存和磁盘中，在一段时间后重新写入master来保证数据不丢失；也可以将数据写入kafka消息队列，隔一段时间去消费kafka中的数据。

53、Redis哨兵是怎么工作的?

1. 每个Sentinel以每秒钟一次的频率向它所知的Master，Slave以及其他 Sentinel 实例发送一个 PING 命令。
2. 如果一个实例（instance）距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值， 则这个实例会被当前 Sentinel 标记为主观下线。
3. 如果一个Master被标记为主观下线，则正在监视这个Master的所有 Sentinel 要以每秒一次的频率确认Master的确进入了主观下线状态。
4. 当有足够数量的 Sentinel（大于等于配置文件指定的值）在指定的时间范围内确认Master的确进入了主观下线状态， 则Master会被标记为客观下线 。
5. 当Master被 Sentinel 标记为客观下线时，Sentinel 向下线的 Master 的所有 Slave 发送 INFO 命令的频率会从 10 秒一次改为每秒一次 （在一般情况下， 每个 Sentinel 会以每 10 秒一次的频率向它已知的所有Master，Slave发送 INFO 命令 ）。
6. 若没有足够数量的 Sentinel 同意 Master 已经下线， Master 的客观下线状态就会变成主观下线。若 Master 重新向 Sentinel 的 PING 命令返回有效回复， Master 的主观下线状态就会被移除。
7. sentinel节点会与其他sentinel节点进行“沟通”，投票选举一个sentinel节点进行故障处理，在从节点中选取一个主节点，其他从节点挂载到新的主节点上自动复制新主节点的数据。

54、故障转移时会从剩下的slave选举一个新的master，被选举为master的标准是什么?

如果一个master被认为odown了，而且majority哨兵都允许了主备切换，那么某个哨兵就会执行主备切换操作，此时首先要选举一个slave来，会考虑slave的一些信息。

1、跟master断开连接的时长。

如果一个slave跟master断开连接已经超过了down-after-milliseconds的10倍，外加master宕机的时长，那么slave就被认为不适合选举为master.

```
( down-after-milliseconds * 10) + milliseconds_since_master_is_in_SDOWN_state
```

2、slave优先级。

按照slave优先级进行排序，slave priority越低，优先级就越高

3、复制offset。

如果slave priority相同，那么看replica offset，哪个slave复制了越多的数据，offset越靠后，优先级就越高

4、run id

如果上面两个条件都相同，那么选择一个run id比较小的那个slave。

55、同步配置的时候其他哨兵根据什么更新自己的配置呢?

执行切换的那个哨兵，会从要切换到的新master（salve->master）那里得到一个configuration epoch，这就是一个version号，每次切换的version号都必须是唯一的。

如果第一个选举出的哨兵切换失败了，那么其他哨兵，会等待failover-timeout时间，然后接替继续执行切换，此时会重新获取一个新的configuration epoch 作为新的version号。

这个version号就很重要了，因为各种消息都是通过一个channel去发布和监听的，所以一个哨兵完成一次新的切换之后，新的master配置是跟着新的version号的，其他的哨兵都是根据版本号的大小来更新自己的master配置的。

56、为什么Redis哨兵集群只有2个节点无法正常工作?

哨兵集群必须部署2个以上节点。

如果两个哨兵实例，即两个Redis实例，一主一从的模式。

则Redis的配置quorum=1，表示一个哨兵认为master宕机即可认为master已宕机。

但是如果是机器1宕机了，那哨兵1和master都宕机了，虽然哨兵2知道master宕机了，但是这个时候，需要majority，也就是大多数哨兵都是运行的，2个哨兵的majority就是2（2的majority=2，3的majority=2，5的majority=3，4的majority=2），2个哨兵都运行着，就可以允许执行故障转移。

但此时哨兵1没了就只有1个哨兵了了，此时就没有majority来允许执行故障转移，所以故障转移不会执行。

57、Redis cluster中是如何实现数据分布的？这种方式有什么优点?

Redis cluster有固定的16384个hash slot（哈希槽），对每个key计算CRC16值，然后对16384取模，可以获取key对应的hash slot。

Redis cluster中每个master都会持有部分slot（槽），比如有3个master，那么可能每个master持有5000多个hash slot。

hash slot让node的增加和移除很简单，增加一个master，就将其他master的hash slot移动部分过去，减少一个master，就将它的hash slot移动到其他master上去。每次增加或减少master节点都是对16384取模，而不是根据master数量，这样原本在老的master上的数据不会因master的新增或减少而找不到。并且增加或减少master时Redis cluster移动hash slot的成本是非常低的。

58、Redis cluster节点间通信是什么机制?

Redis cluster节点间采取gossip协议进行通信，所有节点都持有一份元数据，不同的节点如果出现了元数据的变更之后U不断地i将元数据发送给其他节点让其他节点进行数据变更。

节点互相之间不断通信，保持整个集群所有节点的数据是完整的。 主要交换故障信息、节点的增加和移除、hash slot信息等。

这种机制的好处在于，元数据的更新比较分散，不是集中在一个地方，更新请求会陆陆续续，打到所有节点上去更新，有一定的延时，降低了压力;

缺点，元数据更新有延时，可能导致集群的一些操作会有一些滞后。

59、什么是分布式锁？为什么用分布式锁?

锁在程序中的作用就是同步工具，保证共享资源在同一时刻只能被一个线程访问，Java中的锁我们都很熟悉了，像synchronized 、Lock都是我们经常使用的，但是Java的锁只能保证单机的时候有效，分布式集群环境就无能为力了，这个时候我们就需要用到分布式锁。

分布式锁，顾名思义，就是分布式项目开发中用到的锁，可以用来控制分布式系统之间同步访问共享资源。

思路是：在整个系统提供一个全局、唯一的获取锁的“东西”，然后每个系统在需要加锁时，都去问这个“东西”拿到一把锁，这样不同的系统拿到的就可以认为是同一把锁。至于这个“东西”，可以是Redis、Zookeeper，也可以是数据库。

一般来说，分布式锁需要满足的特性有这么几点：

1、互斥性：在任何时刻，对于同一条数据，只有一台应用可以获取到分布式锁；

2、高可用性：在分布式场景下，一小部分服务器宕机不影响正常使用，这种情况就需要将提供分布式锁的服务以集群的方式部署；

3、防止锁超时：如果客户端没有主动释放锁，服务器会在一段时间之后自动释放锁，防止客户端宕机或者网络不可达时产生死锁；

4、独占性：加锁解锁必须由同一台服务器进行，也就是锁的持有者才可以释放锁，不能出现你加的锁，别人给你解锁了。

60、常见的分布式锁有哪些解决方案?

实现分布式锁目前有三种流行方案，即基于关系型数据库、Redis、ZooKeeper 的方案

1、基于关系型数据库，如MySQL

基于关系型数据库实现分布式锁，是依赖数据库的唯一性来实现资源锁定，比如主键和唯一索引等。

缺点：

- 这把锁强依赖数据库的可用性，数据库是一个单点，一旦数据库挂掉，会导致业务系统不可用。
- 这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在数据库中，其他线程无法再获得到锁。
- 这把锁只能是非阻塞的，因为数据的insert操作，一旦插入失败就会直接报错。没有获得锁的线程并不会进入排队队列，要想再次获得锁就要再次触发获得锁操作。
- 这把锁是非重入的，同一个线程在没有释放锁之前无法再次获得该锁。因为数据中数据已经存在了。

2、基于Redis实现

优点：

Redis 锁实现简单，理解逻辑简单，性能好，可以支撑高并发的获取、释放锁操作。

缺点：

- Redis 容易单点故障，集群部署，并不是强一致性的，锁的不够健壮；
- key 的过期时间设置多少不明确，只能根据实际情况调整；
- 需要自己不断去尝试获取锁，比较消耗性能。

3、基于zookeeper

优点：

zookeeper 天生设计定位就是分布式协调，强一致性，锁很健壮。如果获取不到锁，只需要添加一个监听器就可以了，不用一直轮询，性能消耗较小。

缺点：

在高请求高并发下，系统疯狂的加锁释放锁，最后 zk 承受不住这么大的压力可能会存在宕机的风险。

61、Redis实现分布式锁

分布式锁的三个核心要素

1、加锁

使用setnx来加锁。key是锁的唯一标识，按业务来决定命名，value这里设置为test。

```
setx key test
```

当一个线程执行setnx返回1，说明key原本不存在，该线程成功得到了锁；当一个线程执行setnx返回0，说明key已经存在，该线程抢锁失败；

2、解锁

有加锁就得有解锁。当得到的锁的线程执行完任务，需要释放锁，以便其他线程可以进入。释放锁的最简单方式就是执行del指令。

```
del key
```

释放锁之后，其他线程就可以继续执行setnx命令来获得锁。

3、锁超时

锁超时知道的是：如果一个得到锁的线程在执行任务的过程中挂掉，来不及显式地释放锁，这块资源将会永远被锁住，别的线程北向进来。

所以，setnx的key必须设置一个超时时间，以保证即使没有被显式释放，这把锁也要在一段时间后自动释放。setnx不支持超时参数，所以需要额外指令，

```
expire key 30
```

上述分布式锁存在的问题

通过上述setnx 、del和expire实现的分布式锁还是存在着一些问题。

1、SETNX 和 EXPIRE 非原子性

假设一个场景中，某一个线程刚执行setnx，成功得到了锁。此时setnx刚执行成功，还未来得及执行expire命令，节点就挂掉了。此时这把锁就没有设置过期时间，别的线程就再也无法获得该锁。

解决措施:

由于setnx指令本身是不支持传入超时时间的，而在Redis2.6.12版本上为set指令增加了可选参数, 用法如下：

```
SET key value [EX seconds][PX milliseconds] [NX|XX]
```

- EX second: 设置键的过期时间为second秒；
- PX millisecond：设置键的过期时间为millisecond毫秒；
- NX：只在键不存在时，才对键进行设置操作；
- XX：只在键已经存在时，才对键进行设置操作；
- SET操作完成时，返回OK，否则返回nil。

2、锁误解除

如果线程 A 成功获取到了锁，并且设置了过期时间 30 秒，但线程 A 执行时间超过了 30 秒，锁过期自动释放，此时线程 B 获取到了锁；随后 A 执行完成，线程 A 使用 DEL 命令来释放锁，但此时线程 B 加的锁还没有执行完成，线程 A 实际释放的线程 B 加的锁。

解决办法：

在del释放锁之前加一个判断，验证当前的锁是不是自己加的锁。

具体在加锁的时候把当前线程的id当做value，可生成一个 UUID 标识当前线程，在删除之前验证key对应的value是不是自己线程的id。

还可以使用 lua 脚本做验证标识和解锁操作。

3、超时解锁导致并发

如果线程 A 成功获取锁并设置过期时间 30 秒，但线程 A 执行时间超过了 30 秒，锁过期自动释放，此时线程 B 获取到了锁，线程 A 和线程 B 并发执行。

A、B 两个线程发生并发显然是不被允许的，一般有两种方式解决该问题：

- 将过期时间设置足够长，确保代码逻辑在锁释放之前能够执行完成。
- 为获取锁的线程增加守护线程，为将要过期但未释放的锁增加有效时间。

4、不可重入

当线程在持有锁的情况下再次请求加锁，如果一个锁支持一个线程多次加锁，那么这个锁就是可重入的。如果一个不可重入锁被再次加锁，由于该锁已经被持有，再次加锁会失败。Redis 可通过对锁进行重入计数，加锁时加 1，解锁时减 1，当计数归 0 时释放锁。

5、无法等待锁释放

上述命令执行都是立即返回的，如果客户端可以等待锁释放就无法使用。

- 可以通过客户端轮询的方式解决该问题，当未获取到锁时，等待一段时间重新获取锁，直到成功获取锁或等待超时。这种方式比较消耗服务器资源，当并发量比较大时，会影响服务器的效率。
- 另一种方式是使用 Redis 的发布订阅功能，当获取锁失败时，订阅锁释放消息，获取锁成功后释放时，发送锁释放消息。 具体实现参考：https://xiaomi-info.github.io/2019/12/17/Redis-distributed-lock/

62、RedLock的原理

假设有5个完全独立的Redis主服务器

1、获取当前时间戳

2、client尝试按照顺序使用相同的key,value获取所有Redis服务的锁，在获取锁的过程中的获取时间比锁过期时间短很多，这是为了不要过长时间等待已经关闭的Redis服务。并且试着获取下一个Redis实例。

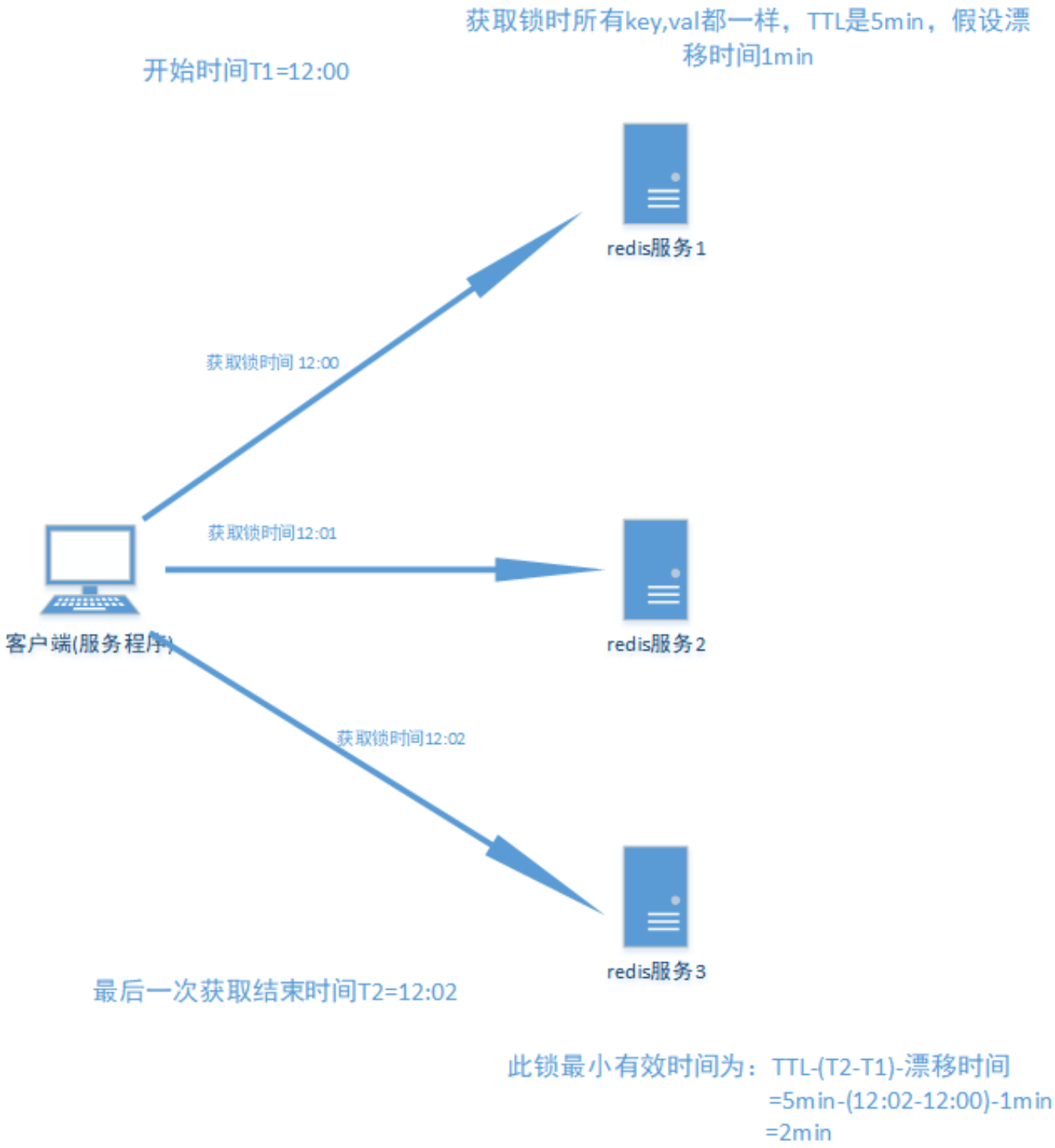
比如：TTL为5s,设置获取锁最多用1s，所以如果一秒内无法获取锁，就放弃获取这个锁，从而尝试获取下个锁

3、client通过获取所有能获取的锁后的时间减去第一步的时间，这个时间差要小于TTL时间并且至少有3个Redis实例成功获取锁，才算真正的获取锁成功

4、如果成功获取锁，则锁的真正有效时间是 TTL减去第三步的时间差 的时间；比如：TTL 是5s,获取所有锁用了2s,则真正锁有效时间为3s(其实应该再减去时钟漂移);

5、如果客户端由于某些原因获取锁失败，便会开始解锁所有Redis实例；因为可能已经获取了小于3个锁，必须释放，否则影响其他client获取锁

算法示意图如下：



消息队列

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**Java面试题**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**Java面试题**」，即可获取最新版 PDF。

1、消息队列的基本作用?

消息队列的主要作用是：解耦、异步、削峰。

- **解耦**

A 系统通过接口调用发送数据到 B、C、D 三个系统。那如果现在 E 系统也要这个数据呢？那如果 C 系统现在不需要了呢？现在 A 系统又要发送第二种数据了呢？这样的话 A 系统的维护成本就非常的高，而且 A 系统要时时刻刻考虑B、C、D、E 四个系统如果出现故障该怎么办？A 系统是重发还是先把消息保存起来呢？使用消息队列就可以解决这个问题。A 系统只负责生产数据，不需要考虑消息被哪个系统来消费。

- **异步**

A 系统需要发送个请求给 B 系统处理，由于 B 系统需要查询数据库花费时间较长，以至于 A 系统要等待 B 系统处理完毕后再发送下个请求，造成 A 系统资源浪费。使用消息队列后，A 系统生产完消息后直接丢进消息队列，不用等待 B 系统的结果，直接继续去干自己的事情了。

- **削峰**

A 系统调用 B 系统处理数据，每天 0 点到 12 点，A 系统风平浪静，每秒并发请求数量就 100 个。结果每次一到 12 点 ~ 13 点，每秒并发请求数量突然会暴增到 1 万条。但是 B 系统最大的处理能力就只能是每秒钟处理 1000 个请求，这样系统很容易就会崩掉。这种情况可以引入消息队列，把请求数据先存入消息队列中，消费系统再根据自己的消费能力拉取消费。

2、消息队列的优缺点有哪些?

- **优点**

消息队列的优点就是：解耦、异步、削峰。

- **缺点**

1. 降低系统的可用性：系统引入的外部依赖越多，越容易挂掉；
2. 系统复杂度提高：使用 MQ 后可能需要保证消息没有被重复消费、处理消息丢失的情况、保证消息传递的顺序性等等问题；
3. 一致性问题：A 系统处理完了直接返回成功了，但问题是：要是 B、C、D 三个系统那里，B 和 D 两个系统写

库成功了，结果 C 系统写库失败了，就造成数据不一致了。

3、如何保证消息队列的高可用?

根据不同的 MQ 或者你用过的 MQ 进行回答：

- **RabbitMQ：镜像集群模式**

RabbitMQ 是基于主从做高可用性的，Rabbitmq有三种模式：单机模式、普通集群模式、镜像集群模式。单机模式一般在生产环境中很少用，普通集群模式只是提高了系统的吞吐量，让集群中多个节点来服务某个 Queue 的读写操作。那么真正实现 RabbitMQ 高可用的是镜像集群模式。

镜像集群模式跟普通集群模式不一样的是，创建的 Queue，无论元数据还是Queue 里的消息都会存在于多个实例上，然后每次你写消息到 Queue 的时候，都会自动和多个实例的 Queue 进行消息同步。这样设计，好处在于：任何一个机器宕机不影响其他机器的使用。坏处在于：1. 性能开销太大：消息同步所有机器，导致网络带宽压力和消耗很重；2. 扩展性差：如果某个 Queue 负载很重，即便加机器，新增的机器也包含了这个 Queue 的所有数据，并没有办法线性扩展你的 Queue。

- **Kafka：partition 和 replica 机制**

Kafka 基本架构是多个 broker 组成，每个 broker 是一个节点。创建一个 topic 可以划分为多个 partition，每个 partition 可以存在于不同的 broker 上，每个 partition 就放一部分数据，这就是天然的分布式消息队列。就是说一个 topic 的数据，是分散放在多个机器上的，每个机器就放一部分数据。

Kafka 0.8 以前，是没有 HA 机制的，任何一个 broker 宕机了，它的 partition 就没法写也没法读了，没有什么高可用性可言。

Kafka 0.8 以后，提供了 HA 机制，就是 replica 副本机制。每个 partition 的数据都会同步到其他机器上，形成自己的多个 replica 副本。然后所有 replica 会选举一个 leader 出来，生产和消费都跟这个 leader 打交道，然后其他 replica 就是 follower。写的时候，leader 会负责把数据同步到所有 follower 上去，读的时候就直接读 leader 上数据即可。Kafka 会均匀的将一个 partition 的所有 replica 分布在不同的机器上，这样才可以提高容错性。

4、如何保证消息不被重复消费？或者说，如何保证消息消费的幂等性?

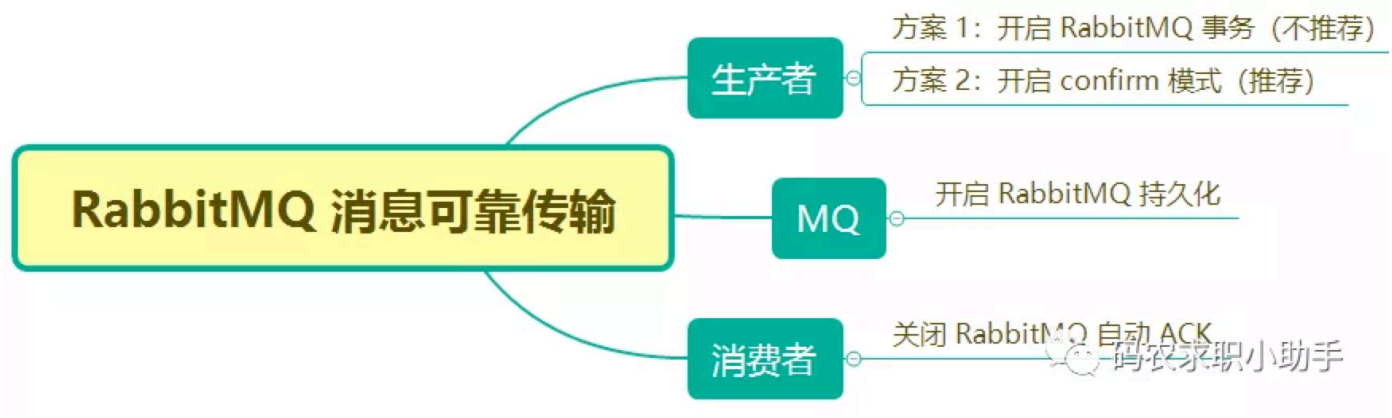
要保证消息不被重复消费，其实就是要保证消息消费时的幂等性。幂等性：无论你重复请求多少次，得到的结果都是一样的。例如：一条数据重复出现两次，数据库里就只有一条数据，这就保证了系统的幂等性。

那么如何保证幂等性呢?

1. 写数据时，先根据主键查一下这条数据是否存在，如果已经存在则 update；
2. 数据库的唯一键约束也可以保证不会重复插入多条，因为重复插入多条只会报错，不会导致数据库中出现脏数据；
3. 如果是写 redis，就没有问题，因为 set 操作是天然幂等性的。

5、如何保证消息的可靠性传输？或者说，如何处理消息丢失的问题?

- **RabbitMQ**



- **Kafka**

6、如何保证消息的顺序性?

- **RabbitMQ**

拆分多个 Queue，每个 Queue一个 Consumer，就是多一些 Queue 而已，确实是麻烦点；或者就一个 Queue 但是对应一个 Consumer，然后这个 Consumer 内部用内存队列做排队，然后分发给底层不同的 Worker 来处理。

- **Kafka**

1. 一个 Topic，一个 Partition，一个 Consumer，内部单线程消费，单线程吞吐量太低，一般不会用这个。
2. 写 N 个内存 Queue，具有相同 key 的数据都到同一个内存 Queue；然后对于 N 个线程，每个线程分别消费一个内存 Queue 即可，这样就能保证顺序性。

7、大量消息在 MQ 里长时间积压，该如何解决?

具体可以看这里：https://www.jianshu.com/p/5f4b3a520719

一般这个时候，只能临时紧急扩容了，具体操作步骤和思路如下：

1. 先修复 consumer 的问题，确保其恢复消费速度，然后将现有 consumer 都停掉；
2. 新建一个 topic，partition 是原来的 10 倍，临时建立好原先 10 倍的 queue 数量；
3. 然后写一个临时的分发数据的 consumer 程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的 10 倍数量的 queue；
4. 接着临时征用 10 倍的机器来部署 consumer，每一批 consumer 消费一个临时 queue 的数据。这种做法相

当于是临时将 queue 资源和 consumer 资源扩大 10 倍，以正常的 10 倍速度来消费数据；
5. 等快速消费完积压数据之后，得恢复原先部署的架构，重新用原先的 consumer 机器来消费消息。

8、MQ 中的消息过期失效了怎么办?

具体可以看这里：https://www.jianshu.com/p/5f4b3a520719

假设你用的是 RabbitMQ，RabbtiMQ 是可以设置过期时间的，也就是 TTL。如果消息在 Queue 中积压超过一定的时间就会被 RabbitMQ 给清理掉，这个数据就没了。这时的问题就不是数据会大量积压在 MQ 里，而是大量的数据会直接搞丢。这个情况下，就不是说要增加 Consumer 消费积压的消息，因为实际上没啥积压，而是丢了大量的消息。

我们可以采取一个方案，就是批量重导。就是大量积压的时候，直接丢弃数据了，然后等过了高峰期以后开始写程序，将丢失的那批数据一点一点的查出来，然后重新灌入 MQ 里面去，把丢的数据给补回来。

9、RabbitMQ 有哪些重要的角色?

RabbitMQ 中重要的角色有：生产者、消费者和代理。

1. 生产者：消息的创建者，负责创建和推送数据到消息服务器；
2. 消费者：消息的接收方，用于处理数据和确认消息；
3. 代理：就是 RabbitMQ 本身，用于扮演“快递”的角色，本身不生产消息，只是扮演“快递”的角色。

10、RabbitMQ 有哪些重要的组件?

1. ConnectionFactory（连接管理器）：应用程序与 rabbit 之间建立连接的管理器，程序代码中使用；
2. Channel（信道）：消息推送使用的通道；
3. Exchange（交换器）：用于接受、分配消息；
4. Queue（队列）：用于存储生产者的消息；
5. RoutingKey（路由键）：用于把生成者的数据分配到交换器上；
6. BindingKey（绑定键）：用于把交换器的消息绑定到队列上。

11、RabbitMQ 有几种广播类型?

RabbitMQ 有三种广播模式：fanout、direct、topic。

1. fanout：所有 bind 到此 exchange 的 queue 都可以接收消息；很像子网广播，每台子网内的主机都获得了一份复制的消息。fanout 交换机转发消息是最快的。
2. direct：通过 routingKey 和 exchange 中的 bindingKey 决定的那个唯一的 queue 可以接收消息；
3. topic：所有符合 routingKey 所 bind 的 queue 可以接收消息。

12、Kafka 可以脱离 zookeeper 单独使用吗？为什么?

Kafka 不能脱离 zookeeper 单独使用，因为 Kafka 使用 zookeeper 管理和协调 Kafka 的节点服务器。

13、Kafka 有几种数据保留的策略?

Kafka 有两种数据保存策略：按照过期时间保留和按照存储的消息大小保留。

14、Kafka 的分区策略有哪些?

所谓分区策略就是决定生产者将消息发送到哪个分区的算法。

1. 轮询策略：默认的分区策略，非常优秀的负载均衡表现，它总是能保证消息最大限度地被平均分配到所有分区上；
2. 随机策略：实现随机策略版的 partition 方法；
3. 按消息键保序策略：也称 Key-Ordering 策略，可以保证同一个 Key 的所有消息都进入到相同的分区里，由于每个分区下的消息处理是有顺序的，所以称之为消息键保序策略；
4. 自定义分区策略：在编写生产者程序时，你可以编写一个具体的类实现 org.apache.kafka.clients.producer.Partitioner 接口。这个接口也很简单，只定义了两个方法：partition() 和 close()，通常只用实现 partition() 方法即可。同时还需要设置 partitioner.class 参数为你自己实现类的全限定类名

Zookeeper

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**Java面试题**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**Java面试题**」，即可获取最新版 PDF。

1、谈下你对 Zookeeper 的认识?

ZooKeeper 是一个分布式的，开放源码的分布式应用程序协调服务。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等。

ZooKeeper 的目标就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

2、Zookeeper 都有哪些功能?

1. 集群管理：监控节点存活状态、运行请求等；

2. 主节点选举：主节点挂掉了之后可以从备用的节点开始新一轮选主，主节点选举说的就是这个选举的过程，使用 Zookeeper 可以协助完成这个过程；

3. 分布式锁：Zookeeper 提供两种锁：独占锁、共享锁。独占锁即一次只能有一个线程使用资源，共享锁是读锁共享，读写互斥，即可以有多线线程同时读同一个资源，如果要使用写锁也只能有一个线程使用。Zookeeper 可以对分布式锁进行控制。

4. 命名服务：在分布式系统中，通过使用命名服务，客户端应用能够根据指定名字来获取资源或服务的地址，提供者等信息。

3、谈下你对 ZAB 协议的了解?

ZAB 协议是为分布式协调服务 Zookeeper 专门设计的一种支持崩溃恢复的原子广播协议。ZAB 协议包括两种基本的模式：崩溃恢复和消息广播。

当整个 Zookeeper 集群刚刚启动或者Leader服务器宕机、重启或者网络故障导致不存在过半的服务器与 Leader 服务器保持正常通信时，所有服务器进入崩溃恢复模式，首先选举产生新的 Leader 服务器，然后集群中 Follower 服务器开始与新的 Leader 服务器进行数据同步。当集群中超过半数机器与该 Leader 服务器完成数据同步之后，退出恢复模式进入消息广播模式，Leader 服务器开始接收客户端的事务请求生成事物提案来进行事务请求处理。

4、Zookeeper 怎么保证主从节点的状态同步?

Zookeeper 的核心是原子广播机制，这个机制保证了各个 server 之间的同步。实现这个机制的协议叫做 Zab 协议。Zab 协议有两种模式，它们分别是恢复模式和广播模式。

- **1. 恢复模式**

当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数 server 完成了和 leader 的状态同步以后，恢复模式就结束了。状态同步保证了 leader 和 server 具有相同的系统状态。

- **2. 广播模式**

一旦 leader 已经和多数的 follower 进行了状态同步后，它就可以开始广播消息了，即进入广播状态。这时候当一个 server 加入 ZooKeeper 服务中，它会在恢复模式下启动，发现 leader，并和 leader 进行状态同步。待到同步结束，它也参与消息广播。ZooKeeper 服务一直维持在 Broadcast 状态，直到 leader 崩溃了或者 leader 失去了大部分的 followers 支持。

5、Zookeeper 有几种部署模式?

Zookeeper 有三种部署模式：

1. 单机部署：一台集群上运行；
2. 集群部署：多台集群运行；
3. 伪集群部署：一台集群启动多个 Zookeeper 实例运行。

6、说一下 Zookeeper 的通知机制?

client 端会对某个 znode 建立一个 watcher 事件，当该 znode 发生变化时，这些 client 会收到 zk 的通知，然后 client 可以根据 znode 变化来做出业务上的改变等。

7、集群中为什么要有主节点?

在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，其他的机器可以共享这个结果，这样可以大大减少重复计算，提高性能，于是就需要进行 leader 选举。

8、集群中有 3 台服务器，其中一个节点宕机，这个时候 Zookeeper 还可以使用吗?

可以继续使用，单数服务器只要没超过一半的服务器宕机就可以继续使用。

集群规则为 2N+1 台，N >0，即最少需要 3 台。

9、说一下两阶段提交和三阶段提交的过程？分别有什么问题?

- **两阶段提交协议 2PC**

1. 第一阶段（投票阶段）：

（1）协调者节点向所有参与者节点询问是否可以执行提交操作(vote)，并开始等待各参与者节点的响应；

（2）参与者节点执行询问发起为止的所有事务操作，并将Undo信息和Redo信息写入日志。

（3）各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实际执行成功，则它返回一个”同意”消息；如果参与者节点的事务操作实际执行失败，则它返回一个”中止”消息。

2. 第二阶段（提交执行阶段）：

当协调者节点从所有参与者节点获得的相应消息都为”同意”时：

（1）协调者节点向所有参与者节点发出”正式提交(commit)”的请求；

（2）参与者节点正式完成操作，并释放在整个事务期间内占用的资源；

（3）参与者节点向协调者节点发送”完成”消息；

（4）协调者节点受到所有参与者节点反馈的”完成”消息后，完成事务。

两阶段提交存在的问题：

1. 执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态；
2. 参与者发生故障：协调者需要给每个参与者额外指定超时机制，超时后整个事务失败；
3. 协调者发生故障：参与者会一直阻塞下去。需要额外的备机进行容错；
4. 二阶段无法解决的问题：协调者再发出 commit 消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

- **三阶段提交协议 3PC**

与两阶段提交不同的是，三阶段提交有两个改动点：

1. 引入超时机制。同时在协调者和参与者中都引入超时机制；
2. 在第一阶段和第二阶段中插入一个准备阶段。保证了在最后提交阶段之前各参与节点的状态是一致的。

也就是说，除了引入超时机制之外，3PC 把 2PC 的准备阶段再次一分为二，这样三阶段提交就有 CanCommit、PreCommit、DoCommit 三个阶段。

1. CanCommit 阶段

3PC 的 CanCommit 阶段其实和 2PC 的准备阶段很像。协调者向参与者发送 commit 请求，参与者如果可以提交就返回 Yes 响应，否则返回 No 响应。

（1）事务询问：协调者向参与者发送 CanCommit 请求。询问是否可以执行事务提交操作。然后开始等待参与者的响应。

（2）响应反馈：参与者接到 CanCommit 请求之后，正常情况下，如果其自身认为可以顺利执行事务，则返回 Yes 响应，并进入预备状态。否则反馈 No。

2. PreCommit 阶段

协调者根据参与者的反应情况来决定是否可以继续事务的 PreCommit 操作。根据响应情况，有以下两种可能：

假如协调者从所有的参与者获得的反馈都是 Yes 响应，那么就会执行事务的预执行。

（1）发送预提交请求：协调者向参与者发送 PreCommit 请求，并进入 Prepared 阶段。

（2）事务预提交：参与者接收到 PreCommit 请求后，会执行事务操作，并将 undo 和 redo 信息记录到事务日志中。

（3）响应反馈：如果参与者成功的执行了事务操作，则返回 ACK 响应，同时开始等待最终指令。

假如有任何一个参与者向协调者发送了 No 响应，或者等待超时之后，协调者都没有接到参与者的响应，那么就执行事务的中断。

（1）发送中断请求：协调者向所有参与者发送 abort 请求。

（2）中断事务：参与者收到来自协调者的 abort 请求之后（或超时之后，仍未收到协调者的请求），执行事务的中断。

3. doCommit 阶段

该阶段进行真正的事务提交，也可以分为以下两种情况。

3.1 执行提交

（1）发送提交请求：协调接收到参与者发送的 ACK 响应，那么他将从预提交状态进入到提交状态。并向所有参与者发送 doCommit 请求。

（2）事务提交：参与者接收到 doCommit 请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。

（3）响应反馈：事务提交完之后，向协调者发送 ACK 响应。

（4）完成事务：协调者接收到所有参与者的 ACK 响应之后，完成事务。

3.2 中断事务

协调者没有接收到参与者发送的 ACK 响应（可能是接受者发送的不是 ACK 响应，也可能响应超时），那么就会执行中断事务。

（1）发送中断请求：协调者向所有参与者发送 abort 请求。

（2）事务回滚：参与者接收到 abort 请求之后，利用其在阶段二记录的 undo 信息来执行事务的回滚操作，并在完成回滚之后释放所有的事务资源。

（3）反馈结果：参与者完成事务回滚之后，向协调者发送 ACK 消息。

（4）中断事务：协调者接收到参与者反馈的 ACK 消息之后，执行事务的中断。

三阶段提交的问题：

网络分区可能会带来问题。需要四阶段解决：四阶段直接调用远程服务的数据状态，确定当前数据一致性的情况。

10、Zookeeper 宕机如何处理?

Zookeeper 本身也是集群，推荐配置不少于 3 个服务器。Zookeeper 自身也要保证当一个节点宕机时，其他节点会继续提供服务。如果是一个 Follower 宕机，还有 2 台服务器提供访问，因为 Zookeeper 上的数据是有多个副本的，数据并不会丢失；如果是一个 Leader 宕机，Zookeeper 会选举出新的 Leader。

Zookeeper 集群的机制是只要超过半数的节点正常，集群就能正常提供服务。只有在 Zookeeper 节点挂得太多，只剩一半或不到一半节点能工作，集群才失效。所以：

3 个节点的 cluster 可以挂掉 1 个节点(leader 可以得到 2 票 > 1.5)

2 个节点的 cluster 就不能挂掉任何1个节点了(leader 可以得到 1 票 <= 1)

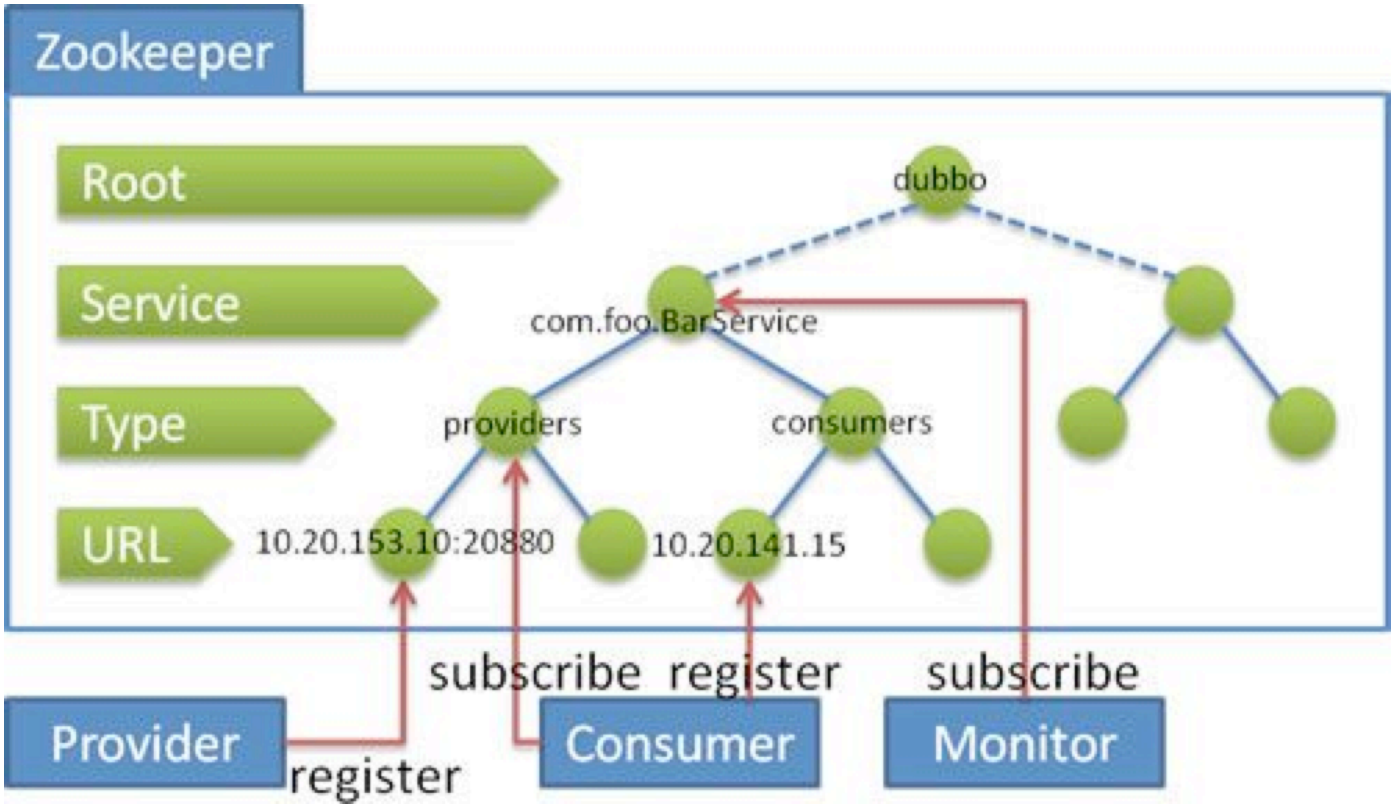
11. 说下四种类型的数据节点 Znode?

1. PERSISTENT：持久节点，除非手动删除，否则节点一直存在于 Zookeeper 上。
2. EPHEMERAL：临时节点，临时节点的生命周期与客户端会话绑定，一旦客户端会话失效（客户端与 Zookeeper连接断开不一定会话失效），那么这个客户端创建的所有临时节点都会被移除。
3. PERSISTENT_SEQUENTIAL：持久顺序节点，基本特性同持久节点，只是增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。
4. EPHEMERAL_SEQUENTIAL：临时顺序节点，基本特性同临时节点，增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

12、Zookeeper 和 Dubbo 的关系?

Dubbo 的将注册中心进行抽象，是得它可以外接不同的存储媒介给注册中心提供服务，有 ZooKeeper，Memcached，Redis 等。

引入了 ZooKeeper 作为存储媒介，也就把 ZooKeeper 的特性引进来。首先是负载均衡，单注册中心的承载能力是有限的，在流量达到一定程度的时 候就需要分流，负载均衡就是为了分流而存在的，一个 ZooKeeper 群配合相应的 Web 应用就可以很容易达到负载均衡；资源同步，单单有负载均衡还不 够，节点之间的数据和资源需要同步，ZooKeeper 集群就天然具备有这样的功能；命名服务，将树状结构用于维护全局的服务地址列表，服务提供者在启动 的时候，向 ZooKeeper 上的指定节点 /dubbo/${serviceName}/providers 目录下写入自己的 URL 地址，这个操作就完成了服务的发布。 其他特性还有 Mast 选举，分布式锁等。


Zookeeper
Root
Service
Type
URL
dubbo
com.foo.BarService
providers
consumers
10.20.153.10:20880
10.20.141.15
subscribe
register
subscribe
Provider
register
Consumer
Monitor

d