

JSD阶段模拟面试题

1. 第一教学月

1.1 编程基础

1.1.1 说说JDK、JRE、JVM的区别？

JDK (Java Development Kit)、JRE (Java Runtime Environment) 和 JVM (Java Virtual Machine) 是 Java 开发和运行环境的三个重要组成部分。

- JDK 是 Java 开发工具包，它包含了编译器 (javac)、调试器 (jdb) 以及其他用于开发和调试 Java 程序的工具。JDK 是开发人员在开发 Java 应用程序时所需的基本工具。
- JRE 是 Java 运行环境，它包含了用于执行 Java 程序的运行时库 (Java API) 以及 Java 虚拟机 (JVM)。JRE 只用于运行 Java 应用程序，而不包含开发工具。
- JVM 是 Java 虚拟机，它是一种能够执行 Java 字节码的虚拟机。JVM 是将 Java 代码翻译成机器码并在计算机上执行的关键组件。它负责加载字节码，执行字节码并进行垃圾回收等任务。JVM 是 Java 跨平台特性的基础，它使得 Java 程序能够在不同的操作系统和硬件上运行。

简单来说，JDK 包含了开发和调试 Java 程序所需的工具，JRE 是用于运行 Java 程序的环境，而 JVM 负责实际的程序执行。

1.1.2 java中的基本数据类型有哪些？

在 Java 中，有以下基本数据类型：

- byte:字节型，用于存储整数的，占用1个字节，范围-128到127
- short:短整型，用于存储整数的，占用2个字节，范围-32768到32767
- int:最常用的整型，用于存储整数的，占用4个字节，范围-2³¹到2³¹-1
- long:长整型，用于存储较大的整数，占用8个字节，范围-2⁶³到2⁶³-1
- float:单精度浮点数，用于存储小数的，占用4个字节，不能表示精确的值

- double:双精度浮点数，最常用的存储小数的类型，占用8个字节，不能表示精确的值
- boolean:布尔型，用于存储true或false，占用1个字节
- char:字符型，采用Unicode字符编码格式，用于存储单个字符，占用2个字节



这些基本数据类型在 Java 中具有固定的大小和范围，并且它们是不可变的。它们可用于声明变量、方法参数和方法返回值。

1.1.3 switch可应用于哪些数据类型上？

在Java中，switch语句是一种条件控制语句，用于根据不同的条件执行不同的代码块。switch语句使用一个或多个case标签来匹配某个表达式的值，并根据匹配结果执行相应的代码。switch语句的基本语法如下：

```
1 switch (expression) {  
2     case value1:  
3         // 执行代码块1break;  
4     case value2:  
5         // 执行代码块2break;  
6     case value3:  
7         // 执行代码块3break;  
8     // 可以有多个casedefault:  
9         // 执行默认代码块break;  
10 }
```

switch语句的执行流程如下：

1. 表达式会被计算出一个值。
2. switch语句会逐个比较case标签与表达式的值，如果匹配成功，则执行对应的代码块。
3. 如果没有匹配的case标签，则执行default代码块（可选）。
4. 执行完匹配的代码块后，会通过break语句跳出switch语句，继续执行后续的代码。

需要注意的是：

- 表达式的类型可以是byte、short、char、int、枚举类型或String类型。
- case标签的值不能重复，且必须是常量表达式（即不能是变量）。
- 每个case标签后面需要使用break语句来结束该分支的执行，否则会继续执行下一个case分支。
- 可以使用default关键字来定义默认的代码块，当没有匹配的case标签时将执行该代码块。



总而言之，switch语句是一种根据不同条件执行不同代码块的条件控制语句。通过比较表达式的值和case标签，可以选择执行相应的代码块。

1.1.4 说说Java中的数组？

在Java中，数组是一种用于存储相同类型元素的容器。它是一个固定长度的对象，可以通过索引(下标)来访问和操作其中的元素。数组在使用前需要被声明，然后通过指定索引位置来存储和读取元素。

在Java中，数组有以下几个特点：

声明数组：使用以下语法来声明一个数组：

```
1 type[] arrayName;
```

其中，type表示数组中存储的元素类型，arrayName是数组的名称。

创建数组：创建数组需要使用new关键字，语法如下：

```
1 arrayName = new type[length];
```

其中，length表示数组的长度，即数组可以存储的元素个数。

访问数组元素：数组中的元素可以通过索引来访问，索引从0开始，最大索引值为数组长度减1。例如：

```
1 int[] numbers = new int[5];  
2 numbers[0] = 10; // 给第一个元素赋值  
   int x = numbers[2]; // 读取第三个元素的值
```

数组的长度：可以使用数组的length属性来获取数组的长度，即存储的元素个数。例如：

```
1 int size = numbers.length; // 获取数组的长度
```

遍历数组：可以使用循环结构（如for循环）来遍历数组的所有元素。例如：

```
1 for (int i = 0; i < numbers.length; i++) {
2     System.out.println(numbers[i]);
3 }
```



需要注意的是，数组在创建后，长度是固定的，无法改变。如果需要动态调整长度或者存储不同类型的元素，可以使用集合类（如ArrayList）代替数组。

总结起来，数组是Java中用于存储相同类型元素的容器。通过索引可以访问和操作数组中的元素。使用数组可以方便地组织和处理大量相同类型的数据。

1.1.5 说说Java中的多维数组？

在Java中，多维数组是一种由多个一维数组组成的数据结构。它可以表示为一个表格或矩阵，具有行和列的概念。在多维数组中，每个元素都可以通过一组索引来访问。

Java中的多维数组可以是二维、三维，甚至更高维度的数组。声明和创建多维数组的语法如下：

```
1 type[][] arrayName; // 二维数组
2 type[][][] arrayName; // 三维数组// 更多维度的数组类似，以此类推
```

其中，type表示数组中存储的元素类型，arrayName是数组的名称。

创建多维数组需要为每个维度指定长度，例如：

```
1 int[][] matrix = new int[3][4]; // 创建一个3行4列的二维数组
```

访问多维数组中的元素需要使用多个索引来指定位置，例如：

```
1 int value = matrix[1][2]; // 访问第2行第3列的元素
```

可以使用嵌套的循环来遍历多维数组的所有元素，例如：

```
1 for (int i = 0; i < matrix.length; i++) {
2     for (int j = 0; j < matrix[i].length; j++) {
3         System.out.println(matrix[i][j]);
4     }
5 }
```

```
4     }  
5 }
```



需要注意的是，多维数组实质上是数组的数组，每个维度可以有不同的长度。每个维度的长度必须在创建数组时指定，并且每个维度的索引范围都是从0到长度减1。

总结起来，多维数组是由多个一维数组组成的数据结构，用于表示表格、矩阵等具有多行多列的数据。通过多个索引可以访问和操作多维数组中的元素。多维数组的长度在创建时需要指定，并且每个维度的索引范围从0到长度减1。

1.2 面向对象设计

1.2.1 描述一下封装以及优势和劣势？

Java中的封装（Encapsulation）是面向对象编程的核心特性之一，它允许将数据和方法包装在一个类中，并对外隐藏内部细节。

封装的定义和特点：

- 将数据（属性）和操作数据的方法（行为）封装在一个类中
- 通过接口方式和访问修饰符来控制对类内部成员的访问，隐藏内部实现细节。

封装的优势：

- 数据隐藏：封装允许将数据隐藏在类的内部，通过定义私有（private）属性，可以避免直接访问和修改对象的数据，提高数据的安全性和可靠性。
- 隔离复杂性：封装将类的实现细节隐藏起来，只暴露必要的接口给外部使用者，减少了对外部的依赖，降低了类与类之间的复杂性。
- 提供公共接口：通过封装，可以定义公共的接口（public方法），对外提供统一的访问和操作方式，提供更好的交互方法，便于使用和维护。

封装的劣势：

可见性控制的复杂性：对类内部成员的可见性需要仔细控制，使用不当可能导致对属性的过度保护或过度开放。



总结起来，封装是面向对象编程的重要特性，它提供了数据隐藏和公共接口的优势，可以降低代码的复杂性，提高代码的可维护性和安全性。然而，封装也可能增加代码量和访问的间接性等劣势，需要权衡使用。

1.2.2 描述一下继承以及优势和劣势？

Java中的继承是面向对象编程的核心特性之一，它允许一个类继承另一个类。然后使用父类中的属性和方法。

继承的定义和特点：

- 继承是指一个类型（子类/派生类）继承另一个类类型（父类/基类/超类）。
- 子类通过关键字 `extends` 来继承父类，子类可以直接使用父类的非私有属性和方法。
- 子类可以覆盖（重写）父类的方法，从而实现自己的行为。

继承的优势：

- 代码复用：继承允许子类直接使用父类的属性和方法，避免了重复编写相同的代码，提高了代码复用性。
- 扩展性：子类可以在继承父类的基础上进行修改和扩展，添加新的属性和方法，满足不同的需求。
- 继承建立了对象之间的关系，通过继承，可以形成类的体系结构，使得程序逻辑更加清晰和有组织性。

继承的劣势：

- 继承层次过深复杂：如果继承结构过于复杂，层次过深，会增加类之间的关系和依赖，理解和维护代码可能会更加困难。
- 导致类之间的紧耦合：子类继承了父类的属性和方法，子类与父类之间存在较强的耦合关系，当父类需要进行修改时，可能会影响到所有的子类。
- 单一继承限制：Java只支持单一继承，一个类只能继承一个父类，无法同时继承多个类，这在某些情况下会限制程序的设计。

为了克服继承的劣势，可以采用其他面向对象编程机制，如接口（Interface）、组合等来替代继承。接口可以定义一组方法的规范，类可以实现多个接口，而不受单一继承的限制。组合可以将多个类组合在一起，以实现代码的复用和扩展。



总结起来，继承作为面向对象编程的重要概念，具有代码复用、扩展性和类的组织性等优势，但也存在复杂性、紧耦合和单一继承限制等劣势。在设计和选择继承结构时，需要综合考虑这些优劣势，选择适合的设计方案。

1.2.3 描述一下Java中的多态以及优势和劣势？

Java中的多态（Polymorphism）是面向对象编程的核心特性之一，它描述的是一种行为因为对象不同，执行结果则不相同，例如生活中的睡觉这种行为，发生在不同对象身上时，结果是不相同的，有人睡觉时磨牙，有人睡觉时打呼噜，还有梦游的。程序中的多态是指同一个类型的变量引用不同类型的对象，然后在运行时根据对象的实际类型来动态执行相应的方法。

Java中实现多态的条件：

1. 存在继承关系，即子类继承父类。
2. 子类重写父类的方法，即子类覆盖了父类的方法。
3. 使用父类类型的引用变量来引用子类对象。

多态的优势：

1. 代码的灵活性：通过多态，可以使用父类类型的引用变量来引用不同子类类型的对象。这样一来，我们可以通过统一的接口来操作不同的对象，从而提高代码的灵活性和可维护性。当需求变化时，我们只需要更换具体的子类对象，而无需修改现有的代码。
2. 代码的可扩展性：当面对需要添加新的功能时，我们可以通过创建新的子类来扩展现有的类体系，并在子类中重写需要改变的方法。这样一来，我们可以在不修改现有代码的情况下，灵活地添加新的功能。
3. 可以实现接口的透明性：接口是多态的重要应用之一。通过面向接口编程，我们可以将对象的实际类型隐藏起来，只关注接口定义的方法。这样一来，我们可以轻松地切换不同的实现类，而不需修改引用变量的类型。
4. 可以实现方法的回调：通过多态，我们可以将一个对象作为参数传递给其他方法，从而实现方法的回调。这在事件处理、回调函数等场景中非常实用，可以让程序在运行时决定具体调用哪个方法。
5. 提高代码的可读性和可维护性：多态性可以让代码更加简洁和清晰，不需要针对每个子类编写特定的代码。通过面向接口或父类编程，代码更易读懂，并且更容易被其他开发人员理解和维护。

多态的劣势：

1. 简单功能的过度抽象化：使用多态可能导致有些功能被过度抽象，无法直接调用子类特定的方法，限制了某些具体功能的使用。
2. 运行时开销：在运行时，需要通过动态绑定和虚函数表等机制来确定方法的调用。这些额外的运行时开销可能会对性能产生一定的影响。
3. 引入复杂性：多态的使用可能引入复杂性，对于初学者来说，尤其是在处理继承关系复杂的情况下，可能会增加代码阅读和理解的困难。



总结起来，多态是面向对象编程的重要特性，它通过统一处理不同类型的对象，提高了代码的灵活性、可替换性和可扩展性。然而，多态也可能导致抽象过度、带来运行时开销和引入复杂性等劣势，需要谨慎使用。

1.2.4 重写与重载的区别

重写:发生在父子类中，方法名相同，参数列表相同

重载:发生在同一类中，方法名相同，参数列表不同

1.2.5 实例变量与静态变量的区别

实例变量:是属于对象的，在创建对象时存储在内存堆中，创建多少个对象，则实例变量就会在内存中存在多少份，需要通过引用变量来访问。

静态变量:是属于类的，在类被加载时存储在内存方法区中，无论创建多少个对象，静态变量在内存中都只有一份，常常通过类名点来访问。

1.2.6 描述一下抽象类的特点？

- 由abstract修饰
- 可以包含变量、常量、构造方法、普通方法、静态方法、抽象方法
- 派生类通过extends继承
- 只能继承一个(单一继承)
- 抽象类中的成员，任何访问权限都可以(默认为默认权限(同包中))

1.2.7 描述一下接口的特点？

- 由interface定义
- 可以包含常量、抽象方法、静态方法(1.8后)、默认方法(1.8后)、私有方法(1.9后)
- 实现类通过implements实现
- 可以实现多个(体现多实现)
- 接口中的成员，访问权限只能是public(默认public权限)

1.2.8 说说java中的值传递含义？

- 对于基本类型而言，传递的是具体的值的副本
- 对于引用类型而言，传递的是具体的地址的副本

1.2.9 ==和equals方法的含义？

== 比较的是变量(栈)内存中存放的对象的(堆)内存地址，用来判断两个对象的地址是否相同，即是否是指相同一个对象。比较的是真正意义上的指针操作。

- 1、比较的是操作符两端的操作数是否是同一个对象。
- 2、两边的操作数必须是同一类型的（可以是父子类之间）才能编译通过。
- 3、比较的是地址，如果是具体的阿拉伯数字的比较，值相等则为 true，如：int a=10 与 long b=10L 与 double c=10.0都是相同的（为true），因为他们都指向地址为10的堆。

equals：equals用来比较的是两个对象的内容是否相等，由于所有的类都是继承自java.lang.Object类的，所以适用于所有对象，如果没有对该方法进行覆盖的话，调用的仍然是Object类中的方法，而Object中的equals方法返回的却是==的判断。



总结：所有比较是否相等时，都是用equals 并且在对常量相比较时，把常量写在前面，因为使用object的 equals object可能为null 则空指针 在阿里的代码规范中只使用equals，阿里插件默认会识别，并可以快速修改，推荐安装阿里插件来 排查老代码使用“==”，替换成equals。

1.2.10 说说Java中的this关键字？

在Java中，this是一个特殊的关键字，它代表当前对象（实例）。this关键字可以用于以下几个方面：

1. 引用当前对象：

- 在一个实例方法中，this关键字用于引用当前调用该方法的对象。
- 当需要访问当前对象的成员变量或调用当前对象的其他方法时，可以使用this关键字来引用。

2. 解决命名冲突：

- 当方法内部存在与成员变量同名的局部变量时，使用this关键字可以解决命名冲突。
- this关键字指明了要访问的是当前对象的成员变量，而不是局部变量。

3. 在构造方法中引用其他构造方法：

- 当一个类中定义了多个构造方法时，可以使用this关键字来引用其他构造方法。
- 当需要在一个构造方法中调用其他构造方法时，可以使用this关键字并传递相应的参数。



需要注意的是，this关键字只能在实例方法和构造方法中使用，不能在静态方法中使用。因为静态方法不依赖于具体的对象实例，所以在静态方法中无法引用this关键字。

总而言之，this关键字用于引用当前对象，解决命名冲突，以及在构造方法中引用其他构造方法。通过this关键字，我们可以更方便地访问当前对象的成员变量和调用当前对象的方法。

1.2.11 说说Java中的super关键字？

在Java中，super是一个特殊的关键字，用于引用父类（超类）的成员或调用父类的构造方法。super关键字可以用于以下几个方面：

1. 引用父类的成员：

- 当子类和父类中有同名的成员变量或方法时，可以使用super关键字来引用父类的成员。
- 通过super关键字，可以明确指定要访问父类的成员，而不是子类的成员。

2. 调用父类的构造方法：

- 当子类继承父类时，可以使用super关键字来调用父类的构造方法。
- 在子类的构造方法中使用super关键字调用父类的构造方法，可以完成对父类的初始化操作。

✨ 总之，super关键字在子类中用于引用父类的成员或调用父类的构造方法。通过super关键字，可以方便地访问父类的成员和完成对父类的初始化操作。

1.2.12 说说static关键字的作用？

在Java中，static关键字有以下几种用法：

1. **修饰变量：**使用static关键字修饰变量，表示该变量为静态变量（类变量），属于类而不是实例。静态变量在程序启动时会被初始化，且所有该类的实例将共享同一个静态变量。可以通过类名直接访问静态变量。例如：`public static int count = 0;`
2. **修饰方法：**使用static关键字修饰方法，表示该方法为静态方法，属于类而不是实例。静态方法可以通过类名直接调用，无需创建实例。静态方法只能直接访问静态变量和调用静态方法。例如：`public static void printCount() { ... }`
3. **静态代码块：**使用static关键字定义静态代码块，在类加载时执行且只执行一次。静态代码块常用于进行类的初始化操作，如加载配置文件、初始化静态变量等。例如：`static { ... }`
4. **静态内部类：**使用static关键字定义静态内部类，静态内部类与外部类的实例无关，可以直接通过外部类名访问。静态内部类不能访问外部类的非静态成员，但可以访问外部类的静态成员。例如：`public static class InnerClass { ... }`



静态成员和静态方法可以在没有创建实例的情况下被访问和调用，因此它们与类关联而不是与特定实例关联。静态成员和静态方法可以用于实现单例模式、工具方法和共享数据等功能。然而，过度使用静态成员和静态方法可能会带来内存泄漏和扩展性问题，开发者应根据具体需求和设计考虑来决定是否使用static关键字。

1.2.13 说说Java中final关键字的作用？

在Java中，final关键字有以下几种用法：

1. 修饰变量：使用final关键字修饰变量，表示该变量的值一旦被初始化就不能再被修改。修饰的变量必须进行显式初始化，可以在声明时或构造方法中进行初始化。例如：`final int num = 10;`
2. 修饰方法：使用final关键字修饰方法，表示该方法不能被子类重写。这样可以确保方法的行为不会被子类改变。例如：`public final void method() { ... }`
3. 修饰类：使用final关键字修饰类，表示该类不能被继承。这样可以确保类的实现不会被修改和继承。例如：`public final class MyClass { ... }`
4. 修饰参数：在方法中，将参数使用final关键字修饰，表示该参数的值不能在方法内部被修改。这主要用于确保方法的参数在方法内部不会被错误地修改。例如：`public void method(final int num) { ... }`



需要注意的是，final关键字的使用可以增加代码的可读性和安全性，同时有助于编译器进行优化。然而，过度使用final关键字可能会影响代码的灵活性和可扩展性，开发者应根据具体的需求和设计考虑来决定是否使用final关键字。

1.2.14 说说Java中的内部类？

在Java中，内部类是指定义在其他类内部的类。根据内部类的定义位置和访问权限，Java中的内部类可分为以下几种类型：

1. **成员内部类**：成员内部类是定义在另一个类的内部的类。它可以访问外部类的成员变量和方法，包括私有成员。成员内部类的实例化需要先实例化外部类的对象才能访问，即成员内部类的对象与外部类对象是有关联的。
2. **静态内部类**：静态内部类是定义在另一个类内部并且使用static修饰的类。它与普通的静态成员类似，不依赖于外部类的实例化对象而存在。静态内部类可以访问外部类的静态成员，但不能访问外部类的非静态成员。

3. **局部内部类**：方法内部类是定义在方法内部的类。它的作用域仅限于该方法内部，只能在该方法内使用。方法内部类可以访问外部类的成员变量和方法，包括私有成员，但是只能访问方法中的 `final` 局部变量。
4. **匿名内部类**：匿名内部类是没有命名的内部类，它通常作为实现接口或继承父类的方式来创建一个对象。匿名内部类的定义和创建都在同一个语句中完成，使用时一般作为参数传递给方法或者作为局部变量使用。匿名内部类可以访问外部类的成员变量和方法，但需要注意外部类成员变量需要为 `final` 类型或 `effectively final` 才能被匿名内部类访问。



总之，内部类可以访问外部类的私有成员，而外部类无法直接访问内部类的成员，从而实现了更好的封装性。并可以将类关联属性和方法放在一起，提高代码的组织性和可读性。需要注意的是，内部类在使用时需要注意命名冲突和内存消耗问题。要根据具体需求合理使用，以提高代码的可维护性和可扩展性。

1.3 基础API应用分析

1.3.1 Object类常用方法及应用场景？

Object类是所有类的根类，在Java中，每个类都直接或间接继承自Object类。因此，Object类中的方法在Java中非常常用。下面列举了一些常用的Object类的方法以及它们的应用场景：

1. `equals(Object obj)`：用于比较两个对象是否相等。该方法默认比较两个对象的引用是否相等，但可以根据需要进行重写，实现自定义的相等判断。
2. `hashCode()`：返回对象的哈希码。可以用于自定义对象的散列算法，如将对象放入哈希表（HashMap）中进行查找。
3. `toString()`：返回对象的字符串表示。通常用于将对象转换为人可读的格式，便于输出和调试。建议在自定义类中重写该方法，以便更好地展示对象的信息。
4. `getClass()`：返回对象的运行时类。可以通过该方法获取对象所属的类的信息，例如获取类的名称、包名等。
5. `finalize()`：当垃圾回收器确定不再有对该对象的引用时，会调用该方法进行资源的释放。一般不推荐使用该方法，因为它的具体调用时间不确定。
6. `clone()`：创建并返回当前对象的一个拷贝（即克隆）。使用该方法，对象所属的类必须实现Cloneable接口。可以用于实现深拷贝，即复制对象及其引用。
7. `wait()`、`notify()`和`notifyAll()`：用于线程间的通信和同步。这些方法配合synchronized关键字使用，实现线程的等待和唤醒机制。

8. instanceof：判断对象是否属于某个类或其子类的实例。可以用于类型检查，以确保对象在进行类型转换之前是安全的。



这些方法都是Object类中定义的常用方法，可以在其他类中使用，或通过类的继承关系来进行重写，以满足自定义类的具体需求。

1.3.2 String、StringBuffer、StringBuilder有什么不同

String、StringBuffer和StringBuilder是在Java中用于处理字符串的三个类。它们之间的区别如下：

1. 可变性：

- String是不可变的类，即一旦创建了字符串对象，就不能修改其内容。对字符串进行拼接、插入或删除等操作会创建新的字符串对象。
- StringBuffer和StringBuilder是可变的类，它们支持对字符串进行动态修改。可以对字符串进行拼接、插入、删除和替换等操作，不会创建新的对象，而是在原有对象上进行修改。

2. 线程安全性：

- String是线程安全的，适用于多线程环境。这是因为String的不可变性，不会出现多个线程同时修改同一个字符串对象的情况。
- StringBuffer是线程安全的，它的方法使用了synchronized关键字进行同步，保证了多线程环境下的安全性。
- StringBuilder是非线程安全的，它的方法没有进行同步处理。在单线程环境下，StringBuilder的性能比StringBuffer更高。

3. 性能：

- String的不可变性给它带来了一些性能上的优化，比如字符串常量池的使用。但是在频繁的字符串拼接操作中，由于会创建大量的临时对象，性能会受到一定的影响。
- StringBuffer和StringBuilder的可变性使得它们在频繁的字符串操作中性能更好，特别是StringBuilder，因为它不再需要进行同步操作。



总结起来，如果需要对字符串进行频繁的操作和修改，并在多线程环境下使用，应选StringBuffer。如果只在单线程环境下进行字符串操作，并且性能要求较高，可以选择StringBuilder。如果字符串不需要修改，或者在多线程环境下使用，应选择不可变的String类。

1.3.3 如何理解自动拆箱封箱操作？

- 装箱就是自动将基本数据类型转换为包装器类型，例如Integer的 `valueOf(int)` 方法。
- 拆箱就是自动将包装器类型转换为基本数据类型，例如调用Integer的 `intValue` 方法。

1.3.4 说说ArrayList的特点及应用场景？

ArrayList是Java中常用的动态数组实现类，它继承了AbstractList抽象类，并实现了List、RandomAccess、Cloneable和Serializable接口。ArrayList具有以下几个特点和应用场景：

1. **动态数组**：ArrayList底层使用数组来实现，具有动态扩容的特性。在添加和删除元素时，ArrayList可以自动调整底层数组的大小，以适应变化的数据量。
2. **随机访问**：由于ArrayList内部使用数组实现，在获取元素时具有高效的随机访问能力。通过索引可以快速获取指定位置的元素。
3. **允许存储重复元素**：ArrayList中可以存储重复的元素。每个元素都可以通过索引访问和修改。
4. **非线程安全**：ArrayList是非线程安全的，不适用于多线程的并发操作。如果在多个线程中同时对ArrayList进行修改，可能导致数据不一致的问题。
5. **删除和插入效率较低**：由于ArrayList底层使用数组实现，删除和插入元素时需要移动其他元素，因此效率较低。如果需要频繁进行删除和插入操作，可以考虑使用LinkedList。
6. **应用场景**：ArrayList适合在需要快速访问元素、对元素进行频繁随机访问的场景。它常用于保存和处理大量数据的集合，例如在数据存储和操作、遍历数据等方面都可以使用ArrayList。

⚠ 需要注意的是，由于ArrayList内部使用数组实现，当数据量较大或需要频繁进行添加和删除操作时，可能会导致频繁的内存重新分配和数据复制，影响性能。在这些情况下，可以考虑使用其他更适合的数据结构来提升效率。

1.3.5 说说LinkedList的特点及应用场景？

LinkedList是Java中常用的双向链表实现类，它实现了List、Deque、Cloneable和Serializable接口。LinkedList具有以下几个特点和应用场景：

1. **双向链表**：LinkedList内部使用双向链表来存储元素，每个节点都包含对前后节点的引用，因此可以实现快速的插入和删除操作。
2. **高效的插入和删除**：由于LinkedList内部使用链表实现，插入和删除元素时只需要修改节点的引用，不需要像ArrayList那样进行元素的移动和复制。因此，在需要频繁进行插入和删除操作时，LinkedList具有较高的效率。
3. **支持添加、删除、获取等操作**：LinkedList提供了丰富的方法来支持元素的添加、删除、获取等操作。例如，`addFirst`、`addLast`、`removeFirst`、`removeLast`等方法可以在链表的头部和尾部进行快速的添加和删除操作。

4. 可用作队列和栈：由于LinkedList实现了Deque接口，它可以被当作队列和栈来使用。通过调用LinkedList提供的方法，可以实现先进先出的队列或后进先出的栈。
5. 非线性安全：和ArrayList一样，LinkedList也是非线性安全的，不适用于多线程的并发操作。
6. 应用场景：LinkedList适合在需要频繁进行插入和删除操作的场景。例如，当需要在集合的头部或尾部添加或删除元素时，或者需要高效地实现队列、栈等数据结构时，LinkedList是一个很好的选择。此外，当操作的数量较少且需要频繁地进行遍历或查找操作时，LinkedList可能不如ArrayList效率高。

需要注意的是，虽然LinkedList在插入和删除操作上具有较高的效率，但在随机访问和索引定位方面相对较低。如果需要频繁进行随机访问操作，或者需要通过索引快速获取元素时，建议使用ArrayList。

1.3.6 ArrayList和LinkedList的区别？

ArrayList和LinkedList是在Java中用于存储和操作集合的两种常见实现方式。它们之间的区别如下：

1. 底层数据结构：

- ArrayList是使用数组实现的，内部维护一个连续的内存空间。可以根据索引快速访问和修改元素。
- LinkedList是使用链表实现的，内部的元素通过节点相互连接。插入和删除操作效率较高，但访问和修改元素需要遍历整个链表。

2. 插入和删除操作：

- ArrayList在末尾进行插入和删除操作的效率较高，时间复杂度为 $O(1)$ 。但是在中间或开头进行插入和删除操作时，需要进行元素的位移，效率较低，时间复杂度为 $O(n)$ 。
- LinkedList在任意位置进行插入和删除操作的效率都很高，时间复杂度为 $O(1)$ 。这是因为它只需要修改节点的指针，而不需要移动元素。

3. 访问元素的效率：

- ArrayList可以通过索引直接访问元素，时间复杂度为 $O(1)$ 。适用于频繁访问和随机访问的场景。
- LinkedList需要从链表头或尾开始遍历，直到找到目标位置的元素。时间复杂度为 $O(n)$ ，适用于按顺序访问的场景。

4. 空间消耗：

- ArrayList在创建时会分配一块连续的内存空间，并根据实际需要进行动态扩容。如果元素数量较多，可能会浪费一些空间。
- LinkedList在存储元素时还需要额外的空间用于存储节点之间的链接关系。所以，相同数量的元素，LinkedList通常占用的空间更多。



根据不同的需求和操作场景，选择合适的集合类是很重要的。如果需要频繁进行插入和删除操作，或者按顺序遍历，选择LinkedList会更合适。如果需要频繁访问元素或根据索引进行操作，选择ArrayList会更合适。

1.3.7 说说Java中的泛型及应用？

泛型提供了编译时类型安全检测机制，该机制允许程序员在编译时检测到非法的类型。泛型的本质是参数化类型，它允许在类、接口、方法的定义中使用类型参数，使得我们可以在编译时期指定集合、类或方法的数据类型，以提高代码的安全性和可读性。

Java中的泛型主要有以下几个要点：

1. 泛型类和接口：泛型类或接口通过使用尖括号 `<>` 来指定类型参数。例如，`List<E>` 表示一个可以存放元素类型为 `E` 的列表。
2. 泛型方法：泛型方法是定义在普通类或接口中的方法，并在方法返回值类型前使用尖括号 `<>` 来指定类型参数。例如，`<E> void printList(List<E> list)` 是一个泛型方法，参数列表中的 `List<E>` 可以接收任意类型的 `List`。
3. 类型通配符：当我们需要在泛型中使用未知的类型时，可以使用通配符来表示。通配符有两种形式：`?` 和 `? extends Type`。例如，`List<?>` 表示一个未知类型的列表，而 `List<? extends Number>` 表示一个可以存放任意 `Number` 类型或其子类的列表。`List<? super Integer>` 表示一个可以存放 `Integer` 类型或其类型的父类类型。
4. 类型推断：Java 7 引入了类型推断，使得在使用泛型时可以省略类型参数的声明。例如，使用 `diamond` 语法可以简化创建泛型实例的代码：`List<String> list = new ArrayList<>();`



使用泛型的好处是可以提高代码的类型安全性，减少类型转换的错误，并且使代码更加通用和可读。通过使用泛型，可以在编译时期对代码进行更严格的类型检查，并提供更好的代码提示。

1.3.8 说说Java中的泛型类型擦除？


Java 中的泛型基本上都是在编译器这个层次来实现的。在生成的 Java 字节代码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，会被编译器在编译的时候去掉。这个过程就称为类型

擦除。如在代码中定义的 List 和 List 等类型，在编译之后 都会变成 List。JVM 看到的只是 List，而由泛型附加的类型信息对 JVM 来说是不可见的。类型擦除的基本过程也比较简单，首先是找到用来替换类型参数的具体类。这个具体类一般是 Object。如果指定了类型参数的上界的话，则使用这个上界。把代码中的类型参数都替换 成具体的类。

1.3.9 说说你对Java中泛型的理解？

Java中的泛型是一种在编译时期进行类型检查和类型约束的机制，它的特点如下：

1. 类型安全性：泛型可以在编译阶段检查类型的正确性，避免了在运行时发生类型转换错误的问题。通过使用泛型，可以在编写代码时就能够确定对象的类型，并且编译器会确保在使用对象时类型的一致性。
2. 代码重用性：泛型可以使代码更加通用和可重用，因为可以将类型参数化，使得同一段代码可以适用于不同类型的数据。例如，可以定义一个通用的集合类，可以存储不同类型的对象，提高代码的灵活性。
3. 可读性和安全性提高：有了泛型，代码更加清晰和易读，因为泛型可以提供更具描述性的类型信息。通过使用泛型，可以减少类型转换和强制类型检查的需要，使得代码可读性更强，同时也减少了出错的可能性。
4. 避免了强制类型转换：在使用非泛型的集合类时，需要进行强制类型转换以获取存储在集合中的对象。而泛型可以省略这一步骤，使得代码更加简洁和安全。

 需要注意的是，泛型只存在于编译阶段，在运行时会被擦除。这意味着在生成的字节码中，所有的泛型信息都会被擦除，泛型类或方法在运行时将没有泛型类型的信息。这也是为了保持与之前版本的Java的兼容性。

1.3.10 说说Java中创建对象的方式？

在Java中，我们可以使用以下几种方式来创建对象：

1. 使用new关键字：这是最常见的创建对象的方式，在堆内存中创建一个新的对象实例。例如：
`MyObject obj = new MyObject();`
2. 使用Class类的newInstance()方法：通过反射机制，我们可以使用Class类的newInstance()方法来创建一个类的实例。这种方式需要无参构造方法。例如：`MyObject obj = MyClass.class.newInstance();`

3. 使用Constructor类的newInstance()方法：同样是通过反射机制，我们可以使用Constructor类的newInstance()方法来创建一个类的实例。这种方式可以使用带参构造方法创建对象。例如：
`Constructor<MyClass> constructor = MyClass.class.getConstructor(String.class); MyClass obj = constructor.newInstance("parameter");`
4. 使用clone()方法：如果一个类实现了Cloneable接口，我们可以使用clone()方法来创建一个对象的副本。使用clone()方法创建对象时，实际上是调用了对象的clone()方法，并返回一个新的对象实例。例如：`MyObject obj2 = (MyObject) obj.clone();`
5. 使用反序列化：如果一个类实现了Serializable接口，我们可以使用反序列化的方式来创建对象。通过将对象转换成字节序列，然后再将字节序列转换回对象，来实现对象的创建。例如：
`ObjectInputStream in = new ObjectInputStream(new FileInputStream("file.ser")); MyObject obj = (MyObject) in.readObject();`



无论使用哪种方式创建对象，最终都会在内存中为对象分配空间，并调用构造方法来进行初始化。每种方式都有其适用的场景和注意事项，开发者应根据具体需求来选择适合的方式来创建对象。

1.3.11 Java中的IO是如何分类的？

根据流的流向，可以将流分为输入流（InputStream）和输出流（OutputStream）。输入流负责从外部读取数据到程序中，而输出流负责将程序中的数据输出到外部。

根据操作单元，可以将流分为字节流（Byte Stream）和字符流（Character Stream）。字节流按字节读写数据，常用于处理二进制数据；字符流按字符读写数据，常用于处理文本数据，内部会根据指定的字符编码进行转换。

根据流的角色，可以将流分为节点流（Node Stream）和处理流（Filter Stream）。节点流是直接与数据源或目标进行连接的流，例如 `FileInputStream` 和 `FileOutputStream` 是直接连接文件的节点流。处理流是通过包装其他流来提供额外功能的流，例如 `BufferedInputStream` 和 `BufferedOutputStream` 是增加了缓冲功能的处理流，它们可以提高读写的效率。



这种分类方式帮助我们理解和选择合适的流来满足不同的需求。注意，在实际使用中，我们可以结合多种流来进行组合，以实现更复杂的数据处理任务。

1.3.12 说说深拷贝和浅拷贝的不同？

深拷贝（Deep Copy）和浅拷贝（Shallow Copy）是在编程中用来复制对象的两种不同的方式。

浅拷贝是创建一个新对象，并将原始对象的引用复制到新对象中。这意味着新对象和原始对象将共享相同的内存地址，修改一个对象的属性会影响到另一个对象。

深拷贝是创建一个新对象，并将原始对象的所有属性的副本复制到新对象中。这意味着新对象和原始对象是完全独立的，修改一个对象的属性不会影响到另一个对象。

在深拷贝中，如果原始对象的属性是值类型（例如整数、浮点数等），则会直接复制它们的值。如果原始对象的属性是引用类型（例如列表、字典、自定义类等），则会递归地复制它们的内容，以确保完全独立的新对象。

在浅拷贝中，无论原始对象的属性是值类型还是引用类型，都只是复制它们的引用。这意味着新对象和原始对象之间共享相同的引用类型属性，修改一个对象的属性会影响到另一个对象。



总结来说，深拷贝创建了原始对象的完全独立副本，而浅拷贝创建了一个新对象，但某些属性仍然共享内存地址。选择深拷贝还是浅拷贝要根据具体需求和对象结构来决定。

1.3.13 请细说一下Java线程对象状态？

在Java中，线程对象可以处于不同的状态，这些状态反映了线程在不同阶段的执行情况。具体状态说明如下：

1. 新建状态（New）：

- 新建状态表示线程对象创建后尚未启动，即尚未调用start()方法。
- 在新建状态下，线程对象的一些初始化操作可以进行，如分配内存、初始化变量等。

2. 就绪状态（Runnable）：

- 就绪状态表示线程对象已经准备好执行，并且等待获取CPU时间片。
- 当线程对象被调度器选中，并分配到CPU时间片时，它会从就绪状态切换到运行状态。

3. 运行状态（Running）：

- 运行状态表示线程对象正在执行run()方法中的代码。
- 在运行状态下，线程会不断占用CPU资源，执行自己的任务。

4. 阻塞状态（Blocked）：

- 阻塞状态表示线程对象由于某些原因暂时无法执行，需要等待唤醒或满足特定条件。
- 当线程处于阻塞状态时，它会释放持有的锁，让其他线程有机会获取锁并执行。线程被唤醒后会从阻塞状态切换到就绪状态。

5. 等待状态 (Waiting) :

- 等待状态表示线程对象正在等待其他线程的特定操作，无法被唤醒。
- 线程进入等待状态的常见情况包括调用了wait()方法、join()方法、park()方法等。
- 当满足某个条件后，其他线程可以通知等待状态的线程，并将其唤醒。被唤醒后的线程会从等待状态切换到就绪状态。

6. 超时等待状态 (Timed Waiting) :

- 超时等待状态与等待状态类似，但是在等待一定时间后会自动返回。
- 线程进入超时等待状态的常见情况包括调用了带有超时参数的wait()方法、sleep()方法、join()方法等。
- 当时间到达或满足某个条件后，线程会从超时等待状态切换到就绪状态。

7. 终止状态 (Terminated) :

- 终止状态表示线程对象已经执行完run()方法，或者出现了异常而终止了执行。
- 在终止状态下，线程对象的执行已经结束，不再具有了执行能力。



需要注意的是，不同状态之间的转换是由Java虚拟机和操作系统来控制的，开发人员无法人为控制和干预线程对象的状态转换过程。

1.3.14 如何理解Java中的守护线程?

在Java中，线程分为两种类型：用户线程和守护线程。用户线程是程序中普通的线程，当所有的用户线程都执行完毕后，Java虚拟机会退出。而守护线程则是一种特殊的线程，主要用于提供后台服务和支持性工作。

守护线程的特点如下：

1. 守护线程是在程序中被创建的，和用户线程类似，可通过setDaemon(true)方法设置为守护线程。
2. 当所有的用户线程结束时，即使守护线程还未执行完毕，Java虚拟机也会退出，不会等待守护线程完成。
3. 守护线程通常用于执行一些非核心的、非必要的任务，例如垃圾回收线程就是一个守护线程。

在实际开发中，守护线程可以用于执行一些后台服务或监控工作。例如，你可以创建一个守护线程用于定期清理临时文件，或者创建一个守护线程用于监控某个资源的变化。守护线程的作用是为用户线

程提供支持，而不会阻止程序的退出。

1.3.15 请细说一下synchronized关键字？

在Java中，synchronized关键字用于实现线程之间的同步。它可以修饰方法、代码块以及静态方法，来保证在多线程访问共享资源时的安全性。下面我详细介绍一下synchronized的使用方式和特点：

1. 修饰方法：

- synchronized可以修饰普通方法，使用该关键字修饰的方法称为同步方法。在同步方法中，当线程访问该方法时，会自动获得该方法所在对象的监视器（即锁），其他线程则被阻塞，直到该线程执行完该方法，释放锁。
- 同步方法是隐式获取锁的方式，它的锁是该方法所在对象的实例锁（即对象锁）。

2. 修饰代码块：

- synchronized还可以修饰代码块，使用该关键字修饰的代码块称为同步块。在同步块中，需要指定获取锁的对象或者类。
- 当线程执行到同步块时，会尝试获取指定对象（或类）的监视器（即锁），其他线程则被阻塞，直到该线程执行完同步块，释放锁。
- 同步块的锁可以是任意对象，也可以使用this关键字表示当前对象的实例锁，还可以使用类名.class表示类锁。

3. 特点：

- synchronized保证了同一时间只有一个线程可以执行同步代码块或同步方法，从而保证了共享资源的安全访问。
- synchronized关键字具有原子性，即在同步代码块或同步方法内的操作不会被其他线程打断。
- synchronized还具有可见性，即一个线程对共享变量的修改会立即对其他线程可见。
- synchronized关键字是重量级锁，获取和释放锁涉及到用户态和内核态之间的切换，会带来一定的性能开销。



需要注意的是，synchronized关键字只能实现线程之间的互斥访问共享资源，不能保护数据的原子性。如果需要保证数据操作的原子性，可以使用java.util.concurrent.atomic包下的原子类，或者使用Lock接口的实现类，如ReentrantLock。

1.3.16 说说你对序列化和反序列化的理解？


序列化（Serialization）和反序列化（Deserialization）是将对象转换为字节序列和将字节序列转换为对象的过程。

序列化的过程将对象转换为字节流的形式，使得对象可以被存储、传输或持久化。序列化后的字节流可以保存到文件、数据库中，或者通过网络传输到远程主机。

反序列化的过程则是将字节流重新转换为原始的对象。通过反序列化，可以将序列化的对象反复利用，使得对象的状态可以在不同的运行环境中恢复。

序列化和反序列化在许多实际应用中都扮演着重要的角色。例如，在分布式系统中，可以通过序列化和反序列化将对象在不同的节点之间传递；在Web开发中，可以将对象转换为JSON或XML格式进行传输；在数据库中，可以将对象以二进制形式存储。

Java提供了序列化和反序列化的机制，通过实现Serializable接口并使用ObjectOutputStream和ObjectInputStream类进行对象的序列化和反序列化操作。需要注意的是，被序列化的对象的类必须实现Serializable接口，同时要注意类的版本号和成员变量的序列化和反序列化顺序的一致性。

 总结起来，序列化和反序列化是一种将对象转换为字节序列并恢复为原始对象的机制，它们在实际开发中广泛应用于对象的存储、传输和持久化等方面。

1.3.17 说说你对反射技术的理解？

反射技术是Java中一种强大的功能，它允许程序在运行时动态地获取和操作类的信息。通过反射，可以在运行时通过获取类的名称、方法、字段等信息，动态地创建对象，调用方法，访问和修改字段，以及执行其他与类相关的操作。

反射技术的核心是java.lang.reflect包中的一组类和接口。以下是一些常用的反射类和接口：

1. Class类：代表一个类或接口，在运行时可以通过它获取和操作类的信息，例如获取类的名称、父类、实现的接口、构造函数、方法、字段等。
2. Constructor类：代表类的构造函数，在运行时可以通过它创建对象。
3. Method类：代表类的方法，在运行时可以通过它调用方法。
4. Field类：代表类的字段，在运行时可以通过它访问和修改字段的值。

通过反射，可以实现以下功能：

1. 动态创建对象：通过Class类的newInstance()方法可以创建类的实例，相当于调用了该类的无参构造函数。


```
1 Class<?> clazz = MyClass.class;
2 Object obj = clazz.newInstance();
```

1. 动态调用方法：通过 `Method` 类的 `invoke()` 方法可以调用类的方法，可以传递参数并获取返回值。

```
1 Class<?> clazz = MyClass.class;
2 Object obj = clazz.newInstance();
3 Method method = clazz.getMethod("methodName", int.class);
4 Object result = method.invoke(obj, 123);
```

1. 动态访问和修改字段：通过 `Field` 类可以获取并修改类的字段的值。

```
1 Class<?> clazz = MyClass.class;
2 Object obj = clazz.newInstance();
3 Field field = clazz.getDeclaredField("fieldName");
4 field.setAccessible(true); // 设置访问权限，私有字段需要设置为可访问Object value = f
5 field.set(obj, newValue); // 设置字段的值
```

使用反射技术可以在运行时动态地创建对象、调用方法、访问和修改字段，这使得程序具有更大的灵活性和扩展性。同时，反射技术也带来一定的性能开销，因此在性能敏感的场景下需要谨慎使用。

1.3.18 HashMap特点及应用场景？

HashMap是Java中常用的哈希表实现类，它实现了Map接口，用于存储键值对数据。HashMap具有以下几个特点和应用场景：

1. 高效的查找和插入操作：HashMap的底层是通过哈希表实现的，使用了哈希函数将键转换为数组索引。因此，在查找和插入元素时，HashMap具有较高的效率。
2. 键值对存储：HashMap可以存储键值对类型的数据，每个键都是唯一的。通过键可以快速查找对应的值。HashMap中的键和值都可以为null，但一个HashMap只能有一个null键。
3. 无序的元素：HashMap中元素的存储顺序是不固定的，取决于哈希值的计算结果。如果需要有序的元素，可以考虑使用LinkedHashMap。
4. 线程不安全：HashMap是非线程安全的数据结构。如果在多线程环境下使用HashMap，可能会导致数据不一致的问题。如果需要在多线程环境下使用HashMap，可以考虑使用ConcurrentHashMap。

5. 可变大小：HashMap的大小是可以动态调整的，根据实际的元素数量自动扩容和收缩。
6. 应用场景：HashMap适用于需要根据键快速查找对应值的场景。例如，常见的应用场景包括缓存、数据索引、字典等。通过将键与对应的值进行关联，可以快速地进行查找和操作。



需要注意的是，在使用HashMap时，要特别注意hashCode和equals的重写。由于HashMap使用键的hashCode值来确定数组索引，因此，hashCode的结果应该尽量分布均匀，以提高哈希表的性能。同时，equals方法的结果也会影响键的比较和查找。如果重写了equals方法，需要保证与hashCode方法的一致性。

1.3.25 LinkedHashMap特点及应用场景？

LinkedHashMap是Java中的一种Map实现类，它基于哈希表和双向链表的数据结构，可以保持插入顺序或者访问顺序的有序性。LinkedHashMap具有以下几个特点和应用场景：

1. 有序性：LinkedHashMap可以按照元素的插入顺序或者访问顺序进行遍历。在插入顺序模式下，元素的顺序与插入的顺序一致；在访问顺序模式下，元素会根据访问的顺序进行排序。
2. 高效的检索操作：由于LinkedHashMap基于哈希表的实现，在读取操作时具有较高的性能。通过哈希表的散列算法，可以快速定位到元素所在的位置，提高了检索的效率。
3. 可变大小：LinkedHashMap的大小是可以动态调整的，可以根据实际元素数量自动扩容和收缩。这样可以节省内存空间，并提高性能。
4. 迭代器的一致性：LinkedHashMap的迭代器可以保证在遍历过程中，元素的顺序与插入或访问的顺序一致。这对于需要保持顺序的遍历操作非常有用。
5. 应用场景：LinkedHashMap适用于需要有序存储和遍历的场景。例如，在LRU（Least Recently Used）缓存实现中，可以使用LinkedHashMap来维护缓存的顺序，把最近使用的元素放在链表的头部，最少使用的元素放在链表的尾部。同时，LinkedHashMap还可以用于实现LRU近似算法的缓存淘汰策略，通过设置accessOrder为true，使得最近访问的元素被放在链表的尾部，从而实现近似的LRU策略。



需要注意的是，LinkedHashMap相对于HashMap来说，由于需要维护链表的结构，在插入和删除操作上会有一定的性能开销。所以在不需要保持有序性的场景下，选择HashMap可能更合适。

1.3.26 说说TCP协议以及其特点？

TCP（传输控制协议）是一种可靠的传输层协议，用于在互联网上可靠地传输数据。以下是TCP协议的特点：

1. 可靠性：TCP提供数据传输的可靠性，通过使用确认机制、序列号和超时重传来确保数据的可靠传递。TCP会对每个发送的数据包进行确认，如果发送方没有收到确认，会重新发送数据包，直到对方确认接收。
2. 有序性：TCP保证数据的有序传输。每个TCP报文段都有一个序列号，接收方根据序列号对接收的数据进行按序重组，保证数据的正确排序。
3. 流量控制：TCP使用滑动窗口机制来进行流量控制，确保发送方发送的速度不会超过接收方的处理能力。接收方会通知发送方可接收的数据量，发送方根据接收方的反馈进行调整，以避免数据丢失和网络拥塞。
4. 拥塞控制：TCP使用拥塞控制算法来避免网络拥塞。通过动态调整发送速率和传输窗口大小，TCP可以适应网络的变化，保持网络的稳定性和公平性。
5. 面向连接：TCP是一种面向连接的协议，通信前需要建立一个连接。连接的建立使用了三次握手的过程，即发送方和接收方都需要确认连接的建立。通信结束后，需要进行四次挥手的过程来正常关闭连接。
6. 全双工通信：TCP支持全双工通信，即发送方和接收方可以同时发送和接收数据，实现双向的数据传输。
7. 基于字节流：TCP以字节流方式传输数据，将数据分割成较小的TCP报文段进行传输。



由于TCP具备可靠性和流控机制，适用于对数据传输的可靠性、有序性要求较高的场景，例如文件传输、电子邮件等。但是相对于UDP协议，TCP的传输效率较低。

1.4 API技术拓展分析

1.4.1 常用日期API有哪些？

Java中常用的日期API主要集中在 `java.time` 包下，这是Java 8引入的新的日期和时间API。以下是一些常用的日期API：

1. `LocalDate`：表示日期，包含年、月、日的值。
2. `LocalTime`：表示时间，包含时、分、秒的值。
3. `LocalDateTime`：表示日期和时间的组合。
4. `Instant`：表示时间戳，代表从1970年1月1日开始的纳秒数。

5. `Duration`：用于处理时间段的类，可以表示秒、纳秒等单位的时间差。
6. `Period`：用于处理日期间隔的类，可以表示年、月、日等单位的日期差。
7. `ZoneId`：表示时区的类，用于在不同时区之间进行转换。
8. `ZonedDateTime`：表示带有时区的日期和时间。
9. `DateTimeFormatter`：提供了格式化和解析日期时间字符串的功能。

除了 `java.time` 包，Java中还有其他一些日期API，如：

1. `Date` 和 `Calendar`：是早期的日期API，虽然已经过时，但仍然可以在某些情况下使用。
2. `SimpleDateFormat`：用于格式化和解析日期时间字符串，使用较为方便，但不是线程安全的。

以上就是Java中常用的日期API。在使用日期和时间相关的操作时，建议使用新的 `java.time` 包中的API，它们提供了更丰富的功能和更好的易用性。

1.4.2 wait/notify/notifyAll方法应用

基于Object类中的wait/notify/notifyAll方法可以实现线程之间的通讯，这里的wait/notify/notifyAll方法必须用在同步方法或同步代码块内部，由对象锁调用,具体方法含义如下：

1. wait方法使当前线程进入等待状态，同时会放弃对象锁，并且等待其他线程调用同一对象上的notify或notifyAll方法来唤醒它。
2. notify方法用于唤醒在同一对象上调用wait方法进入等待状态的线程。如果有多个线程在该对象上等待，则只能唤醒其中一个线程，并由系统决定具体唤醒哪个线程。
3. notifyAll方法用于唤醒在同一对象上调用wait方法进入等待状态的所有线程，使它们有机会继续执行。

1.4.3 sleep和wait方法的区别？

sleep和wait方法是在编程中用于控制线程执行的两种常见方式，它们的主要区别如下：

1. 功能不同：

- sleep方法是用于暂停当前线程的执行，让其进入阻塞状态，以便其他线程有机会执行。
- wait方法是用于将当前线程放入等待状态，并释放它所持有的锁，直到其他线程调用同一对象上的notify或notifyAll方法来唤醒它。

2. 调用方式不同：

- sleep方法是被线程对象直接调用的，例如Thread.sleep(millis)。
- wait方法是在对象上调用的，例如object.wait()，其中object是一个具有同步能力的对象。


3. 锁的释放不同：

- sleep方法并不会释放线程持有的锁，线程在阻塞期间仍然保持对锁的所有权。

- wait方法会主动释放线程持有的锁，这允许其他线程在同一对象上获得锁并执行相关操作。

4. 唤醒的方式不同：

- sleep方法在指定的时间到期后会自动唤醒线程，线程将继续执行。
- wait方法需要在其他线程调用同一对象上的notify或notifyAll方法，来唤醒等待中的线程。

 总的来说，sleep方法主要用于控制线程的暂停执行时间，而wait方法主要用于线程间的协作和同步。使用它们时需要根据具体的需求和场景来选择适合的方法。

1.4.4 描述下Java线程池中的常见参数？

Java线程池提供了一些常见的参数，用于配置线程池的行为和性能。下面是Java线程池常见参数的描述：

1. `corePoolSize`（核心池大小）：表示线程池中核心线程的数量。核心线程会一直存活，即使没有任务需要执行。默认情况下，核心池大小为0。
2. `maximumPoolSize`（最大池大小）：表示线程池中最大线程的数量。当工作队列已满并且池中的线程数小于最大池大小时，线程池会创建新的线程来处理任务，直到达到最大池大小。默认情况下，最大池大小为`Integer.MAX_VALUE`。
3. `keepAliveTime`（线程空闲时间）：表示当线程池中的线程数量超过核心池大小时，空闲线程的存活时间。当线程空闲时间超过`keepAliveTime`时，多余的线程会被销毁，直到线程池中的线程数等于核心池大小。默认情况下，空闲线程会立即被销毁。
4. `unit`（时间单位）：用于指定`keepAliveTime`的时间单位，可以是纳秒（`TimeUnit.NANOSECONDS`）、微秒（`TimeUnit.MICROSECONDS`）、毫秒（`TimeUnit.MILLISECONDS`）、秒（`TimeUnit.SECONDS`）、分钟（`TimeUnit.MINUTES`）、小时（`TimeUnit.HOURS`）、天（`TimeUnit.DAYS`）。
5. `workQueue`（工作队列）：用于存储等待执行的任务。Java提供了多种类型的工作队列，如`SynchronousQueue`、`LinkedBlockingQueue`、`ArrayBlockingQueue`等。默认情况下，使用无界队列`LinkedBlockingQueue`。
6. `threadFactory`（线程工厂）：用于创建新线程。可以自定义线程工厂来指定线程的命名规则、优先级等。
7. `handler`（拒绝策略）：用于处理线程池队列已满且无法接受新任务时的情况。Java提供了几种内置的拒绝策略，如`AbortPolicy`、`CallerRunsPolicy`、`DiscardPolicy`、`DiscardOldestPolicy`。也可以自定义拒绝策略。

1.4.5 线程池的拒绝策略有哪些？

- a. AbortPolicy: 丢弃任务并抛出 `RejectedExecutionException` 异常（默认拒绝策略）。
- b. DiscardPolicy: 丢弃任务，但是不抛出异常。
- c. DiscardOldestPolicy: 丢弃队列最前面的任务，然后重新提交被拒绝的任务。
- d. CallerRunsPolicy: 由调用线程（提交任务的线程）处理该任务。

1.4.6 Java中线程池执行任务的过程？

这里以 `ThreadPoolExecutor` 为例进行说明，`ThreadPoolExecutor` 是Java中线程池的实现类，它实现了 `ExecutorService` 接口，用于执行任务的管理和调度。下面是 `ThreadPoolExecutor` 对象执行任务的过程：

1. 状态检查：在执行任务之前，`ThreadPoolExecutor` 会检查自身的状态，例如是否已关闭。
2. 判断核心线程数：`ThreadPoolExecutor` 会检查当前线程池中的线程数量，如果未达到核心线程数（通过 `corePoolSize` 指定），则创建新的线程来执行任务。
3. 任务队列处理：如果已达到核心线程数，`ThreadPoolExecutor` 会将任务放入任务队列（通过 `BlockingQueue` 实现），等待执行。
4. 判断最大线程数：如果任务队列已满并且当前线程池中的线程数量还未达到最大线程数（通过 `maximumPoolSize` 指定），则创建新的线程来执行任务。
5. 拒绝策略：如果任务队列已满且当前线程池中的线程数量已达到最大线程数，`ThreadPoolExecutor` 会根据设置的拒绝策略（通过 `RejectedExecutionHandler` 指定）来决定如何处理无法接受的新任务。常见的拒绝策略有：抛出异常、丢弃任务、丢弃队列中最旧的任务、直接在调用者线程中执行任务。
6. 任务执行：`ThreadPoolExecutor` 会从任务队列中取出任务，并将任务分配给工作线程来执行。工作线程会执行任务中的 `run()` 方法。
7. 结果返回和异常处理：任务执行完成后，可以通过 `Future` 对象获取任务的执行结果。如果任务执行过程中发生异常，`ThreadPoolExecutor` 会捕获并处理异常。
8. 线程池关闭：当需要关闭线程池时，可以调用 `shutdown()` 方法。`ThreadPoolExecutor` 会停止接受新任务，并尽量将已提交的任务执行完毕。如果需要立即停止所有的任务并返回尚未执行的任务列表，可以调用 `shutdownNow()` 方法。



`ThreadPoolExecutor` 提供了灵活的线程池管理和任务调度，能够根据配置的参数和策略来控制线程池的行为，以提高性能和效率。

1.4.7 核心线程池大小如何设置？

1. CPU 密集型任务($N+1$)：这种任务消耗的主要是 CPU 资源，可以将线程数设置为 N （CPU 核心数）+1，比 CPU 核心数多出来的一个线程是为了防止线程偶发的缺页中断，或者其它原因导致的任务暂停而带来的影响。一旦任务暂停，CPU 就会处于空闲状态，而在这种情况下多出来的一个线程就可以充分利用 CPU 的空闲时间。
2. I/O 密集型任务($2N+1$)：这种任务应用起来，系统会用大部分的时间来处理 I/O 交互，而线程在处理 I/O 的时间段内不会占用 CPU 来处理，这时就可以将 CPU 交出给其它线程使用。因此在 I/O 密集型任务的应用中，我们可以多配置一些线程，具体的计算方法是 $2N$ 。

1.4.8 如何判断是 CPU 密集还是 IO 密集型任务？

CPU 密集型简单理解就是利用 CPU 计算能力的任务比如你在内存中对大量数据进行排序。单凡涉及到网络读取，文件读取这类都是 IO 密集型，这类任务的特点是 CPU 计算耗费时间相比于等待 IO 操作完成的时间来说很少，大部分时间都花在了等待 IO 操作完成上。

1.4.9 说说synchronized关键字底层锁升级是怎样的？

在Java中，锁是一种用于实现线程同步的机制。为了提高性能，Java中的锁可以进行升级操作，将低级别的锁升级为高级别的锁。这个过程被称为锁升级。

Java中的锁升级主要涉及到三种类型的锁：偏向锁、轻量级锁和重量级锁。

1. 偏向锁 (Biased Locking)：偏向锁是一种针对多线程竞争情况下的常见场景进行优化的锁机制。当一个线程获取锁时，会在对象头部记录该线程的ID，并将锁标记为偏向锁。此后，如果再次执行相同线程请求锁的操作，就可以直接获得锁，而无需进行任何同步操作。这种情况下，锁的性能开销非常小。偏向锁适用于大部分情况下都只有一个线程访问共享资源的场景。
2. 轻量级锁 (Lightweight Locking) (也称之为自旋锁)：当多个线程竞争同一个锁时，偏向锁就会升级为轻量级锁。轻量级锁使用CAS（比较-交换）操作来实现，不涉及系统调用。它通过在对象头部的标记字段存储锁记录的指针以及线程ID的方式，来避免互斥同步操作。当线程获取轻量级锁时，将通过CAS操作将对象头部的标记字段修改为指向线程私有的锁记录。如果CAS操作成功，表示当前线程获取了锁。否则，表示其他线程竞争成功，锁升级为重量级锁。
3. 重量级锁 (Heavyweight Locking)：当多个线程竞争同一个锁并且轻量级锁获取失败时，锁会升级为重量级锁。重量级锁依赖于操作系统的底层同步机制，如互斥量 (mutex) 或信号量 (semaphore)。它会导致线程阻塞和切换，且性能开销较大。重量级锁适用于多个线程频繁争用锁的场景。

总结一下，Java中的锁升级过程是从偏向锁到轻量级锁，再到重量级锁。锁升级的过程是根据线程竞争的情况来进行判断和执行的。在大部分情况下，偏向锁和轻量级锁的性能开销较小，能够提高程序执行效率。只有在多线程竞争激烈的情况下，才会升级为重量级锁以保证线程安全。因此，在设计高并发系统时，需要根据实际情况选择适当的锁机制，以提高系统性能和线程安全性。

1.4.10 说说你对volatile关键字的理解？

在Java中，`volatile` 是一个关键字，用于修饰变量。它具有以下特点：

1. 可见性：`volatile` 关键字保证了变量的可见性。当一个线程修改了 `volatile` 修饰的变量的值时，其他线程可以立即看到修改后的值，而不会使用缓存中的旧值。
2. 禁止指令重排序：`volatile` 关键字禁止了对被修饰变量的指令重排序。指令重排序是编译器和处理器为了提高执行效率而进行的重排序操作，但可能会导致多线程程序出现问题。通过使用 `volatile` 关键字，可以确保特定的变量操作按照预期的顺序执行，避免了一些潜在的线程安全问题。
3. 不保证原子性：`volatile` 关键字并不能保证变量操作的原子性。如果一个操作涉及到多个步骤，且需要保证原子性的话，还需要使用其他的同步手段，例如 `synchronized` 关键字或 `java.util.concurrent` 包下的原子类。



需要注意的是，`volatile` 关键字主要用于修饰多个线程之间共享的变量，在保证可见性和禁止指令重排序方面带来了便利。然而，它并不能解决所有的线程安全问题，例如复合操作的原子性。在编写多线程程序时，仍然需要根据具体场景选择适当的线程同步手段。

1.4.11 说说你对ThreadLocal的理解？

ThreadLocal是一个Java中的线程局部变量。它的主要作用是为每个线程提供一个独立的变量副本，每个线程都可以独立地操作自己的副本，互不干扰。以下是我对ThreadLocal的理解：

1. 独立副本：ThreadLocal为每个线程提供了一个独立的变量副本，每个线程都可以对该副本进行操作，而不会影响其他线程的副本。这样可以保证线程之间的数据隔离，每个线程都可以维护自己的状态。
2. 线程隔离：由于每个线程拥有自己的副本，因此线程之间的变量不会相互冲突。这在多线程并发环境下非常有用，可以避免数据竞争和线程安全问题。

3. 解决共享资源问题：在某些情况下，多个线程需要访问同一个变量，但是每个线程使用的值又不同，这时可以使用ThreadLocal来解决这个问题。每个线程可以通过ThreadLocal获取自己的副本，而无需使用锁或同步机制。
4. 垃圾回收：ThreadLocal使用ThreadLocalMap来存储每个线程的变量副本，键是ThreadLocal实例本身，值是该线程对应的变量副本。当线程结束后，线程本地变量会自动被回收，不会造成内存泄漏。



需要注意的是，ThreadLocal并不能解决所有的线程安全问题，它只是提供了一种线程隔离的机制。使用ThreadLocal时需要小心管理资源，避免造成内存泄漏。同时，在使用ThreadLocal时，如果不再需要使用该变量，要记得及时清除，以免累积过多的无用副本。

1.4.12 说说Java中的四种引用类型？

Java中有四种引用类型，它们分别是强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）和虚引用（Phantom Reference）。它们之间的主要区别如下：

1. 强引用（Strong Reference）：

- 强引用是默认的引用类型，当我们创建一个对象并将其赋值给一个引用变量时，就会创建一个强引用。
- 强引用会阻止垃圾回收器回收对象。只有当没有任何强引用指向对象时，对象才会成为垃圾回收的候选对象。

2. 软引用（Soft Reference）：

- 软引用是用于描述还有用但并非必需的对象。在内存不足时，垃圾回收器会根据各个对象的软引用情况来决定是否回收这些软引用对象。
- 当内存充足时，软引用对象不会被垃圾回收。只有当垃圾回收器判断内存不足时，才会回收这些软引用对象。


3. 弱引用（Weak Reference）：

- 弱引用是用于描述非必需且随时可能被回收的对象。垃圾回收器在执行回收时，不管内存是否充足，都会回收这些弱引用对象。
- 弱引用通常用于解决某些特定的内存敏感问题，可以避免内存泄漏。

4. 虚引用（Phantom Reference）：

- 虚引用是最弱的引用类型，它是为了监控对象被垃圾回收的状态而存在的。虚引用无法通过get()方法获取到对象，也无法访问对象的任何数据。

- 当垃圾回收器即将回收一个对象时，会将该对象的虚引用加入引用队列，通知应用程序对象已经进入可回收状态。

 这四种引用类型在Java中提供了更灵活的内存管理方式。可以根据对象的生命周期和内存需求选择正确的引用类型，以提高内存的利用率和性能。

1.4.13 请细说一下Java中的锁升级？

在Java中，锁升级是指锁从低级别到高级别的转变，这种转变的目的是为了提高并发性能和减少内存开销。


在Java中，锁的级别由低到高依次为：

无锁状态（No lock）：表示没有使用任何锁机制。多线程并发执行时，线程间不会相互干扰，可能出现无序的、不一致的结果。

偏向锁（Biased lock）：当只有一个线程访问临界区时，这个锁会自动升级为偏向锁。偏向锁会记录获取锁的线程的标识，并在后续访问时直接判断是否是同一个线程，避免不必要的同步操作。偏向锁提供了很好的线程竞争消除策略，适用于单线程和只有一个线程访问锁的场景。

轻量级锁（Lightweight lock）：当有多个线程竞争同一个锁时，偏向锁会自动升级为轻量级锁（也称为自旋锁）。轻量级锁使用CAS（比较并交换）操作来实现锁的获取和释放，避免了线程的阻塞和唤醒操作，减少了性能开销。


重量级锁（Heavyweight lock）：当轻量级锁无法获取到锁时，锁会升级为重量级锁。重量级锁使用操作系统的互斥量（mutex）来实现，确保临界区的互斥访问。重量级锁会导致线程的阻塞和唤醒操作，性能较低。

 锁升级的过程是自动进行的，由Java虚拟机自动判断和处理。锁的升级和降级过程都是为了在不同的场景下提供更好的性能和资源利用。具体是根据使用场景、线程的竞争情况和锁的状态等因素来决定是否进行锁升级或降级。

1.4.14 说说Java中的IO与NIO区别？

Java IO（Input/Output）和 NIO（New Input/Output）是 Java 中处理输入输出的两种不同的编程模型，它们之间有以下几个主要区别：

1. **阻塞与非阻塞：**Java IO 是基于阻塞模型的，也称为同步模型。当一个线程在执行 IO 操作时，如果没有数据可读或者无法立即写入，线程会被阻塞，直到有数据可读或者可以进行写入。而 NIO 则是基于非阻塞模型的，也称为异步模型。NIO 中的线程可以在执行 IO 操作时继续做其他的事情，不需要等待 IO 操作完成。
2. **缓冲：**Java IO 使用字节流和字符流进行读写操作时，通常需要借助缓冲流提供缓冲功能，以提高读写的效率。而 NIO 中的 Channel 和 Buffer 提供了内置的缓冲功能，不需要额外的缓冲流。
3. **数据处理方式：**Java IO 是以流为基础的，每次从流中读取一个或多个字节或字符，逐个进行处理。而 NIO 则是以缓冲区（Buffer）为基础的，可以一次读取或写入多个数据。NIO 中的 Channel 和 Buffer 使用起来非常灵活，可以支持复杂的数据处理需求。
4. **选择器（Selector）：**NIO 中的 Selector 提供了多路复用的功能，可以通过一个线程处理多个 Channel 的 IO 操作。这种机制允许同时监听多个输入流，只有在有数据可读或可写入时才会被唤醒，提高了资源利用率。

 总体来说，Java IO 适用于处理少量的连接和简单的读写操作，适合于传统的阻塞模型。而 NIO 适用于处理大量连接和高并发的读写操作，适合于构建高性能和可扩展的网络应用程序。需要注意的是，Java 7 引入的 NIO.2，也称为 AIO（Asynchronous IO），进一步增强了 Java 在异步 IO 方面的能力，提供了更方便的异步文件处理等功能。

1.4.15 ArrayList线程安全吗？如何保证线程安全？

ArrayList线程不安全，可以通过如下几种方案实现线程安全的ArrayList：

- 使用 Vector 代替 ArrayList。（不推荐，Vector是一个历史遗留类）
- 使用 Collections.synchronizedList 包装 ArrayList，然后操作包装后的 list。
- 使用 CopyOnWriteArrayList 代替 ArrayList。

1.4.16 CopyOnWriteArrayList是怎么实现的呢？

CopyOnWriteArrayList就是线程安全版本的ArrayList。

CopyOnWriteArrayList采用了一种读写分离的并发策略。CopyOnWriteArrayList容器允许并发读，读操作是无锁的，性能较高。至于写操作，比如向容器中添加一个元素，则首先将当前容器复制一份，然后在新副本上执行写操作，结束之后再将原容器的引用指向新容器。适合读多写少的场景。

1.4.17 HashMap如何解决散列冲突？

HashMap使用了拉链法来解决散列冲突。当发生散列冲突时，即多个元素哈希值映射到了同一个桶（bucket）时，HashMap会在该桶上使用链表或红黑树来存储冲突的元素。

具体而言，当链表的长度超过阈值（默认为8）时，HashMap会将链表转换为红黑树，从而提高在冲突桶上的查找和删除的性能。而当红黑树的节点数量小于等于6时，HashMap会将红黑树转换回链表结构，以节省内存开销。

这种使用链表和红黑树结合的方式，既可以在冲突较少的情况下保持较低的内存消耗，又可以在冲突较多的情况下提供更高的性能。

需要注意的是，JDK8中对HashMap做了一些优化，例如，当数组容量较小时，使用链表代替红黑树；当数组容量达到阈值（默认为64）时，才会使用红黑树结构。这样可以减少在容量较小的情况下使用红黑树带来的额外开销。

综上所述，JDK8的HashMap使用拉链法解决散列冲突，通过链表和红黑树的结合使用，提供了更好的性能和内存利用率。

✨ 总的来说，桶数组是用来存储数据元素，链表是用来解决冲突，红黑树是为了提高查询的效率。

- 数据元素通过映射关系，也就是散列函数，映射到桶数组对应索引的位置
- 如果发生冲突，从冲突的位置拉一个链表，插入冲突的元素
- 如果链表长度 >8 & 数组大小 ≥ 64 ，链表转为红黑树
- 如果红黑树节点个数 <6 ，转为链表

1.4.18 HashMap的扩容机制是什么样的？

HashMap底层的数组长度是固定，随着元素增加，需要进行动态扩容。

扩容的时机发生在put的时候，当当前HashMap的元素个数达到一个临界值的时候，就会触发扩容，把所有元素rehash之后再放在扩容后的容器中，这是一个相当耗时的操作。

临界值threshold 就是由加载因子和当前容器的容量大小来确定的。临界值（threshold）= 默认容量（DEFAULT_INITIAL_CAPACITY）* 默认扩容因子（DEFAULT_LOAD_FACTOR）。比如容量是16，负载因子是0.75，那就是大于 $16 \times 0.75 = 12$ 时，就会触发扩容操作，扩容之后的长度是原来的二倍。容量都是2的幂次方。

1.4.19 为什么HashMap的容量是2的幂次方呢？

- 第一个原因是为了方便哈希取余。

将元素放在table数组上面，是用hash值%数组大小定位位置，而HashMap是用hash值&(数组大小-1)，却能和前面达到一样的效果，这就得益于HashMap的大小是2的倍数，2的倍数意味着该数的二进制位只有一位为1，而该数-1就可以得到二进制位上1变成0，后面的0变成1，再通过&运算，就可以得到和%一样的效果，并且位运算比%的效率高得多。HashMap的容量是2的n次幂时，(n-1)的2进制也就是1111111***111这样形式的，这样与添加元素的hash值进行位运算时，能够充分的散列，使得添加的元素均匀分布在HashMap的每个位置上，减少hash碰撞。

- 第二个方面是在扩容时，利用扩容后的大小也是2的倍数，将已经产生hash碰撞的元素完美的转移到新的table中去。

1.4.20 HashMap中为什么选择了0.75作为默认加载因子？

简单来说，这是对 空间 成本和 时间 成本平衡的考虑。

我们都知道，HashMap的散列构造方式是Hash取余，负载因子决定元素个数达到多少时候扩容。

假如我们设的比较大，元素比较多，空位比较少的时候才扩容，那么发生哈希冲突的概率就增加了，查找的时间成本就增加了。

我们设的较小的话，元素比较少，空位比较多的时候就扩容了，发生哈希碰撞的概率就降低了，查找时间成本降低，但是就需要更多的空间去存储元素，空间成本就增加了。

1.4.21 HashMap初始化传17，实际容量是多少？

简单来说，就是初始化时，传的不是2的倍数时，HashMap会向上寻找，离得最近的2的倍数，所以传入17，但HashMap的实际容量是32。

1.4.22 Jdk8中rehash的过程是怎么样的？

HashMap扩容后，需要将每一个元素重新rehash放置到HashMap中。jdk1.8中的做了优化操作，可以不需要再重新计算每一个元素的哈希值。

因为HashMap的初始容量是2的次幂，扩容之后的长度是原来的二倍，新的容量也是2的次幂，所以，元素，要么在原位置，要么在原位置再移动2的次幂。

看下这张图，n为table的长度，图 a 表示扩容前的key1和key2两种key确定索引的位置，图 b 表示扩容后key1和key2两种key确定索引位置。

1.4.23 HashMap的put操作流程是什么样的？

HashMap的put操作主要包括以下几个步骤：

1. 计算键的哈希值：首先，通过键的hashCode()方法计算键的哈希值。哈希值是通过将键对象转换为整数计算出来的，用于将键映射到哈希表的索引位置。
2. 映射哈希值到数组位置：通过对哈希值进行与操作（ $\text{hash} \& (\text{length}-1)$ ），将哈希值映射到哈希表的数组索引位置。
3. 检查数组位置是否已经存在元素：在该索引位置上检查是否已经存在元素。如果该位置为空，表示没有冲突，直接将键值对放入该位置。
4. 冲突处理（链表或红黑树）：如果该位置已经存在元素（可能是哈希冲突），则需要进行冲突处理。HashMap中采用拉链法（链表或红黑树）来解决冲突。首先，查找链表（或红黑树）以检查是否存在相同的键。如果找到相同的键，则更新对应的值。如果没有找到相同的键，则将新的键值对插入链表（或红黑树）的末尾。
5. 动态扩容：在插入键值对后，检查是否需要扩容。如果当前元素数量大于等于负载因子乘以容量，就需要进行扩容操作。扩容是为了保持哈希表的平均填充因子在一个较低的水平，以提高查询的效率。

通过上述流程，HashMap的put操作可以将键值对插入到哈希表中。值得注意的是，HashMap是非线程安全的，如果在多线程环境下使用，需要采取额外的同步措施。

1.4.24 HashMap中的散列值（哈希值）是怎么得到的？

HashMap的哈希函数是先拿到 key 的hashCode，是一个32位的int类型的数值，然后让hashCode的高16位和低16位进行异或操作(这种操作我们通常称之为扰动)。这么做主要是为了降低哈希冲突的概率。

1.4.25 解决哈希冲突的方法有哪几种，HashMap中采用的什么方式？

解决哈希冲突的方法常见的有下面几种：

1. 链地址法：在冲突的位置拉一个链表，把冲突的元素放进去。
2. 开放定址法：开放定址法就是从冲突的位置再接着往下找，给冲突元素找个空位。
3. 再哈希法：换种哈希函数，重新计算冲突元素的地址。
4. 建立公共溢出区：再建一个数组，把冲突的元素放进去。

HashMap中采用的是链地址法来解决Hash冲突。

1.4.26 HashMap中为什么要转换为红黑树？为什么不用二叉树/平衡树呢？

其实主要就是解决hash冲突导致链化严重的问题，如果链表过长，查找时间复杂度为 $O(n)$ ，效率变慢。

本身散列表最理想的查询效率为 $O(1)$ ，但是链化特别严重，就会导致查询退化为 $O(n)$ 。

严重影响查询性能了，为了解决这个问题，JDK1.8它才引入的红黑树。红黑树其实就是一颗特殊的二叉排序树，这个时间复杂度是 $\log(N)$ 。

红黑树本质上是一种二叉查找树，为了保持平衡，它又在二叉查找树的基础上增加了一些规则：

1. 每个节点要么是红色，要么是黑色；
2. 根节点永远是黑色的；
3. 所有的叶子节点都是黑色的（注意这里说叶子节点其实是图中的 NULL 节点）；
4. 每个红色节点的两个子节点一定都是黑色；
5. 从任一节点到其子树中每个叶子节点的路径都包含相同数量的黑色节点；

红黑树是一种平衡的二叉树，插入、删除、查找的最坏时间复杂度都为 $O(\log n)$ ，避免了二叉树最坏情况下的 $O(n)$ 时间复杂度。平衡二叉树是比红黑树更严格的平衡树，为了保持平衡，需要旋转的次数更多，也就是说平衡二叉树保持平衡的效率更低，所以平衡二叉树插入和删除的效率比红黑树要低。

1.4.27 什么时候HashMap链表转红黑树呢？为什么阈值为8呢？

树化发生在table数组的长度大于64，且链表的长度大于8的时候。

为什么是8呢？

红黑树节点的大小大概是普通节点大小的两倍，所以转红黑树，牺牲了空间换时间，更多的是一种兜底的策略，保证极端情况下的查找效率。阈值为什么要选8呢？和统计学有关。理想情况下，使用随机哈希码，链表里的节点符合泊松分布，出现节点个数的概率是递减的，节点个数为8的情况，发生概率仅为0.00000006。

至于红黑树转回链表的阈值为什么是6，而不是8？是因为如果这个阈值也设置成8，假如发生碰撞，节点增减刚好在8附近，会发生链表和红黑树的不断转换，导致资源浪费。

1.4.28 jdk1.8对HashMap主要做了哪些优化呢？为什么？

jdk1.8 的HashMap主要有五点优化：

1. 数据结构：数组 + 链表改成了数组 + 链表或红黑树。

原因：发生 hash 冲突，元素会存入链表，链表过长转为红黑树，将时间复杂度由 $O(n)$ 降为 $O(\log n)$

2. 链表插入方式：链表的插入方式从头插法改成了尾插法

简单说就是插入时，如果数组位置上已经有元素，1.7 将新元素放到数组中，原始节点作为新节点的后继节点，1.8 遍历链表，将元素放置到链表的最后。

原因：因为 1.7 头插法扩容时，头插法会使链表发生反转，多线程环境下会产生环。

3. 扩容rehash：扩容的时候 1.7 需要对原数组中的元素进行重新 hash 定位在新数组的位置，1.8 采用更简单的判断逻辑，不需要重新通过哈希函数计算位置，新的位置不变或索引 + 旧的数组容量大小。

原因：提高扩容的效率，更快地扩容。

4. 扩容时机：在插入时，1.7 先判断是否需要扩容，再插入，1.8 先进行插入，插入完成再判断是否需要扩容；

5. 散列函数：1.7 做了四次移位和四次异或，jdk1.8只做一次。原因是做 4 次的话，边际效用也不大，改为一次，提升效率。

1.4.29 TreeMap特点及应用场景？

TreeMap是Java中常用的红黑树实现类，它实现了SortedMap接口，用于存储键值对数据，并且可以根据键的排序顺序进行排序和检索。TreeMap具有以下几个特点和应用场景：

1. 有序的元素：TreeMap中的元素是按照键的自然顺序或者自定义的比较器顺序进行排序的。在插入和删除元素时，TreeMap会根据排序顺序来调整数据结构，以保证有序性。
2. 键值对存储：TreeMap和HashMap一样可以存储键值对类型的数据，每个键都是唯一的。通过键可以快速查找对应的值。TreeMap中的键和值都可以为null，但一个TreeMap只能有一个null键。
3. 高效的检索操作：由于TreeMap内部使用红黑树实现，检索操作的时间复杂度为 $O(\log N)$ 。通过键的有序性，可以进行范围查询等更加灵活的操作。
4. 线程不安全：和HashMap一样，TreeMap也是非线程安全的数据结构。如果在多线程环境下使用TreeMap，可能会导致数据不一致的问题。如果需要在多线程环境下使用TreeMap，可以考虑使用ConcurrentSkipListMap。
5. 可变大小：TreeMap的大小是可以动态调整的，根据实际的元素数量自动扩容和收缩。
6. 应用场景：由于TreeMap中的元素是有序的，适用于需要有序存储和检索的场景。例如，在需要按照键进行范围查询、排序遍历等操作时，TreeMap是一个很好的选择。常见的应用场景包括缓存、数据索引、字典等。



需要注意的是，在使用TreeMap时，要特别注意键的比较和排序。如果使用自定义的对象作为键，需要实现Comparable接口或者传入一个自定义的比较器来定义键的排序规则。另外，

如果需要线程安全的有序映射表，可以考虑使用ConcurrentSkipListMap，它是线程安全的TreeMap的实现。

1.4.30 ConcurrentHashMap特点及应用场景？

JDK8中的ConcurrentHashMap相较于之前的版本，进一步改进了性能和扩展性。以下是JDK8ConcurrentHashMap的特点和应用场景：

1. 分段锁的改进：JDK8中的ConcurrentHashMap使用了分段锁的机制，将整个数据结构分成了多个段，并且每个段下面是一个链表结构，可以同时执行多个读操作和少量的写操作。这样可以提高并发度，减少锁竞争，提升并发性能。
2. CAS操作：在JDK8中，ConcurrentHashMap的更新操作使用了CAS（Compare and Swap）操作，避免了线程阻塞等待锁的情况，进一步提升了并发性能。
3. 扩容机制的改进：JDK8中的ConcurrentHashMap的扩容机制不再一次性将整个数据结构扩容，而是逐段扩容。这样可以减少扩容时的锁竞争和数据搬迁的时间开销，提升了扩容的效率。
4. 可调整的并发级别：ConcurrentHashMap依然支持可调整的并发级别，可以通过设置不同的并发级别来控制同步的粒度，从而平衡并发性和内存占用。
5. 吞吐量优化：JDK8中的ConcurrentHashMap通过减少了锁的使用，提高了并发操作的吞吐量。同时，迭代器的性能也得到了优化，可以在不阻塞其他线程的情况下进行快速迭代。
6. 应用场景：JDK8中的ConcurrentHashMap适用于需要在高并发环境下进行读写操作的场景。例如，在高并发的缓存系统、并发的计数器、高并发的数据聚合等场景中，ConcurrentHashMap可以提供高性能的并发操作和线程安全性。



需要注意的是，在选择使用ConcurrentHashMap时，应根据具体的并发情况、安全要求和性能要求进行权衡。根据实际情况，可能需要结合其他同步机制或数据结构来实现更复杂的并发需求。

1.4.31 说说UDP协议以及它的特点？

UDP（用户数据报协议）是一种无连接的传输层协议，用于在互联网上进行快速传输的协议。以下是UDP协议的特点：

1. 无连接性：UDP是一种无连接的协议，通信双方不需要建立连接就可以直接进行数据传输。这使得UDP的开销较小，适用于一对一或一对多的简单通信。

2. 不可靠性：UDP不保证数据传输的可靠性，不提供确认机制和重传机制。一旦发送数据，就没有办法知道数据是否成功到达接收方。如果发生丢包或错误，UDP不会进行重发，接收方也无法知道数据丢失。
3. 无序性：UDP对数据的传输顺序不做保证，发送方发送的数据可能会以任意顺序到达接收方。如果需要保证数据的有序性，需要在应用层进行处理。
4. 低延迟：由于UDP协议没有连接建立和可靠性机制，数据的传输速度较快。UDP首部开销较小，适用于对实时性要求较高的应用，如视频流、语音通话等。
5. 支持广播和多播：UDP支持向多个接收方同时发送数据，包括广播和组播两种方式。广播是将数据发送给同一网络中的所有主机，而组播是将数据发送给一组特定的主机。
6. 数据包大小限制：由于底层网络通信协议的限制，UDP对每个数据包的大小有限制（最大长度为64KB）。如果发送的数据包超过限制，将会被分割成多个数据包进行传输。

UDP适合用于传输对实时性要求较高的数据，如音频和视频流等。由于UDP的简单性和低开销，它的性能更高，但可靠性和数据完整性较差。因此，需要在应用层进行数据校验和重传等错误处理，以保证数据的正确性。

1.4.32 TCP/UDP协议的异同点？

TCP（Transmission Control Protocol，传输控制协议）和UDP（User Datagram Protocol，用户数据报协议）是互联网中常用的两种传输层协议，它们在以下几个方面有异同：

1. 连接方式：
 - TCP是面向连接的协议，通过三次握手建立连接，可以进行可靠的数据传输，保证数据的顺序和可靠性。
 - UDP是无连接的协议，每个数据包之间相互独立，没有建立连接的过程，数据包不会确认接收，也无法保证数据的可靠性和顺序性。
2. 数据传输特性：
 - TCP提供面向字节的传输，将数据拆分成小的数据块进行传输，并对数据进行分段、排序和重组。TCP保证数据的可靠性，确保数据包按照发送的顺序到达目标，且不会丢失。
 - UDP以数据报形式传输数据，每个数据报都是独立的单个数据单元。UDP不保证数据的可靠性和顺序，数据包可能会丢失或乱序到达。
3. 特点和应用场景：
 - TCP适用于对数据传输可靠性要求较高的场景，如文件传输、网页浏览、电子邮件等，需要使用流式传输和确保数据完整性的应用。
 - UDP适用于对实时性要求较高，但可靠性要求相对较低的场景，如实时音视频传输、游戏、DNS查询等。UDP具有较低的延迟和传输开销，适合那些能够容忍数据丢失的场景。
4. 头部开销：

- TCP协议头部包含较多的控制信息，导致额外的头部开销，增加传输的开销。
- UDP协议头部较小，没有复杂的控制信息，传输的开销较小。

5. 流量控制和拥塞控制：

- TCP采用流量控制和拥塞控制机制，通过滑动窗口和拥塞窗口来控制数据的传输速率，以适应网络的状况。
- UDP不提供流量控制和拥塞控制机制，数据包以最快速度发送，不考虑网络状况，有可能导致网络拥塞。



总之，TCP和UDP协议在连接方式、数据传输特性、应用场景以及头部开销等方面存在较大差异。选择合适的协议取决于具体的应用需求，需要权衡可靠性、实时性和开销等因素。

1.4.33 简述一下JDK7中的 try-with-resources 好处？

try-with-resources是Java 7中引入的一个语言特性，用于更简洁和安全地处理需要关闭的资源，例如文件、数据库连接、网络连接等。它的主要好处有以下几个方面：

1. 简洁：try-with-resources语法使得
2. 代码更加简洁和易读。不再需要显式地编写finally块来关闭资源，而且资源的声明和关闭都在同一个代码块内，使得代码结构更加清晰。
3. 自动关闭：使用try-with-resources，当代码块执行完毕或者发生异常时，会自动关闭在try括号内声明的资源，无需手动关闭，减少了代码编写的工作量。
4. 安全性：try-with-resources保证了资源在代码块执行完毕时一定会被关闭，即使发生异常也不会导致资源泄漏。这有助于避免常见的错误，例如忘记关闭资源或者关闭资源位置不正确等。
5. 支持多个资源：可以在try-with-resources中同时声明和管理多个资源，使用分号 (;) 分隔各个资源的声明即可。



总的来说，try-with-resources提供了一种更简洁、更安全的方式来处理需要关闭的资源。使用这种语法可以减少资源泄漏和错误，并降低代码的复杂度，有助于提高代码质量和可维护性。

1.4.34 简述一下面向对象的”六原则一法则”。

面向对象的“六原则一法则”是软件开发中常用的一组设计原则，用来指导设计和编写高质量、可维护和可扩展的面向对象程序。以下是对这些原则和法则的简述：

1. 单一职责原则（SRP）：一个类应该有且仅有一个引起它变化的原因。每个类应该只负责一项具体职责，遵循单一职责原则可以提高类的内聚性，并减少类之间的耦合。
2. 开放封闭原则（OCP）：软件实体（类、模块、函数等）应该对扩展开放，对修改封闭。当需求变化时，应该通过扩展现有模块，而不是直接修改现有模块的代码，从而实现软件的可扩展性和可维护性。
3. 里氏替换原则（LSP）：子类对象必须能够替换其父类对象，而不会影响程序的正确性。子类必须完全遵循父类的契约，尽量不要改变父类已有定义的行为。
4. 依赖倒置原则（DIP）：高层模块不应该依赖于低层模块，两者应该都依赖于抽象。抽象不应该依赖于具体实现细节，具体实现细节应该依赖于抽象。通过依赖倒置原则，可以降低模块之间的耦合度，并增加系统的灵活性和可替换性。
5. 接口隔离原则（ISP）：拆分大接口为多个小接口，客户端只需知道自己需要使用的接口。接口隔离原则可以避免类依赖不必要的接口，从而减少了类之间的依赖关系。
6. 合成复用原则（CRP）：尽量使用合成/聚合关系，而不是继承关系来实现类的复用。通过组合关系可以更灵活地扩展和修改现有类，避免类层次结构的僵化和不必要的耦合。
7. 迪米特法则（LoD）：一个对象应该对其他对象具有最少的了解。迪米特法则强调模块之间的解耦，一个对象只与其直接的朋友通信，而不需要了解朋友的朋友。



这些原则和法则在面向对象设计中起到了重要的指导作用，能够帮助开发人员设计出高内聚、低耦合、易于拓展和维护的软件系统。

1.5 JVM技术拓展

1.5.1 JVM的构成有哪几部分？

- 第一：类加载子系统(负责将类读到内存，校验类的合法性，对类进行初始化)；
- 第二：运行时数据区(方法区/堆区/栈区/计数器,负责存储类信息，对象信息，执行逻辑)
- 第三：执行引擎(负责从指定地址对应的内存中读取数据然后解释执行以及垃圾回收操作)
- 第四：本地库接口(负责实现JAVA语言与其它编程语言之间的协同,例如调用C库)

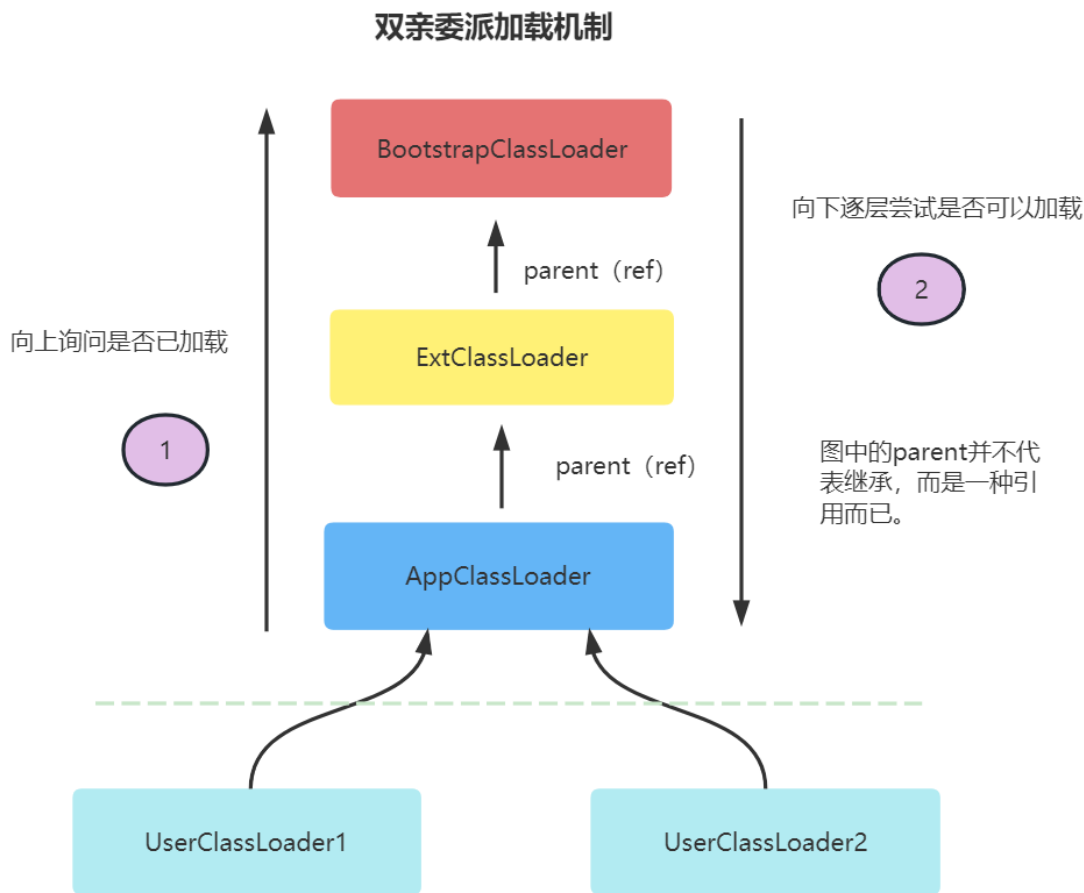
1.5.2 你知道哪些类加载器？

- 第一：BootStrapClassLoader（根类加载器，使用c编写，负责加载基础类库中的类，例如Object,String,...）
- 第二：ExtClassLoader（扩展类加载器，负责加载jdk自带的扩展类,例如javax.xxx包中的类）

- 第三：AppClassLoader（应用类加载器，负责加载我们自己写的类）
- 第四：自定义ClassLoader（当系统提供的默认类加载器不满足我们需求时，可以自己创建）

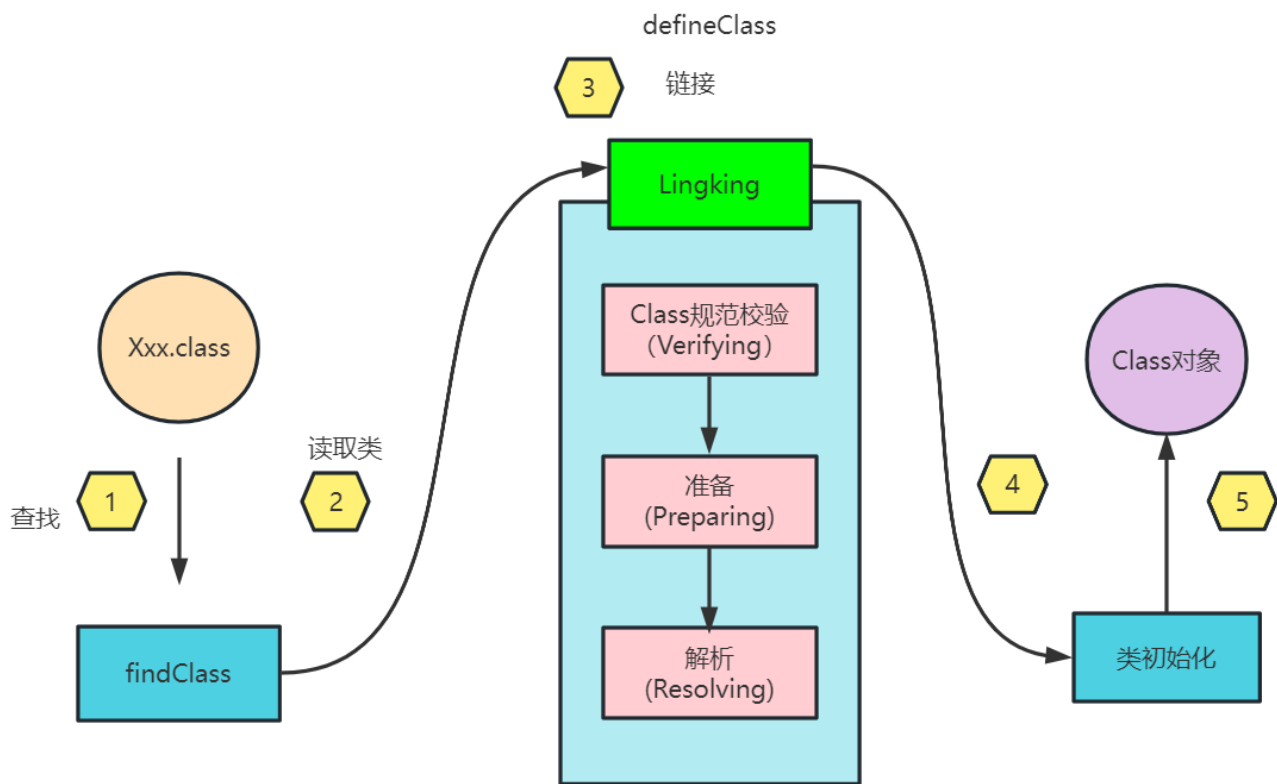
1.5.3 什么是双亲委派类加载模型？

所谓双亲委派模型可以简单理解为向上询问、向下委派。当我们的类在被加载时，首先会询问类加载器对象的parent对象(两者之间不是继承关系)，是否已经加载过此类，假如当前parent没有加载过此类，则会继续向上询问它的parent，依次递归。如果当前父加载器可以完成类加载则直接加载，假如不可以则委托给下一层类加载器去加载（可以理解为逐层分配任务）。



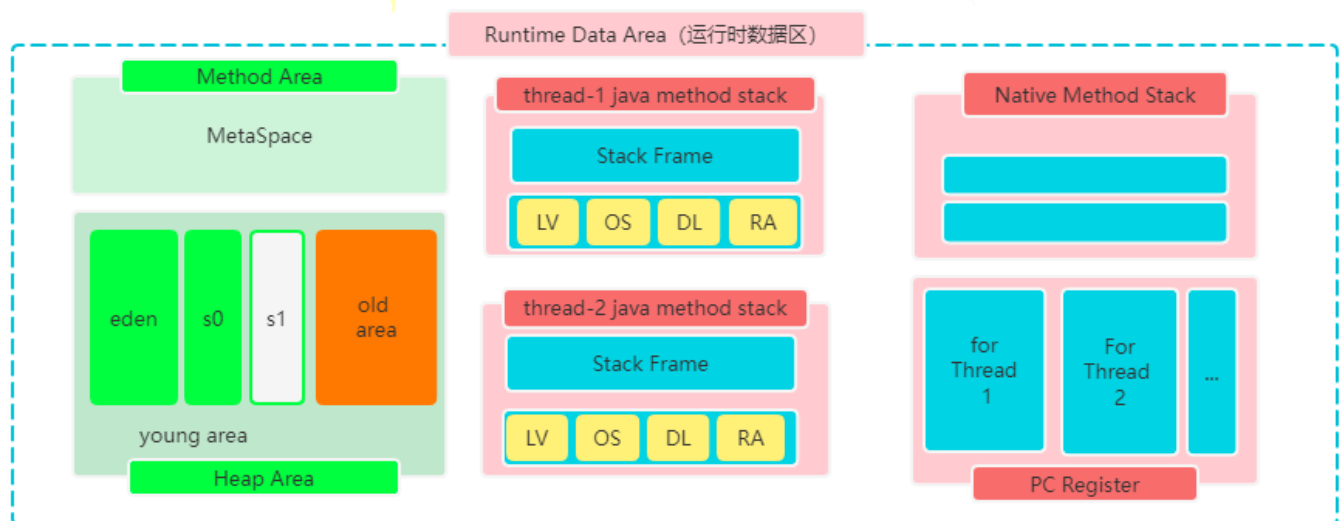
1.5.4 描述一下类加载时候的基本步骤是怎样的？

- 第一：查找类(例如通过指定路径+类全名找到指定类)
- 第二：读取类(通过字节输入流读取类到内存，并将类信息存储到字节数组)
- 第三: 对字节数组中的信息流进行校验分析以及初始化并将其结构内容存储到方法区。
- 第四: 创建字节码对象(java.lang.Class)，基于此对象封装类信息的引用，基于这些引用获取方法区类信息。



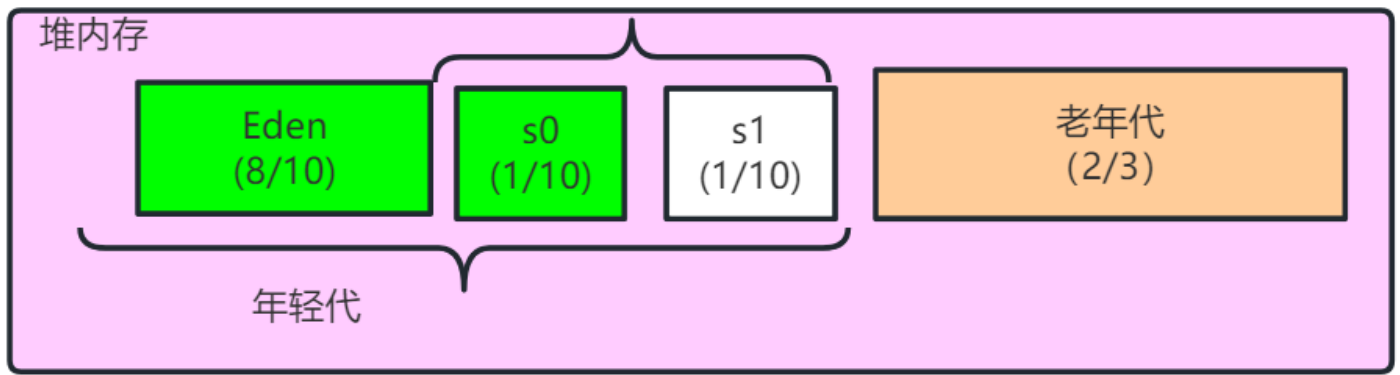
1.5.5 JVM运行内存是如何划分的？

JVM运行时内存从规范上讲有方法区(Method Area)、堆区(Heap)、Java方法栈(Stack)、本地方法栈、程序计数器(寄存器)。



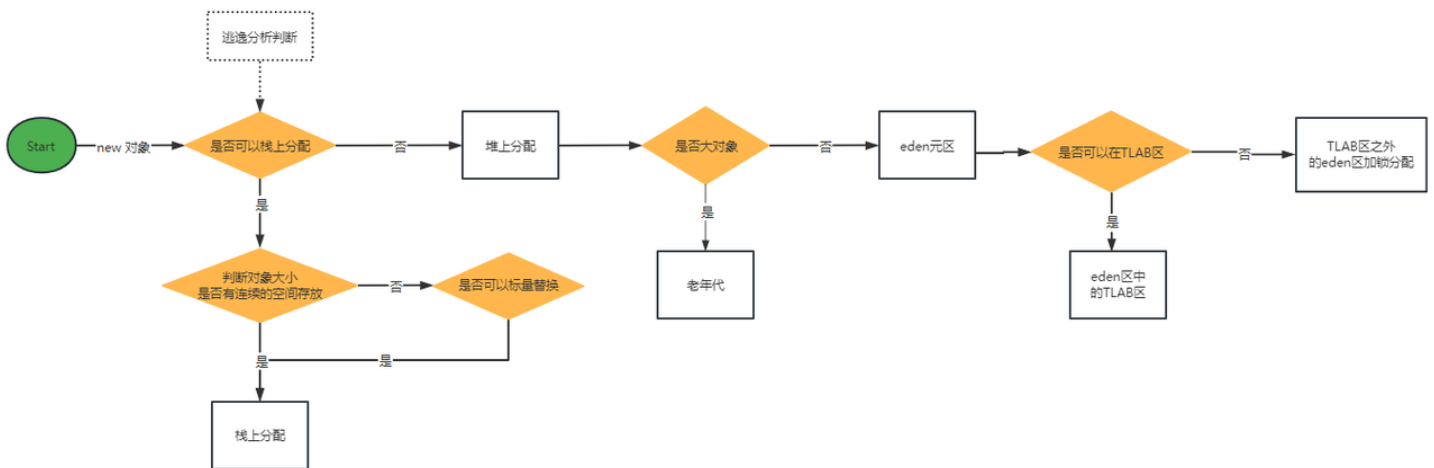
1.5.6 JVM堆的构成是怎样的？

JVM堆主要用于存储我们创建Java对象，从由年轻代(Young 区)和老年代 (Old 区) 构成，年轻代又分伊甸园区 (Eden) 和两个幸存区(s0,s1)。



1.5.7 Java对象分配内存的过程是怎样的？

- 1)编译器通过逃逸分析（JDK8已默认开启），确定对象是在栈上分配还是在堆上分配。
 - 2)如果是在堆上分配，则首先检测是否可在TLAB（Thread Local Allocation Buffer）上直接分配。
 - 3)如果TLAB上无法直接分配则在Eden加锁区(CAS算法进行加锁)进行分配(线程共享区)。
 - 4)如果Eden区无法存储对象，则执行Yong GC（Minor Collection）。
 - 5)如果Yong GC之后Eden区仍然不足以存储对象，则直接分配在老年代。
- 其中,逃逸分析为一种判定方法内创建的对象是否发生了逃逸的一种算法。



1.5.8 JVM年轻代的幸存区设置的比较小会有什么问题？

伊甸园区被回收(GC)时，对象要拷贝到幸存区(s0,s1)，假如幸存区比较小，拷贝的对象比较大，对象就会直接存储到老年代，这样老年代对象多了，就会增加老年代GC的频率（老年代GC一般会触发FullGC-大GC，而FullGC需要的时间的更长）。而分代回收的思想就会被弱化。

说明,GC时,业务线程(用户线程)会暂停(STW-Stop The World),从而影响用户体验.

1.5.9 JVM年轻代的伊甸园区设置的比例比较小会有什么问题？

我们程序中新创建的对象，大部分要存储到伊甸园区(Eden)，假如伊甸园设置的比较小，会增加GC的频率（GC次数越多,GC消耗的时长就会越长），可能会导致STW（Stop The World）的时间变长，进而影响系统性能。

1.5.10 JVM堆内存为什么要分成年轻代和老年代？

为了更好的实现垃圾回收（这里采用的是分代回收思想），减少GC时长、提高其执行效率。（思考GC系统是扫描小块内存比较快还是扫描大块内存速度快）

1.5.11 项目中最大堆和初始堆的大小为什么推荐设置为一样的？

我们在设置JVM初始化堆(-Xms)和最大堆(-Xmx)的大小为一样的目的是，避免程序运行过程中，因对象多少或GC后内存发生了变化而调整堆大小，带来的更大系统开销。在很多大厂的开发规范中都推荐初始堆和最大堆的大小是一样的。（例如阿里的开发手册）

1.5.12 什么情况下对象会存储到老年代？

第一：创建的对象比较大，年轻代没有空间存储这个对象。

第二：经过多次GC，没有被回收的对象年龄在增加，默认15岁后会移动老年代(老年代的对象一般都是些生命力比较顽强的对象)。

1.5.13 Java中所有的对象创建都是在堆上分配内存吗？

随着技术的升级，这个说法现在不准确了。对象还可以分配到栈上了(未逃逸的小对象可以分配在栈上)，栈上分配对象的好处是可以减少GC的次数（栈上分配的对象，在方法执行结束会自动销毁），于此同时用户业务停顿的时间就会减少，用户体验就提高了。

```
1 void create(){
2     byte[] b1=new byte[1]; //对象在方法内部创建，在方法内部使用，这样的对象称之为未逃逸对象
3 }
```

1.5.14 如何理解JVM方法区以及它的构成是怎样的？

方法区（Method Area）是JVM中的一种逻辑上的规范，不同JDK对规范的落地会有不同，例如在JDK8的HotSpot虚拟机中称之为Metaspace（元空间-存储的是元数据，元数据可以理解为描述数据的数据）。方法区主要用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译后的代码等数据。

1.5.15 JDK8中Hotspot虚拟机的方法区内存在哪里？

JVM堆外内存，严格来讲属于操作系统的一部分内存，也可以通过参数设置具体大小，假如没有设置，可以无限增大，直到操作系统内存不足。

例如: JDK8中的Hotspot虚拟机可以通过 -XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M 方式设置元空间的内存大小.

1.5.16 什么是逃逸分析以及可以解决什么问题?

逃逸分析一种数据分析算法, 基于此算法可以检测对象是否发生了逃逸, 未逃逸的小对象可以分配栈上, 也可以进行标量替换, 还可以实现锁消除。总之, 可以有效减少Java对象在堆内存中的分配, 可以减少线程阻塞, 提高其执行效率。

```
1  //-Xms16m -Xmx16m -XX:+DoEscapeAnalysis -XX:+PrintGC
2  class EscapeAnalysisTests{
3      public static void main(String[] args){
4          for(int i=0;i<1000000;i++){
5              alloc1();
6          }
7      }
8
9      static void alloc1(){
10         byte[]b1=new byte[1];//b1变量指向对象发生逃逸了吗?没有
11     }
12
13     static byte[]b2;
14     static void alloc2(){
15         b2=new byte[1];//在方法内部定义的对象,在方法外部有引用,这样对象可以称之为逃逸对象
16     }
17
18     static byte[] alloc3(){
19         byte[]b1=new byte[1];//b1变量指向对象发生逃逸了吗?逃逸了
20         return b1;
21     }
22 }
```



打开分析,可以使用-XX:+DoEscapeAnalysis 这个参数.假如关闭逃逸分析,可以写将DoEscapeAnalysis前面的"+"换成"-".

1.5.17 如何理解对象的标量替换,为什么要进行标量替换?

标量替换首先是依赖于逃逸分析的,假如对象确定可以分配在栈上,但此时栈上有没有连续的内存空间,可以将未逃逸的小对象直接打散(标量替换)分配到栈上,减少堆中对象的创建次数。堆中对象创建的少

了，GC的频率就会降低，GC频率降低了，系统正常业务的执行效率就会提高。

```
1  //-Xms16m -Xmx16m -XX:+DoEscapeAnalysis -XX:+EliminateAllocations -XX:+PrintGC
2  class EliminateAllocationsTests{
3
4      static class Point{
5          int x,y;
6          public Point(int x,int y){
7              this.x=x;
8              this.y=y;
9          }
10     }
11
12     public static void main(String[] args){
13         for(int i=0;i<1000000;i++){
14             alloc();
15         }
16     }
17     static void alloc(){
18         Point p1=new Point(10,20);
19         //存储时的标量(基本数据类型)替换,相当于执行了如下两个步骤
20         //int x=10;
21         //int y=20;
22     }
23
24 }
```



实现标量替换,可以使用 -XX:+DoEscapeAnalysis -XX:+EliminateAllocations 这两个参数

1.5.18 什么是内存溢出以及导致内存溢出的原因？

内存中剩余的内存不足以分配给新的内存请求，此时就会出现内存溢出（OutOfMemoryError）。内存溢出可能直接导致系统崩溃。导致内存溢出的原因可能会有如下几种：

- 创建的对象太大导致堆内存溢出(内存中没有连续的内存空间可以存储你这个大对象)
- 创建的对象太多导致堆内存溢出(对象创建的太多，又不能及时回收这些对象)
- 方法出现了无限递归调用导致栈内存溢出(每次方法的调用都会对应这个一个栈帧对象的创建，同时将栈帧入栈)

- 方法区内存空间不足导致内存溢出。(将如内存中不断的加载新的类，类越来越多，此时可能出现内存溢出)
- 出现大量的内存泄漏



JVM运行时内存中唯一一块不会出现内存溢出的区域是程序计数器.

1.5.19 什么是内存泄漏以及导致内存泄漏的原因？

程序运行时，动态分配的内存空间，在使用完毕后未得到释放，结果导致一直占用着内存单元，直到程序运行结束。这个现象称之为内存泄漏。导致内存泄漏的原因可能有如下几点：

- 大量使用静态变量(静态变量与程序生命周期一样)
- IO/连接资源用完没关闭(记得执行close操作)
- 内部类的使用方式存在问题(实例内部类会默认引用外部类对象)
- ThreadLocal应用不当(用完记得执行remove操作，需要学完线程再进行讲解)
- 缓存(Cache)应用不当(尽量不要使用强引用-我们直接用对象类型定义的变量，一般都可以看成是强引用)

内部类导致内存溢出的案例分析：(这个案例中的内部类用到了多线程，所以案例的演示放在同学们学完线程以后再进行演示)

```
1 package cn.tedu.jvm;
2 class Outer{
3     //实例内部类对象,默认会保存外部类引用
4     class Inner extends Thread{
5         @Override
6         public void run() { //this
7             System.out.println(this); //内部类对象引用
8             System.out.println(Outer.this); //外部类对象应用
9             while(true){}
10        }
11    }
12
13 }
14
15 class StaticOuter{//优化
16     //静态内部类,不会保存对外部类的引用,即便是此对象一直在运行,外部类也可以被销毁。
```

```

17     static class StaticInner extends Thread{
18         @Override
19         public void run() {
20             //System.out.println(Outer.this); //这样是错的
21             while(true){}
22         }
23     }
24 }
25
26 public class OuterInnterTests {
27     static void doInstanceInner(){
28         Outer outer=new Outer();
29         Outer.Inner inner=outer.new Inner(); //内部类对象依赖于外部类对象实例
30         inner.start();
31         outer=null;
32         //outer置为null后,后面就访问不到Outer对象,但是这个对象不会销毁,因为Inner对象一j
33     }
34     static void doStaticInner(){
35         StaticOuter outer=new StaticOuter();
36         StaticOuter.StaticInner inner=new StaticOuter.StaticInner();
37         inner.start();
38         outer=null;
39         //这里的outer置为null后,StaticOuter对象是可以销毁的
40     }
41     public static void main(String[] args) {
42         //doInstanceInner();
43         doStaticInner();
44     }
45 }
46

```

1.5.20 JAVA中的四大引用类型有什么特点？

Java中为了更好地控制对象的生命周期，提高对象对内存的敏感度，设计了四种类型的引用。按其内存中的生命力强弱，可分为强引用（通过引用变量直接引用对象）、软引用（SoftReference）、弱引用（WeakReference）、虚引用（PhantomReference）。其中，“强引用”引用的对象生命力最强，其它引用引用的对象生命力依次递减。JVM的GC系统被触发时，会因对象引用的不同，执行不同的对象回收逻辑。

强引用：此引用引用的对象即便是内存溢出，对象也不会销毁。

```
1 Object o1=new Object();//这里的o1就是强引用
```

Soft引用(SoftReference):此引用引用的对象可以在内存不足（例如触发了FullGC）时会被销毁。

```
1 SoftReference<Object> sr=new SoftReference<Object>(new Object());//sr为软引用
2 sr.get();//获取引用的对象
```

弱引用(WeakReference): 此引用引用的对象可以在GC触发时被销毁。

```
1 WeakReference<Object> sr=new WeakReference<Object>(new Object());//sr为弱引用
```

虚引用(PhantomReference): 此引用对对象进行引用时,对象就相当于没有引用,使用它主要用于记录被销毁的对象,当它引用的对象被销毁时,这个引用就存储到引用队列(ReferenceQueue)

```
1 ReferenceQueue<Object> referenceQueue=new ReferenceQueue<Object>();
2 PhantomReference<Object> sr=
3 new PhantomReference<Object>(new Object(),referenceQueue);//sr为虚引用
```

1.5.21 如何判定对象是否为垃圾？

1、引用计数法

引用计数法，会给每个对象分配一个引用计数器，这个计数器用来记录这个对象的引用数量，当引用数量的值为0时表示，这个对象已经没有任何引用了，此对象就可以被回收了。但是这种方案可能会出现因循环引用问题，进而导致的对象不可回收。

循环引用问题分析，假如现在有两个对象，分别为A和B，他们之间相互引用，但是外界又不能直接访问这两个对象，

但他们的计数器又不为0，此时系统不会认为他们是垃圾对象，就不能回收他们。

A->B(B的计数器为1)

B->A(A的计数器为1)

2、可达性分析法

可达性分析是从根对象(GC root对象)开始,查找它引用的对象,假如某个对象通过GC Root对象不可以直接或间接的访问到,说明这个对象是不可达的。对于不可达对象, JVM认为是垃圾对象,是可以直接被回收的。这种可达性分析方法是目前大多数JVM所采用的一种判定对象是否为垃圾对象的方法。



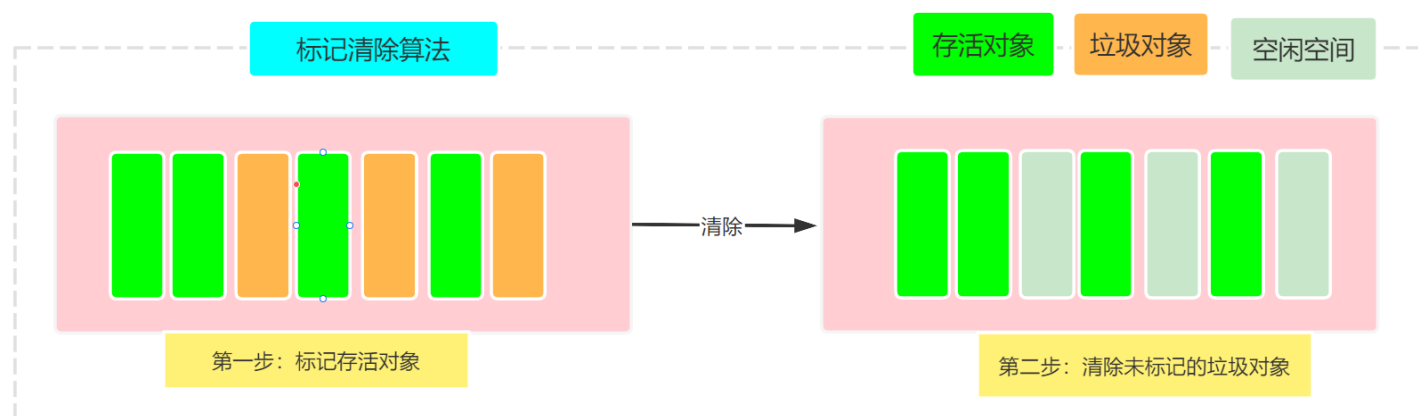
何为GC Root对象呢?

通过实例变量, 类变量, 局部变量可以直接访问到的对象, 都是GC Root对象。

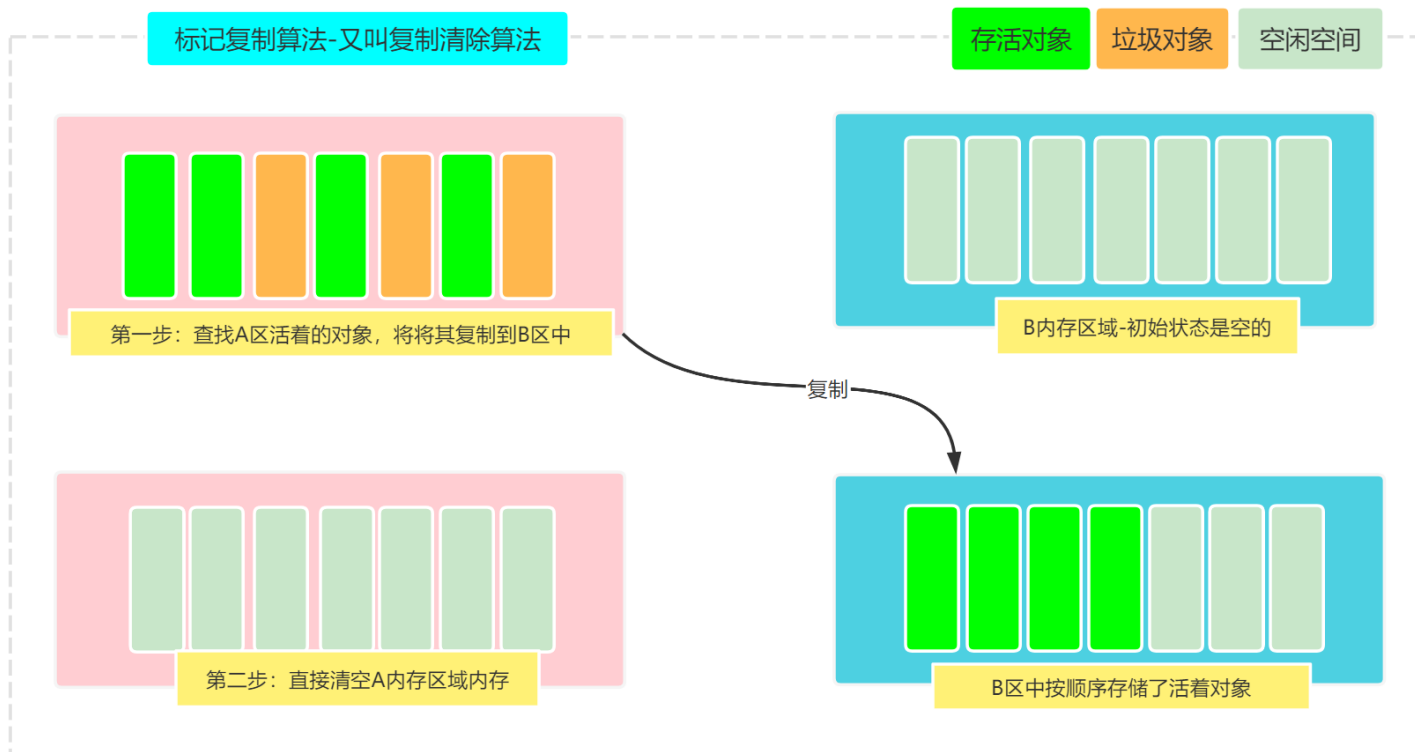
1.5.22 你知道哪些GC算法?

常用GC算法有:标记清除、标记复制、标记整理算法。

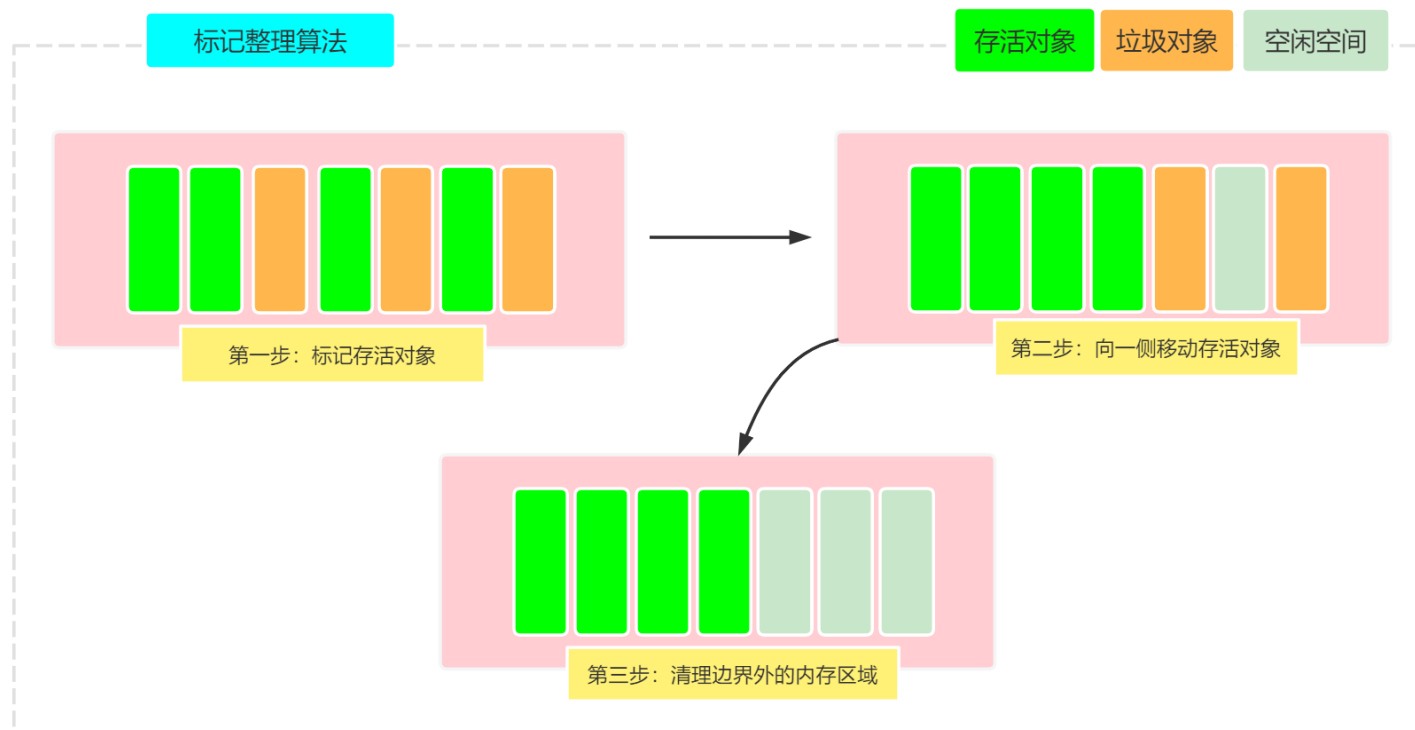
“标记清除法”会首先扫描内存,对内存中活着的对象进行标记,然后再次扫描内存对未标记的对象进行清除,这个算法会扫描两次内存,效率可能会比较低--消耗时间比较长,同时还可能会产生大量碎片。这种算法可以应用于JVM中的老年代,因为老年代GC次数比较少,老年代垃圾对象比较少。



“标记复制(又称之为复制清除法)”这个算法首先会扫描内存,标记活着的对象,并将活着的对象拷贝一块空闲的内存中,最后将原先的内存进行释放,同时也不会产生大量内存碎片。但这种算法会牺牲一定的空间,适合活着的对象比较少的内存区,例如JVM中年轻代-活着的对象少复制的效率就会比较高。



“**标记整理算法**” 首先会扫描内存，找到活着对象，然后将这些对象向一侧移动，最后将边界外的内存进行清空即可。此算法可以考虑应用在老年代。



1.5.23 JVM中有哪些垃圾回收器？

JVM中常用的垃圾回收器(将判定对象是否为垃圾的方法、回收垃圾对象的算法、线程应用策略进行了整合)包括:

1. 串行（Serial）:只有一个线程(GC线程)执行垃圾回收.
2. 并行(Parallel):允许多个线程（GC线程）并行执行垃圾回收.

3. 并发(CMS):并发指的是GC线程执行垃圾回收的同时,可以有业务线程(用户线程)执行业务逻辑.
4. G1(收集器):将原有JVM物理内存连续的分代逻辑打散为逻辑上内存连续的分代区域.

1.5.24 服务频繁fullgc, younggc次数较少, 可能原因?

- 1.经常有超过大对象阈值的对象进入老年代,可以通过-XX:PretenureSizeThreshold设置,大于这个值的参数直接在老年代分配。
- 2.老年代参数设置不当, -XX:CMSInitiatingOccupancyFaction=92设置不合理 (阈值达到多少才进行一次CMS垃圾回), 导致频繁FULLGC
- 3.FULLGC之后没有整理老年代内存碎片, 导致没有连续可用的内存地址, 进入恶性循环, 导致频繁老年代GC, -XX:CMSFullGCsBeforeCompaction可以设置
- 4.新生代过小, 或者e区和s区比例不当, 对象通过动态年龄判断机制频繁进入老年代。
- 5.不合理使用System.gc(), 造成频繁的FullGC, -XX:+DisableExplicitGC这个参数可以禁用System.gc()。
- 6.存在内存泄露, 老年代中驻扎着大量不可回收的对象, 一定程度上缩小了老年代的大小, 造成对象一进入老年代就触发FULLGC
- 7.Meatspace不够用引发fullgc,甚至无限fullgc,这类问题常见于tomcat热部署, 以及使用反射不当。

2. 第二教学月

2.1 数据库设计

2.1.1 什么是实体关系模型 (ER模型)?


实体-关系模型 (Entity-Relationship Model, 简称ER模型) 是一种用于数据库设计的概念工具。它描述了现实世界中各种实体以及它们之间的关系。在ER模型中, 实体表示现实世界中的一个独立对象, 可以是具体的物理对象 (如人、车辆) 或抽象的概念 (如公司、部门)。关系表示实体之间的联系或连接方式, 可以是一对一、一对多或多对多的关系。

2.1.2 如何使用ER图进行数据库设计?

使用ER图进行数据库设计通常包括以下步骤:

1. 确定实体: 识别出系统中存在的主要实体, 并为每个实体确定一个适当的名称。
2. 确定属性: 为每个实体确定相应的属性, 这些属性描述了实体的特征和属性。
3. 确定关系: 确定实体之间的关系, 包括一对一、一对多和多对多的关系。

4. 绘制ER图：根据上述信息，使用矩形框表示实体，使用菱形框表示关系，并使用连线表示实体和关系之间的联系。
5. 优化设计：通过检查ER图，消除冗余、确定主键和外键，并进行必要的优化。

 数据库设计中使用ER模型和ER图可以帮助开发人员更好地理解现实世界的问题域，规划和优化数据库结构，确保数据的一致性和完整性。

2.1.3 说说MySQL数据库中的约束？

在MySQL数据库中，约束用于对表中的数据进行限制和规范，确保数据的完整性和一致性。MySQL支持以下几种类型的约束：

1. 主键约束（Primary Key Constraint）：
 - 主键约束用于唯一标识表中的记录。一个表只能有一个主键，且主键不能包含NULL值。
 - 主键的值必须是唯一的，用于确保表中每条记录的唯一性。
 - 在创建表或修改表结构时，可以给某个字段添加主键约束。
2. 唯一约束（Unique Constraint）：
 - 唯一约束用于确保表中某个字段的取值唯一，同一个表可以有多个唯一约束。
 - 唯一约束允许NULL值，但在表中同一字段中只能有一个NULL值。
3. 非空约束（Not Null Constraint）：
 - 非空约束用于确保表中某个字段的值不能为空。
 - 如果一个字段被指定为非空约束，则在插入或更新数据时，必须提供该字段的非空值。
4. 外键约束（Foreign Key Constraint）：
 - 外键约束用于确保表与表之间的关联完整性。
 - 外键约束建立了两个表之间的关系，并指定了外键与其关联的主键之间的关系。
 - 外键约束可以用于限制表之间的数据操作，如级联更新和级联删除。
5. 默认约束（Default Constraint）：
 - 默认约束用于指定字段在未显示插入值时的默认值。
 - 默认约束在插入新记录时，如果没有为字段提供值，则使用默认值。
6. 检查约束（Check Constraint）：

- 检查约束用于检查指定值是否满足check定义的值范围。



总之，约束是用于保证数据的完整性和一致性，提供了对数据进行约束、验证和自我维护的机制。我们可以在创建表时一起定义约束，也可以在表已创建后通过修改表结构语句来添加。

2.1.4 如何理解SQL注入和预防？

SQL注入是一种常见的安全漏洞，它利用应用程序对用户输入数据的处理不当（未经充分过滤和验证），使得恶意用户可以通过构造恶意的输入数据来破坏SQL语句的结构，执行恶意的SQL代码，从而绕过应用程序的安全机制，获取、修改或删除数据库中的数据。为防止SQL注入攻击，可以采取以下几种措施：

- 使用参数化查询（Prepared Statements）或存储过程（Stored Procedures）来执行SQL语句，在执行之前将用户输入作为参数传递，而不是将其直接拼接到SQL语句中。
- 对用户输入进行充分的过滤和验证，包括输入长度限制、类型检查、白名单过滤等，以过滤掉恶意代码和特殊字符。
- 最小化数据库账号的权限，限制数据库账号的能力，确保其只能执行所需的操作，避免攻击者通过注入获得对整个数据库的完全访问权限。
- 进行安全审计和漏洞扫描，及时发现和修复潜在的安全漏洞。


2.1.5 说说你对数据库连接池的理解？

数据库连接池是一种用于管理和维护数据库连接的技术，旨在提高数据库访问的性能和效率。它通过在应用程序和数据库之间建立一组预先分配的数据库连接，并且这些连接可以被重复使用，而不需要频繁地创建和销毁连接。

数据库连接池的主要优势包括：

1. 提高性能：由于连接池中的连接已经被创建并准备好使用，可以避免频繁地创建和销毁连接的开销，从而提供更快数据库访问速度。
2. 优化资源利用：连接池可以管理连接的数量，根据应用程序的需求动态调整连接的数量，避免连接过多或不足的情况，从而优化数据库资源的利用。

3. 连接可重用：连接池中的连接可以被多个线程或请求重复使用，避免每次请求都需要创建新连接的开销，提高数据库访问的效率。
4. 连接管理和监控：连接池可以提供连接的管理和监控功能，包括连接的创建、验证、保持活动状态、回收和释放等，确保连接的可用性和稳定性。

 常见的数据库连接池实现包括Apache Commons DBCP、C3P0、HikariCP，Druid等，不同的数据库连接池有不同的特性和配置选项，可以根据具体的需求选择合适的连接池进行使用。

2.1.6 说说数据库中的三大设计范式？

数据库设计范式是一组规则和原则，用于规范化数据库模式，确保数据库的结构更加合理、灵活和高效。常用的数据库设计范式包括第一范式（1NF）、第二范式（2NF）、第三范式（3NF）等。

1. 第一范式（1NF）：


- 要求数据库表中的每个字段具有原子性，即每个字段不能再分解成更小的数据项。此外，每个字段必须有一个唯一的名称，确保表中没有重复的数据。

2. 第二范式（2NF）：

- 在满足1NF的基础上，要求表中的非主键字段必须完全依赖于完整主键，而不能依赖于部分主键。换句话说，非主键字段必须与完整主键相关，而不是与部分主键相关。


3. 第三范式（3NF）：

- 在满足2NF的基础上，要求表中的非主键字段之间不存在传递依赖关系。即非主键字段只能依赖于表中的主键字段，而不能依赖于其他非主键字段。

 总之，范式的设计原则是为了减少数据冗余、提高数据一致性和完整性，以及简化数据更新操作。范式的级别越高，数据库模式的结构越规范，但有时可能需要根据实际情况进行灵活处理，根据具体需求权衡范式规则和性能的关系。

2.1.7 说说数据库中的反范式设计？

反范式化的主要思想是在设计数据库时引入冗余数据，以避免频繁的连接操作。这样可以提高查询性能，减少联接操作的开销。此外，反范式化还可以简化复杂的数据模型，使得查询和管理数据变得更加直观和高效。然而，反范式化也存在一些问题和风险，需要谨慎使用。增加冗余数据可能导致数据不一致的问题，因此必须确保在更新数据库时对冗余数据进行正确的处理。此外，反范式化还可能增加存储空间的需求，并且对数据模型的修改和维护会更加困难。

 在实际应用中，需要根据具体情况综合考虑数据库的性能需求、数据访问模式，以及规范化和反范式化带来的优缺点，做出合理的数据库设计决策。

2.1.8 什么是事务的隔离级别？默认隔离级别是什么？

事务的隔离级别指的是多个并发事务之间的相互影响程度。以下是四个常见的事务隔离级别：

1. 读未提交（Read Uncommitted）：事务中的修改可以被其他事务读取。
2. 读已提交（Read Committed）：事务中的修改只能在事务提交后被其他事务读取。
3. 可重复读（Repeatable Read）：在事务执行期间，一个事务内部对同一数据的读取应该是一致的。
4. 串行化（Serializable）：事务按顺序依次执行，是最高级别的隔离性，但可能会降低并发性能。

默认隔离级别是因数据库而异的，常见的默认隔离级别是读已提交（Read Committed）。

2.1.9 JDBC编程的基本步骤是怎样的？

JDBC（Java Database Connectivity）是一种用于在Java应用程序和数据库之间进行连接和交互的API。下面是JDBC编程的基本步骤：

1. 加载数据库驱动程序
2. 建立数据库连接
3. 创建Statement
4. 发送并执行SQL
5. 处理SQL执行结果
6. 关闭数据库连接



JDBC编程的基本步骤，可以根据具体情况和需求进行调整和扩展。

2.2 MyBatis框架应用

2.2.1 说说MyBatis框架的优势？

MyBatis是一个优秀的持久层框架，它与传统的JDBC操作相比，具有以下几个方面的优势：

1. 简化SQL编写：MyBatis提供了强大的SQL映射功能，使用XML或注解的方式，将Java对象与SQL语句进行映射。这样可以减少大量的重复的SQL编写，提高开发效率。
2. 灵活的动态SQL应用：MyBatis允许开发人员通过动态SQL的方式，直接在SQL语句中进行逻辑控制。
3. 参数和结构映射：MyBatis支持灵活的参数和结果映射，可以直接将Java对象作为参数传递给SQL语句，也可以将SQL查询结果直接映射到Java对象中。
4. 缓存支持：MyBatis具有强大的缓存功能，可以对查询结果进行缓存，提高查询性能。同时，还可以手动设置缓存的策略，根据实际需求进行灵活的配置。
5. 易于集成：MyBatis可以与各种主流的开发框架（如Spring和Spring Boot）进行无缝集成。可以通过简单的配置，将MyBatis与其他组件进行整合，提高开发效率和易用性。
6. 易于测试：MyBatis的设计使得数据库操作与Java代码解耦，可以方便地进行单元测试和持续集成。可以使用Mock对象或内存数据库进行测试，提高开发质量和效率。
7. 可扩展性：MyBatis支持插件的扩展机制，可以自定义各种插件来扩展框架的功能。可以对查询结果进行加工处理、监控SQL执行、打印SQL日志等，满足特定的业务需求。



综上所述，MyBatis框架使得开发人员更容易编写和维护数据库访问层的代码，提高开发效率和系统性能。

2.2.2 简单描述下mybaitis 的工作原理

MyBatis 的工作原理可以分为三步：首先，MyBatis 会读取 XML 配置文件或注解配置，初始化 Configuration 对象，并解析 Mapper 接口中的注解或 XML 文件。其次，MyBatis 会根据 Mapper 接口和 SQL 语句生成代理对象，并调用代理对象的方法来执行 SQL 语句。最后，MyBatis 会将查询结果映射成 Java 对象，并返回给调用方。

2.2.3 Mybatis的动态sql标签以及执行原理？

- 1、<if>：用于条件判断
- 2、<choose>、<when>和<otherwise>：用于实现类似于Java中的switch-case语句的逻辑
- 3、<trim>、<where>和<set>：用于处理SQL语句中的空格和条件片段
- 4、<foreach>：用于循环遍历集合

原理：

MyBatis使用OGNL表达式语言来解析动态SQL中的条件表达式，判断是否满足条件，进而决定是否包含对应的SQL片段。通过动态生成SQL语句，MyBatis可以根据不同的条件生成不同的SQL查询语句，并将结果映射到Java对象上。

2.2.4 说说mybatis中\${}和#{ }表达式的异同点？

在MyBatis中，`#{ }` 和 `${ }` 是用于占位符的两种不同方式。

`#{ }` 是安全的占位符，它会将传入的参数进行自动的预编译和转义，防止SQL注入攻击。`#{ }` 主要用于接收参数值，可以在SQL语句中使用。例如：

```
1 <select id="getUserById" resultType="User">
2     SELECT * FROM user WHERE id = #{id}
3 </select>
```

在上面的示例中，`#{id}` 会被MyBatis自动处理，保证传入的 `id` 参数被正确地转义，防止SQL注入。

另一方面，`${ }` 是文字替换占位符，它会将传入的参数直接替换到SQL语句中，不会进行预编译和转义。`${ }` 可以用于在SQL语句中拼接表名、列名等动态的部分。例如：

```
1 <select id="getOrderByColumn" resultType="Order">
2     SELECT * FROM ${tableName} ORDER BY ${columnName}
3 </select>
```

在上面的示例中，`${tableName}` 和 `${columnName}` 会直接替换成传入的参数，需要注意的是，如果这些值是由用户输入的，可能存在SQL注入的风险。



因此，通常情况下，推荐使用 `#{}` 来接收参数值，以防止SQL注入攻击，并在必要的情况下使用 `${}` 进行文字替换。值得注意的是，`#{}` 的预编译特性也有助于提高数据库的性能和查询优化。

2.2.5 说说MyBatis中ResultMap元素作用？

在MyBatis中，ResultMap元素用于定义数据库查询结果与Java对象之间的映射关系。它可以通过配置方式实现对查询结果集的处理，将结果集中的列与Java对象的属性进行对应关系的映射。常用的ResultMap子元素包括：

1. **id**：用于给ResultMap元素指定一个唯一的标识符。
2. **result**：用于配置一个属性映射关系，指定查询结果集中一个列与Java对象的一个属性之间的对应关系。可以使用column子元素指定查询结果集中的列名，使用property子元素指定Java对象的属性名。
3. **association**：用于配置一个关联对象的映射关系。
4. **collection**：用于配置一个关联对象的集合映射关系。



总而言之，MyBatis中的ResultMap元素是实现数据库查询结果与Java对象之间映射的重要配置元素。通过定义和配置ResultMap元素，可以灵活处理数据库查询结果，实现简单的列名与属性名的映射，处理复杂的类型映射，处理复杂关联查询等，提高开发效率和代码的可维护性。

2.2.6 说说Mybatis 中的缓存机制?如何使用？

MyBatis 中的缓存机制是指在执行 SQL 语句时将结果缓存到内存中，下次执行相同 SQL 语句时可以直接从缓存中获取结果，提高查询效率。MyBatis 中的缓存机制分为一级缓存和二级缓存，其中一级缓存是默认开启的，生命周期为 SqlSession 级别，二级缓存需要手动开启，生命周期为 Mapper 级别。在 MyBatis 中，可以使用 cache 标签来配置缓存，也可以使用@CacheNamespace 注解来开启二级缓存。同时，开发者还可以通过自定义缓存来扩展 MyBatis 的缓存机制。

2.2.7 MyBatis框架中应用过哪些设计模式？

MyBatis框架中用到过什么设计模式

- 建造模式(SqlSessionFactoryBuilder, 此对象用于创建SqlSessionFactory对象);
- 简单工厂模式(SqlSessionFactory, 此对象用于创建SqlSession对象);
- 单例模式(ErrorContext, 线程内部单例-底层实现是ThreadLocal)
- 策略模式(缓存淘汰策略-Lru/FIFO)
- 代理模式(为@Mapper注解描述的接口创建动态代理对象)
- 装饰模式(通过CachingExecutor对一级缓存进行添加二级缓存应用)
- 享元模式(连接池-druid,HikariCP/线程池)、
- 桥接模式(驱动程序-JDBC连接数据库需要通过驱动程序-Driver)、
- 适配器模式(日志Log,可以将log4j等日志API转换mybatis中的实现)、。。。。
- 模板方法模式(SqlSessionTemplate, 此对象提供了访问数据库的一些模板方法)
- 组合模式 (将动态SQL中各个元素组合成一个完整的SQL语句)

2.3 Spring框架应用

2.3.1 Spring是一个什么框架?

Spring是一个资源整合框架,其核心是资源整合(空手套白狼),然后以一种更加科学的方式对外提供服务,例如提高对象的应用效率,降低系统开销,提高代码的可维护性等等。其官方网址为spring.io.

2.3.2 Spring 中如何管理bean对象?

在Spring应用程序中,可以使用ApplicationContext来管理和获取Bean。通过加载配置文件或使用注解扫描, Spring会创建和初始化所有的Bean,并将其保存在ApplicationContext中。可以通过getBean()方法来获取已创建的Bean实例。

2.3.3 何理解Spring中IOC设计?

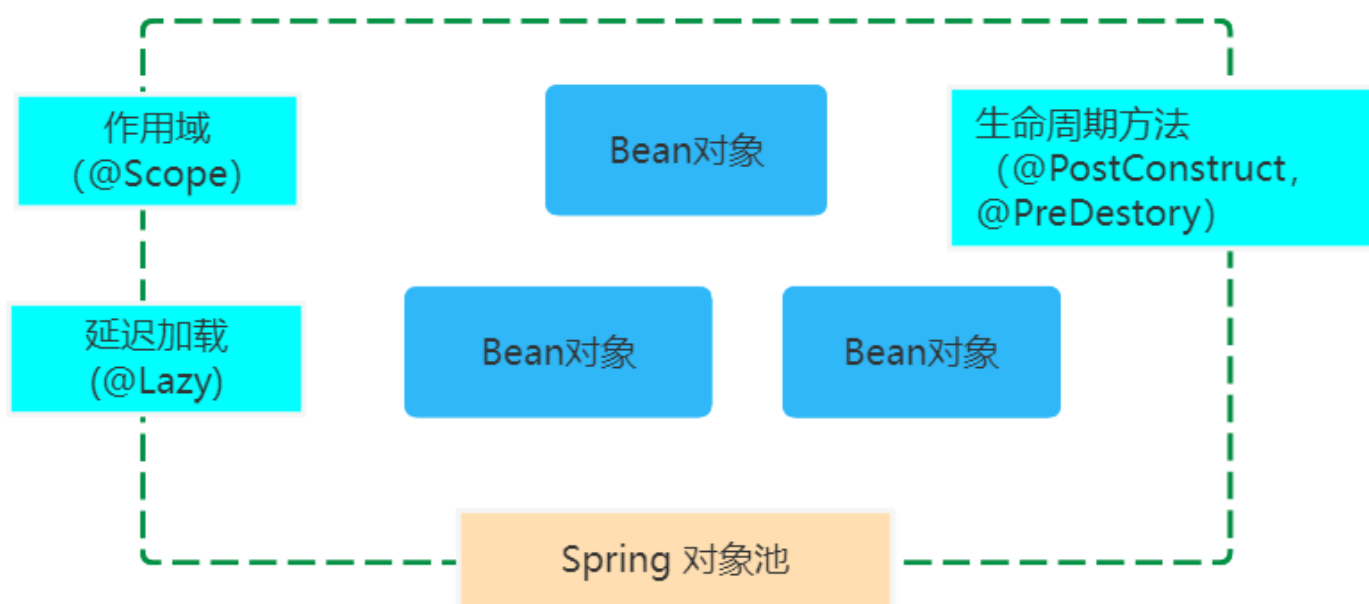
IOC是一种设计思想,我们称之为控制反转,本质上讲解的是对象控制权问题。基于这种设计可以让初学者将对象的控制权转交给第三方,由第三方专业团队管理和应用对象。这样可以更好的避免对象的非正确使用方式,进而更好改善对象在内存中的科学应用。这个IOC设计从生活的角度可以理解为由股票操盘手负责帮你进行资金管理,由父母包办你的婚姻。再简单总结一下的话,IOC可以理解为饭来张口、衣来伸手。

2.3.4 为什么要将对象交给Spring管理?

Spring为我们的对象赋予了很多个更加科学的特性,例如延迟加载,作用域,生命周期方法以及运行时的自动依赖注入(降低耦合,提高程序的可维护性)机制等,基于这些特性可以更好的提高对象的应用性能以及程序的可扩展性。

2.3.5 Spring框架中的Bean有什么特性?

Spring框架为了更加科学的管理和应用Bean对象,为其设计相关特性,例如:懒加载(@Lazy)、作用域(@Scope)以及生命周期方法。



@Lazy注解通常会与@Scope中单例作用域对象结合使用,通过此注解可以告诉系统底层,对象可以延迟创建(何时需要何时创建)。@Scope中指定的作用域为prototype时,每次从容器获取对象,都会创建新的对象。

2.3.6 Spring中依赖注入方式的异同点?

在Spring中,依赖注入的实现主要有三种方式:构造器注入、属性注入和方法注入。

1. 构造函数注入

定义:

构造器注入是通过类的构造方法来实现的,在创建对象的时候,通过构造方法的参数来传递依赖对象。

优点:

对象的依赖关系在创建时就确定,可以保证对象的状态是完全初始化的。

构造函数的参数明确标识了对象所需的依赖,降低了出错的可能性。

缺点：

如果有多个依赖项，构造函数的参数列表会变得很长，不利于维护和扩展。

需要显式地在每次创建对象时传递依赖项。

2. Setter方法注入

定义：

Setter方法注入是通过属性对应的set方法对属性进行赋值。

优点：

对象创建后，可以在任何时候更改依赖项。

在某些情况下，可以允许没有依赖项的情况发生（可选依赖）。

缺点：

对象的依赖关系可能会在不知道的情况下发生更改，导致状态不确定。

需要编写大量的setter方法，增加了代码量和维护成本。

3. 属性注入

定义：

通过反射机制获取属性，并按照指定规则为属性赋值。

优点：

简化了代码，无需编写繁琐的构造函数或setter方法。

可以通过注解自动完成依赖项的注入。

缺点：

对象的依赖关系不易于查看和修改，可能会导致对外部依赖的隐藏与滥用。

不利于单元测试，无法通过构造函数传递模拟依赖



场景选择：

如果依赖关系是必需的，且不希望对象的生命周期内更改依赖项，建议使用构造函数注入

如果依赖关系是可选的，或者需要在对象创建后动态更改依赖项，可以使用setter方法注入

如果对象的依赖关系是确定的，并且不需要进行更改，可以考虑使用字段注入，通过注解实现自动注入

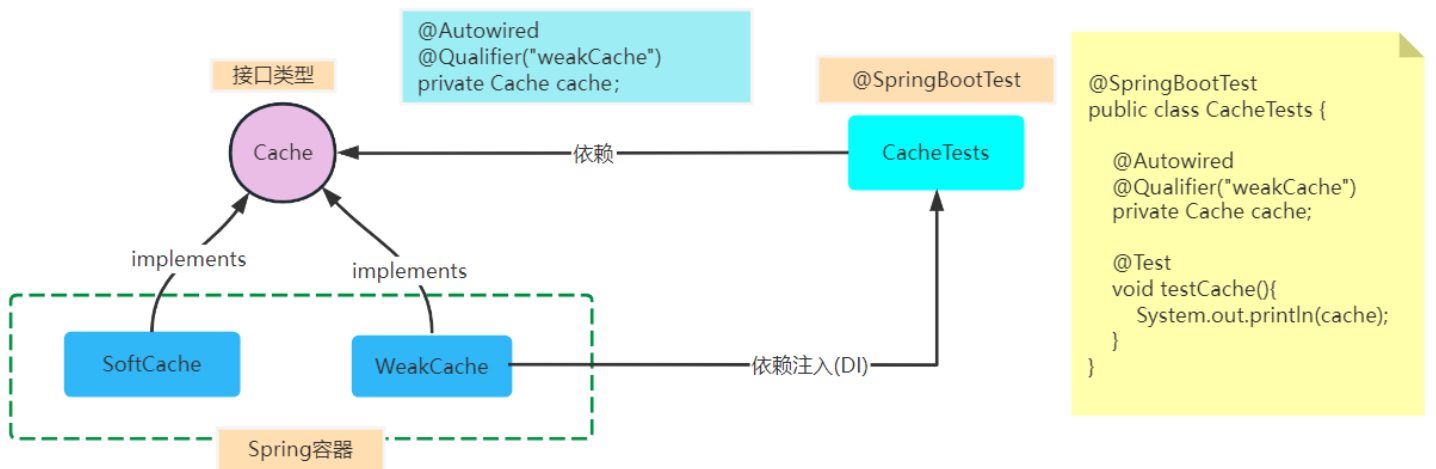
2.3.7 @Autowired注解有什么作用？

@Autowired由spring框架定义，用于描述类中属性或相关方法(例如构造方法)。Spring框架在项目运行时假如发现由他管理的Bean对象中有使用@Autowired注解描述的属性或方法，可以按照指定规则为属性赋值(DI)。

其基本规则是：首先要检测容器中是否有与属性或方法参数类型相匹配的对象，假如有并且只有一个则直接注入。其次，假如检测到有多个，还会按照@Autowired描述的属性或方法参数名查找是否有名字匹配的对象，有则直接注入，没有则抛出异常。最后，假如我们有明确要求，必须要注入类型为指定类型，名字为指定名字的对象还可以使用@Qualifier注解对其属性或参数进行描述(此注解必须配合@Autowired注解使用)。

2.3.8 描述下@Qualifier注解的作用？

@Qualifier注解用于描述属性或方法参数，当有多个相同类型的bean却只有一个需要自动装配时，将@Qualifier 注解和@Autowired 注解结合使用以消除这种混淆，指定需要装配的确切的bean。



2.3.9 说说Spring Boot的优势？

Spring Boot是一个基于Spring框架的开发框架，它的设计目标是简化Spring应用程序的开发和部署。相比传统的Spring应用程序，Spring Boot具有以下几个优势：

1. 简化配置：Spring Boot提供了自动配置的机制，不再需要手动编写大量的配置文件。只需要通过一些约定的方式配置即可，大大减少了开发过程中的配置工作量。
2. 内嵌式容器：Spring Boot内置了Tomcat、Jetty等多种Web容器，可以直接以可执行的JAR包形式运行。不再需要通过外部Web服务器部署应用程序，简化了部署的流程。
3. 开发效率高：提供了很多开箱即用的功能和组件，比如数据访问、事务管理、安全性等，减少了开发人员的重复工作，提高了开发效率。

4. 微服务支持：Spring Boot做为Spring Cloud的基础，提供了丰富的微服务相关的开发和集成功能，如服务注册与发现、负载均衡、分布式配置等。
5. 易于集成：Spring Boot与其他框架和扩展组件的集成非常方便，提供了大量的Starter依赖，可以简单地引入和配置，快速集成第三方库或服务。
6. 自动化管理：Spring Boot提供了一套强大的开发、部署和监控工具，可以进行自动化的构建、测试、部署等，大大简化了应用程序的管理和维护。
7. 强大的生态系统：Spring Boot拥有一个庞大的社区和生态系统，提供了丰富的文档、示例、教程和支持，能够快速解决开发过程中遇到的问题。



总而言之，Spring Boot通过简化配置、内嵌容器、提高开发效率、支持微服务等特性，使得开发人员可以更加专注于业务逻辑的实现，快速构建高效可靠的Spring应用程序。

2.3.10 Spring框架与Spring Boot有什么关系？

Spring Boot是基于Spring 框架的脚手架,其目的是用于简化Spring框架整合资源的过程,在Spring Boot中提供了一些基础依赖,基础配置,通过这些快速实现开箱即用.

2.3.11 Spring boot的启动流程？

Spring Boot的启动流程主要可以分为以下几个步骤：

1. 加载配置文件和初始化环境：

- Spring Boot会根据默认的配置文件名（application.properties或application.yml）加载配置参数。
- Spring Boot会根据环境变量、命令行参数等信息来初始化应用程序的环境。

2. 扫描并加载主配置类：

- Spring Boot会扫描应用程序中的所有类，查找带有@SpringBootApplication注解的主配置类。
- 主配置类是Spring Boot应用程序的入口点，它包含了应用程序的一些核心配置和组件。

3. 创建并配置Spring应用上下文：

- Spring Boot会根据主配置类创建一个Spring应用上下文（ApplicationContext）。
- Spring应用上下文是Spring IoC容器的实例，用于管理和装配Bean。

4. 执行自动配置：


- Spring Boot根据依赖和配置的自动装配规则，自动配置应用程序的各个组件。
- 自动配置将根据类路径中的依赖和配置文件的配置，为应用程序添加所需的主要组件，如数据库连接池、Web服务器等。

5. 执行应用程序：

- Spring Boot会根据配置的Web服务器（如Tomcat、Undertow）创建对应的Servlet容器。
- Servlet容器启动后，Spring Boot会将请求交给Spring MVC框架处理，并返回响应。
- Spring Boot还可以执行其他应用程序逻辑，如定时任务、消息队列等。

6. 关闭应用程序：

- 当应用程序接收到关闭信号（如Ctrl+C）或调用了Spring Boot提供的关闭接口时，应用程序会执行关闭流程。
- Spring Boot会按照事先定义好的顺序，关闭应用程序的相关组件和资源。

 总的来说，Spring Boot的启动流程通过加载配置、扫描配置类、创建应用上下文、自动配置和执行应用程序来实现。它的设计目标是通过约定大于配置的方式，简化和加速Spring应用程序的开发和部署。

2.3.12 Spring Boot自动装配原理？

Spring Boot的自动装配是通过条件化配置和组件扫描实现的。它利用条件注解（@Conditional）和Spring的组件扫描机制来根据运行时环境和配置，自动装配和配置应用程序的各个组件。

Spring Boot自动装配的原理如下：

1. 条件化配置：

- Spring Boot中使用条件注解（@Conditional）来实现条件化配置，如@ConditionalOnClass、@ConditionalOnBean、@ConditionalOnProperty等。
- 条件注解可以根据一定的条件来判断是否要配置某个Bean或执行某个配置项。
- 当条件注解的条件满足时，相应的装配配置会生效。

2. 组件扫描：

- Spring Boot利用Spring的组件扫描机制来自动发现和注册Bean。
- 在启动时，Spring Boot会扫描应用程序中的所有类，查找带有特定注解的类，如@Component、@Service、@Configuration等。
- 扫描到的类会被自动注册为Bean，可以通过@Autowired等注解来注入并使用。

3. 自动配置：

- Spring Boot通过条件化配置和组件扫描，将自动配置的Bean注册到Spring应用上下文中。
- 自动配置会根据依赖和配置的规则，自动添加所需的组件，如数据库连接池、Web服务器等。
- 自动配置会根据classpath下的各个依赖来判断应用程序的需要，并根据相关的条件注解来判断是否启用相关的自动配置。



总的来说，Spring Boot的自动装配利用条件化配置和组件扫描，根据运行时环境和配置的条件，自动装配和配置应用程序的各个组件。这样可以大大简化Spring应用程序的配置，提高开发效率和部署速度。同时Spring Boot还提供了大量的可配置项，可以根据需要进行个性化的定制。

2.3.13 说说对Spring Starter的理解？

Spring Starter是Spring Boot的一个特性，它是一种快速启动和快速开发的方式。Spring Starter是一种用于初始化和配置Spring Boot项目的依赖管理工具，它可以帮助开发人员更快速地创建和配置Spring Boot项目。

具体来说，Spring Starter提供了一系列预定义的项目起步依赖（Starter Dependencies），这些依赖包含了构建一个具有特定功能的Spring Boot项目所需的所有依赖。开发人员只需要选择适合自己需求的起步依赖，就能自动引入和配置相关的依赖和配置。

Spring Starter的优点和作用如下：

1. 简化项目配置：Spring Starter提供了一系列的Starter Dependencies，开发人员只需添加相应的起步依赖，就能自动引入和配置项目所需的依赖和配置，无需手工配置繁琐的依赖关系。
2. 提升开发效率：Spring Starter为常用的功能提供了快速启动和快速开发的能力，开发人员可以更快速地构建出一个可以运行的、具备特定功能的Spring Boot应用。
3. 约定优于配置：Spring Starter提供了一套约定规范，开发人员按照这些规范组织项目结构和定义Bean，框架会自动根据这些约定进行配置和装配，大大减少了配置的工作量。
4. 降低项目风险：Spring Starter经过了大量的测试和验证，所选用的依赖版本和配置都是经过官方推荐的，这些Starter Dependencies可以保证良好的兼容性和稳定性。



总的来说，Spring Starter是Spring Boot极具特色的一个功能，它为开发人员提供了一种简单、快速、约定的方式来初始化和配置Spring Boot项目，大大减少了开发的复杂性和风险，同时提升了开发效率和项目质量。

2.3.14 说说你对Spring MVC的理解？

Spring MVC是基于Spring框架的一种轻量级的Web开发框架，它提供了一系列的组件和工具，用于简化开发Web应用程序的开发过程。Spring MVC采用经典的MVC设计模式，将应用程序分为三个核心部分，分别为模型、视图和控制器。模型负责处理业务逻辑和数据的封装，视图负责展示数据给用户，控制器负责处理用户请求、协调模型和视图。这样可以将应用程序分为多个层次来促进良好的代码结构。其核心功能包括：

1. 强大的请求处理机制：Spring MVC提供了灵活请求处理机制，可以使用注解或XML配置来映射URL和方法。
2. 视图解析和渲染：Spring MVC支持多种视图技术，如JSP、Thymeleaf、Freemarker等。
3. 强大的表单处理：Spring MVC提供了强大的表单处理支持，包括表单验证、数据绑定和类型转换。它可以自动将请求参数映射到方法的参数上，并提供数据校验和转换功能，大大简化了表单处理的过程。
4. 集成测试支持：Spring MVC提供了丰富的集成测试支持，可以通过模拟HTTP请求和断言结果来测试控制器的行为。这使得开发人员可以方便地编写自动化测试来验证系统的功能和逻辑。



总之，Spring MVC是一个功能强大、灵活易用的Web开发框架，它可以帮助开发人员快速构建可扩展、可维护的Web应用程序。

2.3.15 说说Spring MVC的请求处理流程？

Spring MVC的请求处理流程主要包括以下几个步骤：

1. 用户发送请求：客户端通过浏览器发送一个HTTP请求到服务器。
2. DispatcherServlet接收请求：DispatcherServlet是Spring MVC的中央控制器，它接收所有的HTTP请求，并负责将请求交给合适的处理器进行处理。
3. 处理器映射器（HandlerMapping）匹配处理器：处理器映射器根据请求的URL和其他条件，将请求映射到对应的处理器（Controller）。

4. 处理器适配器（HandlerAdapter）处理请求：处理器适配器根据处理器的类型和实现，调用合适的处理器方法进行处理，并返回ModelAndView对象。
5. 执行处理器方法：处理器方法根据业务逻辑处理请求，并将处理结果封装到ModelAndView对象中。处理器方法可以通过注解或XML配置来定义。在处理器方法中可以访问请求参数、会话参数等等。
6. 视图解析器（ViewResolver）解析视图：视图解析器根据逻辑视图名称解析成实际的视图对象。它可以根据配置的前缀和后缀等规则进行视图解析，并返回相应的视图对象。
7. 视图渲染器（ViewRenderer）渲染视图：视图渲染器将处理结果（Model数据）渲染到视图上，生成最终的响应结果。视图可以是JSP、Thymeleaf、Freemarker等等。
8. 返回响应结果：DispatcherServlet将最终的响应结果返回给客户端，完成整个请求处理过程。



需要注意的是，Spring MVC还提供了一些拦截器（Interceptor）和异常处理器（HandlerExceptionResolver）等组件，用于在请求处理过程中进行拦截和异常处理。这些组件可以用来实现日志记录、权限验证、异常处理等功能，增强了请求处理的灵活性和扩展性。

2.3.16 说说你对Spring AOP的理解？

Spring AOP（Aspect-Oriented Programming）是Spring框架中的一个核心模块，用于实现面向切面编程的技术。面向切面编程是一种编程范式，通过将横跨多个组件和关注点的功能进行封装和解耦，提高代码的可重用性、可维护性和可测试性。

在Spring AOP中，可以通过声明式的方式将一些横切关注点（cross-cutting concern）应用到模块化的代码中，而不需要修改原始代码。这些关注点可以在应用程序的不同层次中，比如日志记录、事务管理、安全性等。

Spring AOP使用切面（aspect）来定义横切关注点以及它们在目标对象中的应用方式。切面由切点（pointcut）和通知（advice）组成。切点定义了目标对象中哪些方法会被应用横切关注点，通知定义了切点被触发时要执行的逻辑。

通知可以分为以下几种类型：

- 前置通知（Before advice）：在目标方法执行之前执行。
- 后置通知（After advice）：在目标方法执行之后（无论是否出现异常）执行。
- 返回通知（After returning advice）：在目标方法正常返回之后执行。
- 异常通知（After throwing advice）：在目标方法抛出异常之后执行。
- 环绕通知（Around advice）：在目标方法执行前和执行后都可以执行自定义逻辑。

Spring AOP可以通过配置文件（XML）或注解的方式进行配置和管理，使得开发者可以将关注点与业务逻辑分离，提高代码的可维护性和可测试性。

总而言之，Spring AOP是一种实现面向切面编程的技术，通过将横切关注点应用到模块化的代码中，提供了更高层次的代码重用和解耦。

2.3.17 Spring中常见哪些设计模式有哪些？

1. 建造模式：BeanDefinitionBuilder (通过此对象构建BeanDefinition对象)
2. 简单工厂模式:BeanFactory。(用于构建Bean对象的工厂)
3. 工厂方法模式:ProxyFactoryBean。(通过Bean对象中的方法构建ProxyFactory)
4. 单例模式: 单例Bean对象。(作用域为singleton的bean对象)
5. 代理模式：Aop Proxy。(JDK代理,CGLIB代理)
6. 策略模式：AOP代理策略, SimpleInstantiationStrategy。
7. 适配器模式: HandlerAdapter,AdvisorAdapter。
8. 模版方法模式:JdbcTemplate。(对JDBC的操作进行了封装,提供了一些访问数据库的模板方法)
9. 责任链模式：HandlerInterceptor。(SpringMVC中的拦截器,控制对Controller对象方法的方法)
10. 观察者模式: ApplicationListener(监听服务的启动,销毁,...)

2.4 通讯协议应用

2.4.1 什么是HTTP协议？

HTTP（Hypertext Transfer Protocol，超文本传输协议）是一种用于在网页浏览器和Web服务器之间传输数据的协议。它是建立在TCP/IP协议之上的应用层协议，用于在客户端和服务端之间传输超文本（如HTML、XML）和其他资源（如图片、视频等）。

2.4.2 HTTP协议有什么特点？

HTTP（Hypertext Transfer Protocol，超文本传输协议）是一种基于客户端-服务器架构的应用层协议，用于在Web浏览器和Web服务器之间传输数据。HTTP协议具有以下特点：

1. **无状态性 (Stateless)**：HTTP协议本身不会保持客户端和服务端之间的状态。每个请求都是相互独立的，服务器不能识别出两个请求是否来自同一个用户。这意味着在处理每个请求时，服务器不会记住之前的请求信息。为了解决这个问题，引入了Cookie和Session等机制来实现状态管理。
2. **可扩展性 (Extensible)**：HTTP协议的请求和响应头是可扩展的。这意味着可以通过添加自定义的首部字段来实现特定的功能。例如，可以添加身份验证、缓存控制、内容协商等功能。
3. **请求-响应模型**：HTTP协议采用请求-响应模型，客户端向服务器发送HTTP请求，服务器接收请求并处理，然后将响应发送回客户端。请求和响应都由起始行 (start line)、首部 (header) 和消息体 (message body) 组成。
4. **支持多种数据格式**：HTTP协议可以传输多种类型的数据，如HTML、XML、JSON等。根据Content-Type首部字段指定的数据类型，客户端和服务端可以采用相应的方式来处理数据。



总之，这些特点使得HTTP协议成为Web应用程序通信的标准协议，并广泛应用于互联网上的各种Web服务和资源传输。

2.4.3 HTTP协议的工作原理是怎样的？

HTTP协议基于客户端-服务器模型，客户端向服务器发送HTTP请求，服务器接收并处理该请求，然后将响应发送回客户端。具体工作原理如下：

1. **建立连接**：客户端通过创建一个TCP连接与服务器建立通信。默认情况下，HTTP使用TCP的80端口进行通信。也可以使用HTTPS进行加密通信，此时使用的端口是443。
2. **发送请求**：客户端构造HTTP请求，包括请求方法（如GET、POST）、请求URI（统一资源标识符）、HTTP版本号、请求头（包含各种首部字段），以及可选的请求消息体。然后将请求发送给服务器。
3. **处理请求**：服务器接收到客户端的请求后，根据请求中的URI和方法等信息来处理请求。根据请求的资源类型、是否需要身份验证等，服务器可能进行各种操作，如返回网页内容、执行服务器端脚本、查询数据库等。
4. **发送响应**：服务器构造HTTP响应，包括状态行（包含HTTP版本号和状态码）、响应头（包含各种首部字段），以及可选的响应消息体。然后将响应发送回客户端。
5. **处理响应**：客户端接收到服务器的响应后，根据状态码判断请求的处理结果。根据响应头中的信息，客户端可能进行各种操作，如解析网页内容、下载文件、处理响应数据等。

6. **断开连接：**一次HTTP请求-响应完成后，客户端和服务端之间的TCP连接可以根据需要继续保持或关闭。



总之，HTTP在互联网上广泛应用，为浏览器和服务端之间的数据传输提供了一种可靠的通信机制。

2.4.4 Get和Post请求的异同点？

GET和POST是HTTP请求方法，用于客户端向服务器发送请求。它们之间的主要区别如下：

1. 参数传递方式：GET请求将参数包含在URL的查询字符串中，例如：<http://example.com/search?keyword=value>。而POST请求将参数放在请求的消息体中，不会直接暴露在URL中。
2. 安全性：GET请求的参数会被保存在浏览器的历史记录、服务器的日志中，甚至可以被书签保存。因此，GET请求在传输敏感信息时不安全。POST请求不会把参数保存在这些地方，相对更安全。
3. 参数长度限制：GET请求对参数长度有限制，不同浏览器和服务端有不同的限制。而POST请求则没有特定的长度限制，但实际上仍然存在服务端端的限制。
4. 幂等性：GET请求是幂等的，即多次执行相同的GET请求，服务器的状态和响应结果不会改变。而POST请求是非幂等的，多次执行相同的POST请求，可能导致服务器状态改变，或者多次创建相同的资源。
5. 缓存机制：GET请求可以被浏览器缓存，下次再请求相同的URL时可以直接读取缓存。而POST请求默认情况下不会被缓存。
6. 用途：GET请求主要用于获取资源，不应该有副作用。而POST请求主要用于提交数据，可能有副作用，如创建、更新或删除资源。




总之，需要根据具体的业务需求和安全要求选择使用GET或POST请求方法。一般来说，GET请求适合使用在获取数据的场景，而POST请求适合使用在提交数据的场景。

2.4.5 请求转发和重定向的区别？

请求转发（forwarding）和重定向（redirecting）是Web开发中常用的两种跳转方式，它们之间有区别：

1. **定义：** 请求转发是服务器端进行的跳转操作，通过服务器将请求发送给另一个资源处理。而重定向是在客户端进行的跳转操作，由服务器返回特定的响应码和URL，然后客户端根据响应码和URL重新发送请求。
2. **执行位置：** 请求转发是在服务器内部完成的，客户端并不感知跳转动作，URL不会发生变化。重定向是由服务器返回响应码和URL给客户端，然后客户端根据此响应码和URL重新发起请求。
3. **数据传递：** 请求转发可以在服务器内部传递数据，请求和转发的资源共享相同的请求和响应对象。重定向是两次请求，因此无法在客户端直接传递数据，需要通过URL参数、Session等方式进行传递。
4. **地址栏显示：** 请求转发时，地址栏中的URL不会发生变化，用户无法直接看到转发后的URL。而重定向时，地址栏中会显示重定向后的URL。
5. **处理方式：** 请求转发用于在服务器内部跳转资源，适用于不同的Servlet或JSP之间的跳转。重定向用于客户端跳转，适用于不同的网页之间跳转，或者在处理POST请求后防止重复提交。
6. **SEO优化：** 请求转发保持原有URL不变，更有利于搜索引擎优化（SEO）。而重定向可改变URL，对于搜索引擎而言是一个新的URL。

 因此，请求转发主要用于在服务器内部的资源跳转，处理速度快，数据共享方便；而重定向主要用于在客户端之间跳转，地址栏显示不同，适合处理不同网页之间的跳转和防止重复提交等场景。具体使用哪种方式需要根据业务需求和实际情况来决定。

2.5 数据库技术拓展

2.5.1 说说MySQL中死锁？

在MySQL中，死锁指的是两个或多个事务在相互等待对方所持有的资源时发生的一种情况，导致它们无法继续执行下去。当发生死锁时，MySQL会自动选择一个事务作为牺牲者，并回滚该事务，以解除死锁。

2.5.2 MySQL中有哪些策略可以减少死锁的发生？

为了减少死锁的发生，可以采取以下几种策略：

- 合理设计数据库模式和应用程序，避免多个事务同时访问并修改相同的数据。
- 保持事务的执行时间尽可能短，减少资源占用的时间。
- 尽量按照相同的顺序获取锁，以避免事务之间产生循环等待。
- 使用合适的索引和查询优化，减少锁的竞争和冲突。
- 在出现死锁时，尽快检测和解决死锁问题，避免影响系统的正常运行。

综上所述，死锁是在MySQL中可能发生的一种情况，可以通过合理的设计和调优来降低死锁的概率，并且MySQL会通过自动回滚事务来解除死锁。

2.5.3 如何理解数据库中的索引？

索引是一种数据结构，用于快速查找和访问数据库中的数据。它类似于书籍的目录，可以根据关键字快速定位到存储在数据库中的特定数据行。

索引的建立需要根据具体的业务需求和查询频率合理选择索引列，通常选择那些经常用于查询条件和连接条件的列作为索引列。但是，过多的索引和不合理的索引设计会增加存储空间和写操作的开销，同时也可能降低更新操作的性能。



在设计和使用索引时需要综合考虑数据的读写比例、查询的频率和范围、数据的关联性以及数据库管理系统的特性等因素。合理地使用索引，可以显著提升数据库的查询性能，并有效地优化系统的整体性能。

2.5.4 如何选择并创建合适的索引以提高查询性能？

在数据库中，索引是用来加快查询速度的重要手段。正确选择并创建合适的索引可以显著提高数据库的查询性能。以下是一些选择和创建索引的指南：

1. 选择需要索引的列：在选择需要索引的列时，应该优先考虑那些经常被用作查询条件的列或参与JOIN操作的列。例如主键列、外键列和经常用于过滤的列等。
2. 选择索引类型：在创建索引时，一般可以选择B-tree索引和哈希索引。B-tree索引适用于范围查询或特定列上的排序查询，而哈希索引适用于等值查询。
3. 创建唯一索引：在保证数据完整性的前提下，应尽可能创建唯一索引，因为唯一索引可以避免重复数据的出现，同时还可以加快查询速度。
4. 避免创建过多索引：创建过多索引会增加查询语句的执行时间和存储空间，因此需要避免不必要的冗余索引。
5. 优化覆盖索引：覆盖索引是指查询结果可以直接从索引中获取，而无需再去原表中查询。通过优化覆盖索引，可以极大地提高查询效率。
6. 定期更新和优化索引：随着数据的增多和修改，索引的使用效果可能会逐渐下降，因此需要定期更新和优化索引，以确保查询性能始终得到最佳状态。



总之，正确地选择和创建索引可以提高查询速度和性能，但需要尽量避免过度索引，定期更新和优化索引以保持最佳状态。

2.5.5 如何对SQL进行调优？

SQL调优是通过优化SQL语句的执行效率来提高数据库查询和操作的性能。一些常见的SQL调优技巧让如下：

1. 优化语句结构：合理设计和编写SQL查询语句，尽量避免不必要的查询和连接操作，减少SQL语句的复杂度。
2. 使用合适的索引：根据查询条件和数据访问模式，为表添加适当的索引，以提高查询性能。但要注意，过多或不恰当的索引也会影响性能。
3. 使用适当的数据类型：选择合适的数据类型，避免过多的数据转换和类型转换的开销。
4. 优化表结构和查询逻辑：合理设计表结构，规范字段类型和长度。优化查询逻辑，避免重复查询和不必要的联接操作。
5. 批量操作和批量提交：对于大量的数据操作，尽量使用批量操作和批量提交，减少与数据库的交互次数。
6. 数据库性能监控与调优工具：使用数据库性能监控工具来诊断和分析SQL语句的执行情况，找出慢查询和瓶颈，有针对性地进行优化。
7. 数据库缓存：合理设置数据库的缓存大小和缓存策略，尽量提高缓存的命中率，减少磁盘IO的开销。
8. 并发控制与锁机制：合理选择并发控制和锁机制，避免长时间的锁等待和死锁现象。
9. 定期维护和优化数据库：定期进行数据库的维护和优化，包括索引重建、统计信息更新、表碎片整理等操作。




SQL调优是一个综合性的工作，需要根据具体的数据库、应用场景和性能问题来进行调整和优化。在进行SQL调优时，可以结合使用数据库的性能监控工具和分析工具，分析和诊断SQL语句的执行计划和性能瓶颈，并进行有针对性的优化。

2.5.6 如何理解数据库中的存储过程？

存储过程（Stored Procedure）是一组预先编译好的SQL语句集合，它将一系列的SQL语句封装在一起，形成一个可被多次调用的数据库对象，存储在数据库中。一般用于实现复杂的业务逻辑、减少客户端与数据库的交互次数、提高数据库性能和安全性。

存储过程的优点：

1. 代码复用：存储过程可以被多次调用，减少了编写相同代码的工作量，提高了开发效率。
2. 性能优化：存储过程在服务器端执行，减少了网络传输的开销，可以显著提高查询和数据操作的性能。
3. 安全性：通过存储过程，可以限制对数据库的直接访问，只通过存储过程提供对数据库的操作，有效地保护数据库的安全性。
4. 维护方便：存储过程的代码存储在数据库中，可以随时修改和维护，不需要修改应用程序代码。

 存储过程可以在数据库管理系统中创建、修改和删除。在使用存储过程时，可以传递参数进行输入和输出操作，并通过调用存储过程来执行相应的数据库操作。存储过程可以用于实现复杂的数据计算、数据操作和业务逻辑等场景。但是，存储过程的设计和使用需要谨慎，过多的存储过程和过于复杂的逻辑可能会导致维护和调试困难，降低系统的可维护性。因此，在设计和使用存储过程时，需要根据具体的应用需求和性能要求进行合理的规划和设计。

2.5.7 如何理解数据库中的函数？

数据库中的函数（Functions）是一种可被调用的数据库对象，用于实现数据的计算、处理和转换等功能。与存储过程类似，函数也是一组预先定义好的SQL语句集合，但函数通常返回一个值作为结果，而不像存储过程可以执行一系列的操作。函数可以接受参数，对参数进行处理后返回计算结果。

函数分类：

1. 系统函数：也称为内建函数，是数据库管理系统提供的一组内置函数，用于执行各种常见的数据计算和处理操作，如字符串操作、数学运算、日期处理、聚合运算等。系统函数可以直接在SQL语句中调用和使用。
2. 用户定义函数：也称为自定义函数，是用户根据具体需求自己定义的函数。用户定义函数可以在数据库中创建和使用，类似于自定义存储过程。用户定义函数可以接受输入参数，并返回一个结果。

函数的优点：

1. 代码复用：函数可以被多次调用，减少了编写相同代码的工作量，提高了开发效率。
2. 简化操作：函数封装了一系列数据处理的逻辑，可以简化复杂的数据计算和处理操作。
3. 提高性能：函数一般在数据库服务器端执行，减少了网络传输开销，可以提高查询和数据处理的性能。



函数在数据库中的创建、修改和删除与存储过程类似。可以通过定义和使用函数，实现在数据库中的一些常见的数据计算、处理和转换功能。在使用函数时，可以传递参数进行输入和输出操作，并通过调用函数来获取所需的计算结果。

2.5.8 什么是时序数据库？

时序数据库（Time Series Database）是一种专门用于处理时间序列数据的数据库。时间序列数据是指按照时间顺序排列的数据，如传感器数据、日志数据、股票价格、天气数据等。时序数据库具有以下特点：

1. 高性能存储和查询：时序数据库针对时间序列数据的存储和查询进行了优化，能够快速写入和查询大量的时间序列数据。
2. 时间索引和优化：时序数据库通常采用时间作为主要索引，可以快速定位和访问特定时间段的数据，提高查询效率。
3. 数据压缩和存储优化：时序数据通常存在大量的重复和周期性数据，时序数据库采用压缩算法和存储优化技术，降低存储成本。
4. 数据保留策略：时序数据库支持根据时间保留数据的策略，可以自动删除过期的数据，以减小存储空间占用。
5. 可扩展性和高可用性：时序数据库支持数据分片和分布式存储，以满足大规模数据的存储和处理需求。同时，时序数据库还支持数据的冗余备份和高可用性架构，保障数据的安全和可靠性。

常见的时序数据库包括：

1. InfluxDB：一种开源的时序数据库，具有高性能、易于扩展和灵活的数据模型。
2. TimescaleDB：建立在PostgreSQL之上的开源时序数据库，提供标准SQL查询支持和水平扩展能力。
3. Prometheus：主要用于监控和度量数据的开源时序数据库，具有强大的标签查询和数据可视化功能。
4. OpenTSDB：基于HBase的开源时序数据库，适用于海量时间序列数据的存储和查询。
5. KairosDB：建立在Cassandra之上的开源时序数据库，具有高可扩展性和高性能。

这些时序数据库各有特点和适用场景，可以根据具体需求和规模选择合适的时序数据库。

2.5.9 http和https的区别？

HTTP（Hypertext Transfer Protocol，超文本传输协议）和HTTPS（HTTP Secure，安全超文本传输协议）是网络传输协议的两种不同形式，它们之间的区别主要在于以下几个方面：

1. 安全性：

- HTTP不提供任何加密机制，通信数据以明文形式传输，安全性较低，容易被第三方窃听、篡改和劫持。
- HTTPS使用SSL/TLS协议进行通信，通过加密传输数据，能有效防止信息被窃听和篡改，提供更高的安全性。

2. 数据传输方式：

- HTTP使用明文传输数据，传输速度较快，但容易被监听和篡改。
- HTTPS使用加密机制，通过SSL/TLS加密传输数据，保证了数据的完整性和安全性，但传输速度会稍微慢一些。

3. 端口：

- HTTP默认使用端口80进行通信。
- HTTPS默认使用端口443进行通信。

4. 证书：

- HTTPS需要使用数字证书来验证服务器的身份，并确保通信的安全性。
- HTTP不需要证书验证，通信安全性无法得到保证。

5. SEO优化：

- 对于搜索引擎而言，由于HTTPS具有更高的安全性，更受欢迎。因此，使用HTTPS协议更有利于SEO（搜索引擎优化）。



综上所述，HTTP适用于一般的数据传输场景，而HTTPS适用于对数据传输安全性要求较高的场景，如进行网上支付、传输敏感信息等。在保护数据安全和防止信息泄露方面，HTTPS比HTTP更可靠。

3. 第三教学月

3.1 前端部分

3.1.1 HTML5 中的一些新特性有哪些？

HTML5 引入了许多新特性，包括语义元素（如 <header>、<nav>、<section> 等）、多媒体支持（<audio>、<video>）、新表单控件（<input type="date">、<input type="email"> 等）以及 Canvas 绘图等。

3.1.2 如何实现响应式设计？

响应式设计是通过使用 CSS 媒体查询（@media）来适应不同设备和屏幕尺寸。通过设置不同的样式规则，可以使网页在不同分辨率下有不同的布局和外观

3.1.3 解释一下事件冒泡和事件捕获？

事件冒泡是指事件从触发事件的元素开始，然后向上冒泡到其祖先元素。事件捕获是指事件从最外层的元素开始，然后传递到具体的元素

3.1.4 如何处理DOM事件？

```
// 选择要处理事件的元素
const button = document.querySelector('#myButton');

// 绑定点击事件处理程序
button.addEventListener('click', function(event) {

    // 在事件处理函数中编写要执行的操作
    alert('按钮被点击了！');

});
```

3.1.5 Cookie和Session的区别？

- 1、存储位置：Cookie 存储在客户端浏览器中，而 Session 存储在服务器上
- 2、容量限制：Cookie 适合存储小量的、不敏感的数据，常用于用户标识、记住登录状态等，Session 可以存储更大量的数据，通常用于存储用户的会话状态
- 3、隐私和安全性：由于 Cookie 存储在客户端，可能受到隐私和安全问题。Session 存储在服务器上，相对更安全
- 4、生命周期：可以设置 Cookie 的过期时间，可以是会话级的（浏览器关闭后删除）或持久性的（在过期时间之前有效），Session 的持续时间取决于用户的活动，当用户关闭浏览器或者会话超时，Session 数据会被销毁

3.1.6 Vue 的双向数据绑定是如何实现的？

通过将数据属性与视图建立关联，并在数据发生变化时自动更新视图，同时在用户与页面交互时，也能反向更新数据，从而实现了数据与视图之间的双向同步。

这个过程是通过 Vue 的响应式系统和虚拟 DOM 技术来实现的。这使得开发者不需要手动处理 DOM 操作，而能够专注于处理数据和交互逻辑

3.1.7 ElementUI 的表单验证是如何实现的？

ElementUI 提供了一种基于规则的表单验证机制，您可以使用 el-form、el-form-item 和 rules 属性来实现。规则可以通过定义验证函数、正则表达式等方式来进行自定义

3.1.8 Axios 如何处理响应拦截器?

Axios 允许您在请求返回之前或之后,对请求和响应进行拦截和处理。您可以使用 `axios.interceptors` 对象来添加请求和响应拦截器,以执行全局的预处理或后处理操作

3.2 Git部分

3.2.1 版本控制用的什么?

用的git,虽然我们团队不大,但是代码仓库公司管理还是比较规范的,主要是线上的代码和本地的代码仓库公司要求要严格分开,公司的master分支和release分支是受保护的,我们没有权限,我们开发会切一个feature分支出来,功能开发都在自己的分支上进行,开发之后会合并到release分支,公司要求只有release分支有上线权限。我们会mr (merge request),leader merge 代码。然后通过公司的自动化运维平台,上线后自动会合并到master,如果后续线上有问题的话,我们会从master切hotfix分支,修改完之后提交到release上线,流程和边一样。我们保证master分支和线上的运行的代码是一致的。

作为一个Git高手,在协作开发中,我常用的分支策略是基于Gitflow工作流。Gitflow工作流是一种流行的分支管理模型,它定义了几种标准的分支类型,并规定了它们之间的合并策略。

3.2.2 项目中是如何进行分支的

以下是Gitflow工作流的分支策略:

1. 主分支 (Master Branch): 主分支对应的是稳定版本,用于发布生产代码,只能从其他分支或Hotfix分支合并。
2. 开发分支 (Develop Branch): 开发分支是所有功能开发的集成分支,开发人员在这个分支上进行日常开发。开发分支一般从主分支派生,并每次完成一个功能或修复后合并回主分支。
3. 功能分支 (Feature Branch): 功能分支是为了开发一个特定功能而创建的临时分支,从开发分支派生而来,完成后再合并回开发分支。
4. 发布分支 (Release Branch): 发布分支是在准备发布新版本时创建的,它为发布前的测试和准备工作提供一个稳定的环境。发布分支从开发分支派生而来,并在测试通过后合并回主分支和开发分支。
5. 热修复分支 (Hotfix Branch): 当出现线上紧急Bug需要修复时,创建一个热修复分支。热修复分支是从主分支派生的,修复完成后需要合并回主分支和开发分支。



这种分支策略能够清晰地划分各个阶段和任务,保持主分支的稳定性,便于团队成员协作,减少冲突和合并问题。它能够确保慎重处理新版本发布和紧急修复的过程,并提供一个清晰的提交历史和版本控制。

3.3 Linux部分

3.3.1 说几个linux常见目录

1. /：根目录，Linux文件系统的起始点。
2. /bin：存放系统必需的二进制可执行文件，例如ls、cp、mv等。
3. /boot：存放启动所需的文件，包括内核和引导加载程序。
4. /dev：存放设备文件，包括硬件设备和外部设备。
5. /etc：存放系统的配置文件，例如网络配置、用户配置等。
6. /home：存放用户的家目录，每个用户都有一个以其用户名命名的子目录。
7. /lib：存放系统所需的共享库文件。
8. /media：用于挂载可移动设备（如光盘、U盘等）的目录。
9. /mnt：用于挂载临时文件系统的目录。
10. /opt：存放可选的应用程序或软件包。
11. /proc：虚拟文件系统，包含有关运行中进程和系统内核的信息。
12. /root：超级用户（root）的家目录。
13. /sbin：存放系统管理员使用的系统管理命令。
14. /tmp：存放临时文件，重启后将被清空。
15. /usr：用于存放用户程序和文件，相当于用户的应用程序目录。
16. /var：存放变量数据，包括日志文件、缓存文件等。

3.3.2 说几个linux常用指令

1. ls：列出目录中的文件和子目录。
2. cd：进入指定的目录。
3. pwd：显示当前工作目录的路径。
4. mkdir：创建一个新目录。
5. rm：删除文件或目录。
6. cp：复制文件或目录。
7. mv：移动文件或目录，也可以用来重命名文件或目录。

8. touch：创建一个新文件或更新文件的时间戳。
9. cat：显示文件的内容。
10. less：分页显示文件的内容。
11. tail：显示文件最后几行
12. grep：在文件中搜索指定的文本模式。
13. find：按照指定的条件在文件系统中查找文件。
14. top：查看系统资源使用情况和运行中的进程。
15. ps：显示当前运行的进程。
16. kill：终止指定的进程。
17. chmod：修改文件或目录的权限。
18. ssh：远程登录到另一台计算机。
19. tar：打包和解包文件。

3.3.3 查看linux文件内容的命令有哪些？

1. cat命令：用于显示整个文件的内容。它逐行显示文件内容，并且在终端上输出。例如：`cat filename`
2. less命令：按页查看文件内容。它支持向前/向后翻页、搜索、跳转等操作。例如：`less filename`
3. more命令：按页查看文件内容，它不支持向前翻页，只能向后翻页。例如：`more filename`
4. head命令：用于显示文件的前几行，默认显示头部10行。例如：`head filename` 或 `head -n 5 filename`（显示前5行）
5. tail命令：用于显示文件的最后几行，默认显示尾部10行。例如：`tail filename` 或 `tail -n 5 filename`（显示最后5行）

3.4 烘焙坊项目

3.4.1 介绍一下你的项目背景？

我们公司是一家给客户定制网站和小程序的，这个客户呢是一家做烘焙的连锁店，全国有几十家门店，他们的需求是做一个烘焙的电商，包括烘焙图片，配方，做法的分享。以及烘焙的产品的在线下单售卖（仅支持同城配送）。我负责的是前端系统的咨询模块主要技术呢包括vue及脚手架，后端包括ssm，spring boot框架等。

3.4.2 这个项目最大的收获是什么？

这是我在公司实习的第一个项目，对项目的全流程有了基本的了解，包括项目的需求分析，接口设计，数据库设计（有小组长带），代码落地，测试到部署上线。在这过程中发现，我们前期对需求的分析和设计很重要。如果数据库接口设计不合理，和前后端的协作会比较麻烦，有时候可能需要和产品经理讨论是否需要需求变更。

还有测试的协作也很重要，我发现自己多一些思考和多做一些测试，少出一些重复性的低级bug 甚至不出bug 会提高整个团队的协作效率。

3.4.3 这个项目最大的亮点是什么？

我们这个项目总得来说比较简单，我觉得两个小点吧，对于我来讲挺重要的，比如数据库设计的时候，加一些冗余字段，比如createUserName 啊，这样接口查询的时候，不需要关联表查询了，性能比较高。另外呢，我们资讯内容基本上不改，我们运维在nginx 层做了静态化，缓存1小时。

3.4.4 项目中你实现的部分有哪几张表？

我负责的资讯业务主要包括5张表的设计：

- 用户表：t_user
- 内容表：t_content
- 轮播图表：t_banner
- 分类表：t_category
- 评论表：t_comment



这里需要思考一下表与表之间的关系，如何建立的关系？

3.4.5 Content内容相关的业务都有哪些？

- 发布内容
- 删除内容
- 修改内容
- 通过id查询内容
- 模糊查询内容
- 查询热门内容
- 通过作者id查询相关内容

- 查询当前用户的所有内容
- 管理员查询所有用户的内容

3.4.6 线上出过问题吗？怎么解决？

我这块的模块比较小，在本地测试的时候比较充分，所以线上的问题不多，主要是前后台接口交互产生的一些小问题。因为线上不能debug，我们会通过logback记录用户的请求日志，然后根据tail -f命令可以grep出相关的关键字，一步一步来排查。

3.5 项目及技术拓展

3.5.1 Http协议存在什么问题？

HTTP协议存在以下一些问题：

1. 安全性问题：HTTP协议是明文传输的，数据在传输过程中不加密和验证。这使得攻击者可以窃听和篡改传输的数据，从而造成信息泄露或篡改的风险。
2. 数据完整性问题：由于HTTP协议不提供数据的完整性验证，攻击者可以在传输过程中篡改数据，从而导致数据被破坏或者不可靠。
3. 隐私问题：由于HTTP协议是明文传输的，用户的敏感信息（如用户名、密码等）在传输过程中容易被窃听和获取，从而造成隐私泄露的风险。
4. 缺乏身份验证问题：HTTP协议没有强制的身份验证机制，无法验证服务器和客户端的真实性。这使得攻击者可以冒充服务器或客户端来进行欺骗和攻击。
5. 性能问题：由于HTTP协议每次请求都需要建立和关闭连接，这种无状态的设计导致了频繁的连接开销和重复的身份验证过程，从而影响了性能和效率。



为了解决这些问题，HTTPS协议被引入，通过使用加密和身份验证机制来增强数据传输的安全性和可靠性。HTTPS可以在一定程度上解决HTTP协议存在的问题，并提供更安全的通信环境。

3.5.2 HTTP 与 HTTPS 的区别。

HTTP（Hypertext Transfer Protocol）和HTTPS（Hypertext Transfer Protocol Secure）是用于在客户端和服务端之间传输数据的通信协议。

HTTP是一种明文传输的协议，数据在传输过程中是以纯文本的形式进行传输，不加密和验证。这意味着攻击者可以窃听和篡改HTTP传输的数据，可能存在安全风险。HTTP通常用于在Web浏览器和服务端之间传输网页、图片、视频等非敏感信息。

HTTPS是通过使用SSL（Secure Socket Layer）或TLS（Transport Layer Security）协议对HTTP进行加密和身份验证的安全版本。HTTPS使用了公钥加密来确保传输的数据在网络中的安全性，并使用数字证书对服务端进行身份验证。使用HTTPS的网站可以确保传输的数据在传输过程中不会被篡改或破坏，提供更高的数据安全性。通常，HTTPS用于传输敏感数据，例如在网上购物、银行等要求保密性的操作中。



HTTP和HTTPS的主要区别在于数据传输的安全性和加密机制。HTTPS通过加密和身份验证的方式保护数据的安全性，而HTTP不提供此类保护。因此，为了数据的安全性和隐私保护，建议在需要保护敏感信息的场景下使用HTTPS。

4. 第四教学月

4.1 Spring技术

4.1.1 @Component和@Configuration注解的异同点？

@Component注解通常用于描述一般bean对象，比如具备一定通用性的对象。@Configuration注解通常用于描述Spring工程中的配置类，是一个增强版的@Component注解，在配置类中定义一些由@Bean注解描述的方法，然后通过这些方法对一些自己定义或第三方的Bean进行对象的创建和初始化。当我们使用这@Configuration注解描述一个配置类时，Spring框架底层会为这个@Configuration注解描述的配置类创建一个CGLIB代理对象，然后由代理对象调用@Bean注解描述的方法，同时底层会检测方法返回的Bean是否已创建，假如已创建则不再创建。使用@Component注解描述类时，系统底层并不会为此类创建代理对象，只是创建当前类的对象，然后调用@Bean注解描述的方法，创建和初始化Bean，方法每调用一次就会创建一个新的对象。

4.1.2 Spring AOP的底层原理是怎样的？

Spring AOP的底层原理主要是基于动态代理实现的。分别是基于JDK动态代理和基于CGLib动态代理。

1. 基于JDK动态代理：当目标对象实现了至少一个接口时，Spring将可使用JDK动态代理。JDK动态代理通过反射机制，在运行时创建一个实现目标接口的代理类，将方法调用转发给代理对象，并给调用前后添加增强逻辑。
2. 基于CGLib动态代理：当目标对象没有实现任何接口时，Spring将可使用CGLib动态代理。CGLib动态代理通过继承的方式，创建一个目标类的子类，并重写其中的方法，将方法调用转发给子类，并给调用前后添加增强逻辑。

4.1.3 Spring中声明式事务的原理？

Spring的声明式事务通过AOP（面向切面编程）来实现，在方法级别上提供了事务的管理，并将事务的控制从业务逻辑中解耦出来。

4.1.4 Spring声明式事务在哪些场景会失效？

Spring声明式事务可能失效的场景包括：

1. 方法没有被代理：Spring的声明式事务是基于AOP实现的，如果目标方法没有被代理，则事务管理无法生效。这可能是因为目标方法不是在Spring容器中被调用，而是在同类的非代理方法中被调用。
2. 异常没有被Spring识别为回滚异常：默认情况下，Spring只对RuntimeException及其子类进行事务回滚。如果被管理的方法抛出的异常不是RuntimeException及其子类，并且没有显示地配置为回滚异常，事务将不会回滚。
3. 事务方法自调用：如果一个事务方法内部对同一个类中的另一个事务方法进行调用，事务可能会失效。这是因为Spring的AOP是基于代理的，默认情况下，代理对象调用自身的方法是不会触发AOP增强的，所以内部调用的事务方法不会启用事务管理。
4. 使用try-catch捕获异常，但没有重新抛出异常：如果在事务方法中使用try-catch捕获了异常，并且没有重新抛出异常，事务将不会回滚。因为Spring依赖于异常的抛出来触发事务回滚。
5. @Transactional注解放在了private或protected方法上：默认情况下，Spring只对public方法应用事务管理。如果将@Transactional注解放在了private或protected方法上，事务将无法生效。
6. 事务方法被其他非事务方法直接调用：只有在通过AOP代理调用事务方法时，事务才会起作用。如果一个事务方法被其他非事务方法直接调用，那么事务将不会被应用。
7. 未正确配置事务管理器：在Spring配置文件中没有正确配置事务管理器，或者配置了错误的事务管理器，会导致事务失效。

4.1.5 说几个Spring框架中用到的设计模式？

Spring框架中使用了多种设计模式，以下是其中几个常见的设计模式示例：

1. 单例模式（Singleton）：Spring中的bean默认使用单例模式，即在容器中只会存在一个实例。这样可以保证资源的复用，提高性能和效率。
2. 工厂模式（Factory）：Spring使用工厂模式来创建和管理bean对象。通过配置文件或注解，Spring可以根据需要动态创建不同类型的bean。
3. 代理模式（Proxy）：在Spring AOP中，Spring使用代理模式来对目标对象进行增强。通过代理，可以在目标对象的方法调用前后添加额外的逻辑。
4. 观察者模式（Observer）：Spring的事件驱动模型使用了观察者模式。通过观察者模式，可以实现事件的发布和订阅，实现不同组件之间的解耦。
5. 适配器模式（Adapter）：Spring中的适配器模式用于适配不同的接口或类。例如，Spring的MVC框架中，适配器模式用于将HTTP请求适配到对应的Controller方法。
6. 模板方法模式（Template）：Spring中的JdbcTemplate和HibernateTemplate等模板类使用了模板模式。这些模板类提供了固定的算法骨架，具体的实现由子类或回调方法完成。

实际上Spring框架中还存在其他设计模式的应用，这些设计模式有助于提高代码的可维护性、可扩展性和性能。

4.2 Redis技术基础

4.2.1 说说你对redis的认识？

Redis（Remote Dictionary Server）是一个使用C语言编写的高性能内存数据库，基Key/Value结构存储数据，一般会用来做缓存、消息队列，分布式锁，同时还支持事务、持久化、主从和集群架构等。

4.2.2 Redis基础数据类型有哪些？

Redis基础数据类型：



1. string以字符串形式存储数据，经常用于记录用户的访问次数、文章访问量等。

2. hash以对象形式存储数据，比较方便的就是操作对象中的某个字段。
3. list以列表形式存储数据，可记录添加顺序，允许元素重复。
4. set以集合形式存储数据，不记录添加顺序，元素不能重复，也不能保证存储顺序。
5. zset排序集合，可对数据基于某个权重进行排序。可做排行榜，取TOP N操作。

4.2.3 说几个Redis的全局指令

- keys：查看所有键
- dbsize：键总数
- exists key：检查键是否存在
- del key [key ...]：删除键
- expire key seconds：键过期
- ttl key: 通过 ttl 命令观察键的剩余过期时间
- type key：键的数据结构类型

4.2.4 Redis过期数据的删除策略？

1、惰性删除

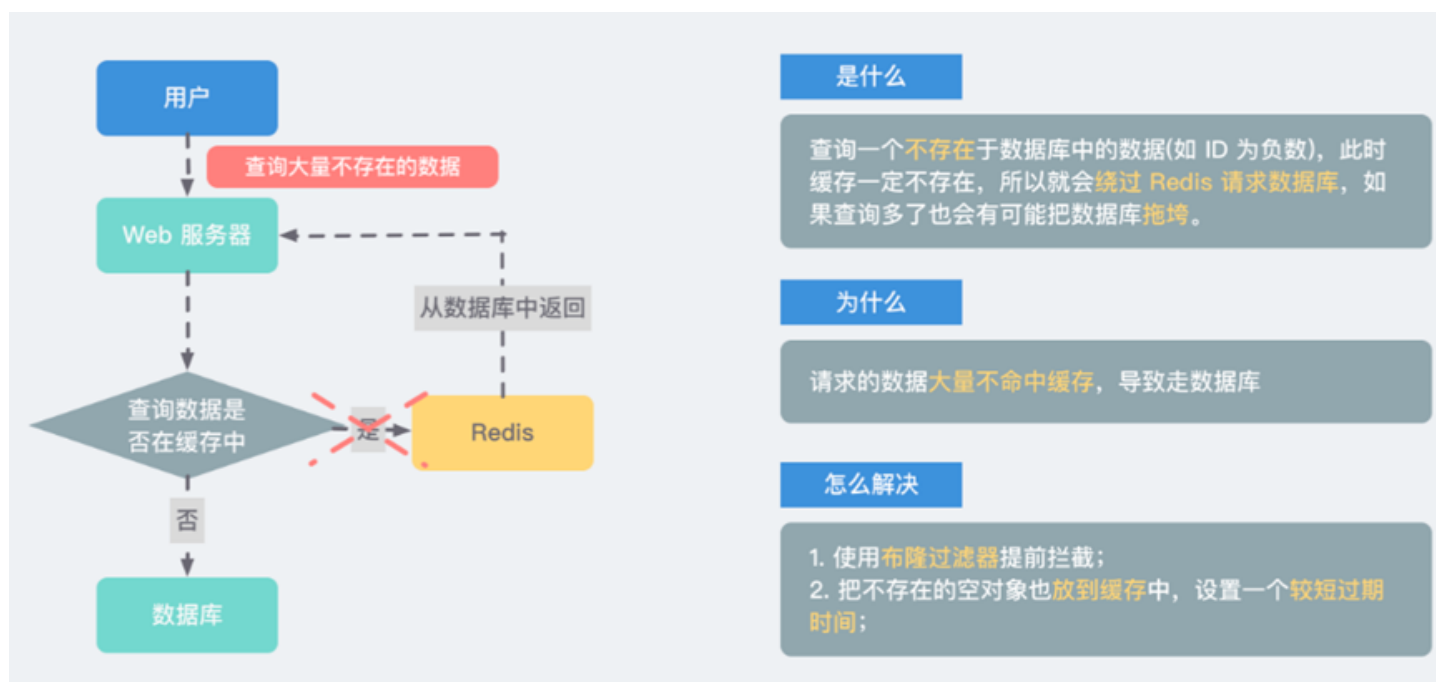
在查询时，检测到key过期了，此时对key进行删除。这里的缺点是，假如key长时间不被访问，也就无法删除，此时就会一直占用着内存。

2、定期删除

每隔一段时间对redis数据进行一次检查，删除过期key。但是这里并不会对所有key进行检查，只是随机取一些key，检查是否过期，过期了然后删除。

4.2.5 什么缓存穿透以及解决方案？

当访问一个缓存和数据库都不存在的 key时，请求会直接打到数据库上，并且查不到数据，没法写缓存，所以下一次同样会打到数据库上。这时缓存就好像被“穿透”了一样，起不到任何作用。假如一些恶意的请求，故意查询不存在的key,请求量很大，就会对后端系统造成很大的压力，甚至数据库挂掉，这就叫做缓存穿透。



解决方案?

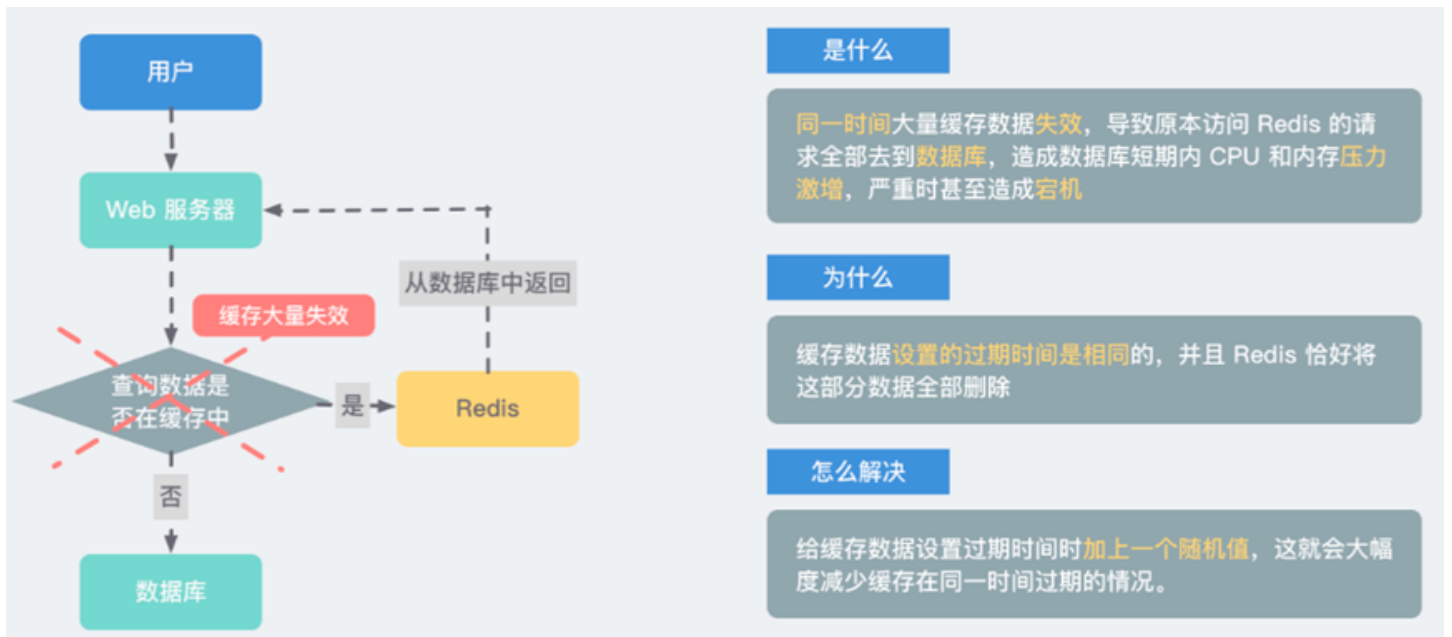
方案1: 接口校验。在正常业务流程中可能会存在少量访问不存在 key 的情况, 但是一般不会出现大量的情况, 所以这种场景最大的可能性是遭受了非法攻击。可以在最外层先做一层校验, 用户鉴权、数据合法性校验等, 例如商品查询中, 商品的ID是正整数, 则可以直接对非正整数直接过滤等等。

方案2: 缓存空值。当访问缓存和DB都没有查询到值时, 可以将空值写进缓存, 但是设置较短的过期时间, 该时间需要根据产品业务特性来设置。

方案3: 布隆过滤器。使用布隆过滤器存储所有可能访问的 key, 不存在的 key 直接被过滤, 存在的 key 则再进一步查询缓存和数据库。可把所有可能存在的key放到一个大的Bitmap中, 查询时通过该 bitmap 过滤。

4.2.6 什么缓存雪崩以及解决方案?

缓存雪崩是当缓存服务器重启或者大量缓存集中在某一个时间段失效, 造成瞬时数据库请求量大, 压力骤增, 导致系统崩溃。缓存雪崩其实有点像“升级版的缓存击穿”, 缓存击穿是一个热点 key, 缓存雪崩是一组热点 key。



解决方案：

- 方案1：打散过期时间。不同的key，设置不同的过期时间（例如使用一个随机值），让缓存失效的时间点尽量均匀。
- 方案2：做二级缓存。A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期。
- 方案3：加互斥锁。缓存失效后，通过加锁或者队列来控制写缓存的线程数量。比如对某个key只允许一个线程操作缓存，其他线程等待。
- 方案4：热点数据不过期。该方式和缓存击穿一样，要着重考虑刷新的时间间隔和数据异常如何处理的情况。

4.3 ES技术基础

4.3.1 什么是Elasticsearch?

Elasticsearch是一个开源的分布式搜索和分析引擎，基于Lucene库构建。它可以帮助我们快速地处理和分析大量的数据，具有高度可扩展性和实时性。

4.3.2 Elasticsearch的主要特点？

- 分布式：Elasticsearch可以将数据分布到多个节点上进行存储和处理，从而实现水平扩展。
- 实时性：Elasticsearch对于写入操作具有几乎实时的响应能力。
- 高可用性：Elasticsearch可以通过复制数据到多个节点，提供高可用性服务。
- 多种查询方式：Elasticsearch支持全文查询、结构化查询、地理位置查询等多种查询方式。
- 自动化：Elasticsearch提供了自动化的集群管理功能，可以自动处理节点故障、重新分片等操作。

4.3.3 Elasticsearch的索引的作用是什么？

在Elasticsearch中，索引是一种用于组织和存储数据的数据结构，类似于关系型数据库中的表。索引可以包含多个文档，每个文档又由多个字段组成。索引在Elasticsearch中扮演着非常重要的角色，它的作用主要包括：

- 索引用于存储和组织数据，使得数据可以被高效地检索和查询。
- 支持分布式搜索和并行处理。
- 实现高性能、复杂的查询操作，包括全文搜索、结构化查询、地理位置查询等。

4.3.4 说说你对倒排索引的理解？

倒排索引是一种将文档中的词条映射到文档的数据结构。在倒排索引中，每个关键词都有一个对应的词项列表，列表中包含使用该关键词的所有文档。倒排索引是建立在Lucene库之上的，它可以高效地进行全文检索。倒排索引可以帮助我们：

- 高效查询：倒排索引使得Elasticsearch可以快速定位到包含指定关键词的文档，并返回相关的结果。
- 降低I/O操作：倒排索引存储了每个关键词在哪些文档中出现，可以减少不必要的I/O操作，提高查询性能。
- 支持高级查询功能：倒排索引可以轻松地支持复杂的查询功能，例如短语匹配、通配符查询、模糊查询等。

4.3.5 如何在Elasticsearch中进行全文搜索？

在Elasticsearch中，可以使用查询语句进行全文搜索。下面是一个示例查询语句：

```
1 GET /my_index/_search
2 {
3   "query": {
4     "match": {
5       "content": "Elasticsearch"
6     }
7   }
8 }
```

上述查询语句将在名为 `my_index` 的索引中搜索包含关键词 `Elasticsearch` 的文档，并返回相关的结果。这个查询将会匹配文档中的 `content` 字段。

4.3.6 怎样在Elasticsearch中执行聚合查询？

在Elasticsearch中，可以使用聚合查询来进行数据分析和统计。下面是一个示例聚合查询：

```
1 GET /my_index/_search
2 {
3   "aggs": {
4     "popular_tags": {
5       "terms": {
6         "field": "tags"
7       }
8     }
9   }
10 }
```

上述查询将在名为 `my_index` 的索引中聚合统计 `tags` 字段的数据。它将返回每个标签及其对应的文档数量。

4.3.7 说说你对分片和副本的理解？

在Elasticsearch中，索引中的数据会被分配到多个分片中进行存储和处理。每个分片是一个独立的工作单元，可以在多个节点上进行复制和分布。副本是分片的复制，用于提供高可用性和负载均衡。分片和副本对于Elasticsearch的性能和可用性有以下影响：

- 性能：增加分片可以提高系统的吞吐量和并行处理能力，但也会增加一定的系统开销。过多的分片数量可能会导致性能下降。
- 可用性：通过创建副本，可以提供数据的冗余备份，保证了系统的高可用性。如果某个节点故障，可以通过副本来提供服务，并且可以在故障节点恢复后进行自动同步。

4.3.8 如何优化Elasticsearch的性能？

要优化Elasticsearch的性能，可以考虑以下几点建议：

- 合理配置分片和副本：分片数量和副本数量要根据需求进行合理配置，避免过多的分片和副本造成性能下降。
- 使用合适的硬件资源：选择高性能的硬件资源，例如快速的磁盘、大内存等，以提高读写性能和响应能力。
- 索引优化：合理设计索引的数据结构，使用适当的数据类型、分词器和标记，提高查询的效率。
- 查询性能优化：合理使用查询优化功能，例如缓存、预热、过滤器等，减少不必要的计算和IO开销。
- 负载均衡：通过增加节点和副本，实现请求的负载均衡，避免单个节点的性能瓶颈。

请注意，上述问题仅为示例，实际的面试问题可能会涉及到更多的话题和细节。最好根据自己对Elasticsearch的了解，进一步准备和学习相关知识。

4.4 学茶网项目

4.4.1 介绍一下学茶这个项目？

学茶商城是一个包含茶相关的资讯与电商的平台，用户可以在这个平台了解茶相关的资讯，例如茶叶的种类、功效、喝茶的禁忌等，用户也可以对这些资讯进行评论，并且，通过这些资讯进行引流，用户可以在平台上直接购买相关的产品，例如茶具、茶叶等。

4.4.2 项目的核心业务有哪些？

此项目主要包括资讯服务，用户体系服务，交易服务、基础服务（附件，基础数据）每个服务都拆分成了前后台系统，前台面对商城用户，后台为公司内部管理人员使用。



注意，这里重点讲对项目的全局认知

4.4.3 你负责开发的功能有哪些？

这里主要描述自己负责的功能模块，包括业务的理解，技术方案，功能实现，和团队协作，比如和产品、测试、运维等团队沟通协作，真实企业开发经验等。

我主要负责的是单点登录服务、账号后台管理服务、商城后台管理服务、商城前台管理服务。

1. 单点登录服务：用户登录、退出登录、登录日志
2. 账号后台管理服务：显示用户列表、修改用户的启用状态、修改用户的基本资料、修改用户密码
3. 商城后台管理服务：商品类别数据的常规管理、商品类别数据的缓存、发布商品、审核商品、上架或下架商品、向ES中写入商品数据供前台搜索

4. 商城前台管理服务：用户浏览商品、搜索商品、收货地址管理、将商品添加到购物车、购物车管理、创建订单、查看订单。



这里，还可以从一个核心业务说起，比如商品是怎么发布和上架的，怎么分类的，后台怎么管理，前台是怎么展示的，数据是怎么通过mysql 同步到es的。还有库存是怎么维护的，包括业务流程和系统交互流程和数据流程。

4.4.4 说说项目的开发流程是怎样的？

由业务人员提供需求，由产品经理提供需求分析文档，定稿后，由项目经理主持需求分析会，要求全体

人员参加，分析需求，开发难点，开发周期以及上线时间，然后由具体的开发人员负责，并按照需求文档进行架构设计，详细设计，代码编写，开发完成后，提交代码到Git分支，由项目经理转测试组进行功

能测试，功能测试无误后，转交业务人员进行业务测试，如果测试都没有问题，那么将进行预上线准备。

4.4.5 项目的开发周期是怎样的？

2个星期的需求分析，1个星期的架构设计，1个星期详细设计，4个星期开发，1个星期测试，1个星期联

调，1个星期修改bug，1个星期上线部署，后续持续更新项目。

4.4.6 项目中的测试数据从哪里来的？

省市信息是真实的全国信息，其它测试数据是通过提前准备的SQL脚本文件批量插入的。

4.4.7 与测试部门的配合工作有哪些？

需求分析和设计：开发和测试部门需要首先进行需求分析和设计，明确项目目标、功能点、需求规格等，并根据需求提出可行性和技术实现方案。

编码和测试用例的设计开发：开发部门负责完成系统的编码实现，同时为测试部门提供完整的代码和文

档资料；而测试部门则需要根据开发完成的代码，设计合适的测试方案与测试用例。

测试环境的搭建：测试部门需要在系统开发前期就和开发部门提出测试环境和测试数据的需求，搭建测试环境，在系统实际开发过程中，可以及时地搭建，并解决测试中发现的问题，从而及时反馈给开发人员。

合作测试：测试阶段通常是开发和测试部门之间配合最紧密的时间段，测试人员需要根据系统设计和测试用例，对开发完成的功能点进行测试，并及时反馈测试结果给开发人员，协助开发人员进行程序修复和系统优化。

系统性能和压力测试：除了功能测试，测试人员还需要完成系统性能和压力测试，确保系统在实际运行中稳定可用，对于系统的性能瓶颈等存在问题，需要及时和开发人员反馈和沟通，协助开发人员优化改进系统。

4.4.8 与运维部门的配合工作有哪些？

配置文件的管理：在Java后台开发中，会涉及到各种配置文件的编辑和管理，这个环节需要和运维部门密切合作，运维部门需要提供相应的配置文件，并负责配置文件的管理和更新。

应用部署和发布：在Java后台开发完成后，需要进行部署和发布，这个环节需要与运维部门共同合作，确定服务器环境要求，以及应用部署所需的资源和技术支持。

系统的监控和维护：运维部门负责系统的监控和维护，对于开发过程中可能出现的错误和异常，运维需要及时发现并通知开发人员，协助开发人员解决问题。

系统的备份和恢复：系统数据备份和恢复是整个系统运维过程中的重要环节，需要和开发人员协同配合，保证备份和恢复的准确性和完整性。

性能调优和故障处理：在系统运行过程中，可能存在性能问题和故障情况，运维部门需要和开发人员配合，分析和解决这些问题，及时对系统进行调优、使用新的技术方案，提高系统的稳定性和性能。

4.4.9 项目上线的基本流程怎样的？

项目组会提前准备打包这次上线的内容，由项目经理与每个开发人员沟通具体内容，确定无误后，将测

试好的代码在定版环境进行测试，没有问题提交jar包发布上线流程申请，经过负责人同意后方可进行上线。

4.4.10 项目上线后要关注什么？

项目上线当天需要对上线的功能进行验证，验证无误后填写上线情况表，之后上报业务人员进行业务验证。每一个发布的版本都在代码中做了预警。当出现具体的情况的时候会通过短信告知预警信息。

4.4.11 接口响应时间是怎样的？

项目的接口响应时间一般取决于多个因素，包括系统的架构、硬件配置、网络状况以及实际业务需求等。一般来说，较理想的接口响应时间应该在几毫秒到几秒之间。

对于一些简单的接口，比如查询数据库中的数据，一般情况下响应时间应该在几十毫秒以内。而对于一些复杂的接口，比如需要进行复杂计算或涉及大数据量的接口，响应时间可能会相对较长，可以在几百毫秒到几秒之间。

我们的项目单节点需要支持TPS \geq 100TPS、对外提供接口的平均响应时间300ms左右。

4.4.12 项目中遇到了什么问题？

注意，这里可以包括技术问题，业务问题，团队协作问题，还有对通用方案的理解等。

问题1：接口响应慢，通过查日志是数据库sql执行过慢，通过explain 定位上线的sql 没命中索引，引出对数据库索引的理解，分析调整sql 上线。

问题2：redis的一些通用方案，缓存一致性的理解，缓存数据丢失，淘汰策略的理解。

问题3：线上问题的定位以及排障，包括事前埋点，日志，事务中日志异常数量观测，告警观测，接口性能观测 事后 对日志异常排查 定位修复。

问题4：团队协作问题，因业务理解上的偏差，导致代码上的返工。

4.4.13 项目是如何验收的？需要注意什么？

确定验收标准和条件：

在项目启动之前，需要和客户或用户明确项目的验收标准和验收条件，包括软件功能、性能、安全、兼容性等方面，以确保项目成功交付并达到客户的期望。

验证交付物：项目验收过程中需要对每个交付物进行验证，确保其符合验收标准及规范，并能够有效地满足用户需求。

进行充分的测试：在进行验收前，需要对项目进行充分的测试，包括功能测试、性能测试、安全测试等，保证项目的质量和稳定性。

撰写验收报告：项目验收完成后，需要编写验收报告，详细记录项目的验收情况和结论，包括验收标准

的执行情况、交付物的验收结果、测试报告、问题和建议等。

跟踪和解决问题：在项目交付后，可能出现问题或者用户有使用上的疑问，需要开展问题解决工作，及

时跟踪和解决问题，保证交付物的稳定性和用户的满意度。

4.4.14 项目的可靠性如何保障？

质量保证：建立严格的质量管理体系，根据项目需求和规范要求，对各个环节进行全程质量控制，确保项目的质量达到预期目标。

风险管理：在项目生命周期的各个阶段，针对潜在风险进行识别、评估和控制，建立风险应对措施，有效避免风险对项目可靠性造成的影响。

设计优化：项目设计时需充分考虑可行性、合理性和可维护性，通过综合分析，采用先进的技术及设备、材料和工艺，以达到最佳的可靠性。

测试验证：在项目实施和交付前，进行全面的测试和验证，验证关键功能性、性能、安全性、稳定性等方面是否达到要求。**持续维护：**项目交付后，建立健全的维护和保养机制，对项目进行定期的维护和检查，及时发现和解决问题，确保良好的运行状态和系统的可靠性。

4.4.15 项目故障处理的方式？

在代码架构中引入日志框架，将系统日志输出到文件。通过日志debug分析问题，error定位具体问题。故障复盘，确定故障的第一责任人。修改功能代码,并提交修改方案。和避免重复出现问题的保证策略。

4.4.16 项目中到了什么技术？

前端框架：vue, Element UI,Axios,

框架方面：Spring, Spring MVC, MyBatis, Spring Boot, Spring Validation、Spring Security、Spring Data Redis、Spring Data Elasticsearch、MyBatis Plus, PageHelper、Lombok、fastjson、hutool 等。

中间件：Redis、Elasticsearch。

开发工具：Git + Maven + JDK1.8

4.4.17 如何保证前后端的数据不重复提交

前端防止重复提交：

- 给提交按钮添加 disabled 属性,提交后将按钮设置为 disabled,防止重复点击提交。
- 使用变量如 let isSubmitting = false,在提交前将它设为 true,提交后设为 false。在提交回调中判断这个变量,如果是 true 就阻止提交。
- 在提交后清空表单中的数据,这样就无法重新提交相同数据了。
- 使用数组存储已提交过的表单数据,提交前判断数组中是否已存在,存在就阻止提交。
- 锁定提交按钮一定时间,例如提交后 2 秒内禁用提交按钮。

后端防止重复提交：

- 在服务端生成一个唯一的请求 ID,提交表单时前端将此 ID 提交到后端。后端可以根据该 ID 来判断是否重复提交。
- 设置 token,提交表单时生成一个 token,写入到表单中。服务端接口验证 token 是否重复。
- 在服务端缓存已处理过的表单数据,比如 redis 等,提交时判断缓存中是否已存在。
- 服务端接收到提交请求后,在返回响应前暂存表单数据,设置超时时间,如果在超时时间内再次收到一模一样的数据则认为是重复提交。

4.4.18 项目中redis存储了哪些数据？

存储了学茶的频道数据、购物车数据、用户的登录信息等。

频道数据：Hash 结构 key为channel field 为频道ID, value为频道json串。

购物车：hash结构, key: shopping_cart field:goods_id , value 是商品相关信息 (包括商品id 商品标题 数量 单价 库存数等)

登录信息：hash 结构, key: user_{user_id} field:status value 状态信息，field:expired,value: 过期时间（续期）

4.5 Redis技术拓展

4.5.1 说说Redis的持久化方式？

Redis持久化是把内存中的数据同步到硬盘文件中，当Redis重启后再将硬盘文件内容重新加载到内存以实现数据恢复的目的。具体持久化方式，分别为RDB和AOF方式。

4.5.2 说说Redis中RDB方式的持久化？

RDB方式的持久化是Redis数据库默认的持久化机制,是保证redis中数据可靠性的方式之一，这种方式可以按照一定的时间周期策略把内存中的数据以快照(二进制数据)的形式保存到磁盘文件中，即快照存储。对应产生的数据文件为dump.rdb。

4.5.3 RDB持久化方式的常用配置参数？

这里表示每隔60s，如果有超过1000个key发生了变更，就执行一次数据持久化。

这个操作也被称之为snapshotting(快照)。

save 60 1000

持久化 rdb文件遇到问题时，主进程是否接受写入，yes 表示停止写入，

如果是no 表示redis继续提供服务。

stop-writes-on-bgsave-error yes

在进行快照镜像时,是否进行压缩。yes:压缩，

但是需要一些cpu的消耗。no:不压缩，需要更多的磁盘空间。

rdbcompression yes

一个CRC64的校验就被放在了文件末尾，当存储或者加载rdb文件的时候

会有一个10%左右的性能下降，为了达到性能的最大化，你可以关掉这个配置项。

rdbchecksum yes

快照的文件名

dbfilename dump.rdb

存放快照的目录
dir /var/lib/redis

4.5.4 什么情况下会触发RDB持久化？

1. 基于配置文件中的save规则周期性的执行持久化。
2. 手动执行了shutdown操作会自动执行rdb方式的持久化。
3. 手动调用了save或bgsave指令执行数据持久化。
4. 主从复制架构下Slave连接到Master时，Master会对数据持久化，然后全量同步到Slave。

4.5.5 save和bgsave有什么不同？

SAVE生成 RDB 快照文件，但是会阻塞主进程，服务器将无法处理客户端发来的命令请求，所以通常不会直接使用该命令。BGSAVE指令会fork 子进程来生成 RDB 快照文件，阻塞只会发生在 fork 子进程的时候，之后主进程可以正常处理请求。

4.5.6 RDB方式持久化有哪些优势？

1. RDB 文件是压缩的二进制文件，占用空间小，保存的是某个时间点的数据，适合做备份。
2. RDB 适用于灾难恢复,它只有一个文件，文件内容都非常紧凑，方便传送到其它数据中心。
3. RDB 持久化性能较好，可由子进程处理保存工作，父进程无须执行任何磁盘 I/O 操作。

4.5.7 RDB方式持久化有哪些缺点？

1、RDB方式在服务器故障时容易造成数据的丢失。

实际项目中，我们可通过配置来控制持久化的频率。但是，如果频率太频繁，可能会对 Redis 性能产生影响。所以通常可能设置至少5分钟才保存一次快照，这时如果 Redis 出现宕机等情况，则意味着最多可能丢失5分钟数据。

2、RDB 方式使用 fork 子进程进行数据的持久化。

子进程的内存是在fork操作时父进程中数据快照的大小，如果数据快照比较大的话，fork 时开辟内存会比较耗时，同时这个fork是同步操作，所以，这个过程会导致父进程无法对外提供服务。

3、RDB持久化过程中的fork操作，可能会导致内存占用加倍。

Linux系统fork子进程采用的是 copy-on-write 的方式（写时复制，修改前先复制），在 Redis 执行 RDB 持久化期间，如果 client 写入数据很频繁，那么将增加 Redis 占用的内存，最坏情况下，内存的占用将达到原先的2倍。

4.5.8 如何理解AOF方式的持久化？

Redis中AOF方式的持久化是将Redis收到的每一个写命令都追加到磁盘文件的最后，类似于MySQL的binlog。当Redis重启时，会重新执行文件中保存的写命令，然后在内存中重建整个数据库的内容。

AOF 持久化默认是关闭的，可以通过配置appendonly yes 开启。当 AOF 持久化功能打开后，服务器在执行完一个写命令之后，会将被执行的写命令追加到服务器端 aof 缓冲区（aof_buf）的末尾，然后再将 aof_buf 中的内容写到磁盘。

Linux 操作系统中为了提升性能，使用了页缓存（page cache）。当我们将 aof_buf 的内容写到磁盘上时，此时数据并没有真正的落盘，而是存在在 page cache 中，为了将 page cache 中的数据真正落盘，需要执行 fsync / fdatasync 命令来强制刷盘。这边的文件同步做的就是刷盘操作，或者叫文件刷盘可能更容易理解一些。

4.5.9 AOF持久化方式有什么优势？

- AOF方式 比 RDB方式的持久化更加可靠。

你可以设置不同的 fsync 策略（no、everysec 和 always）。默认是 everysec，在这种配置下，redis 仍然可以保持良好的性能，并且就算发生故障停机，也最多只会丢失一秒钟的数据。

- AOF文件是一个基于纯追加模式的日志文件。

即使日志因为某些原因而包含了未写入完整的命令（比如写入时磁盘已满，写入中途停机等等），我们也可以使用 redis-check-aof 工具也可以轻易地修复这种问题。

- AOF文件太大时，Redis 会自动在后台进行重写。

重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。整个重写是绝对安全，因为重写是在一个新的文件上进行，同时 Redis 会继续往旧的文件追加数据。当新文件重写完毕，Redis 会把新旧文件进行切换，然后开始把数据写到新文件上。

- AOF 文件以Redis协议的格式有序地保存了对数据库执行的所有写操作，可读性好。

对如果你不小心执行了 FLUSHALL 命令把所有数据刷掉了，但只要 AOF 文件没有被重写，那么只要停止服务器，移除 AOF 文件末尾的 FLUSHALL 命令，并重启 Redis，就可以将数据集恢复到 FLUSHALL 执行之前的状态。

4.5.10 AOF持久化方式有什么劣势？

- 对于相同的数据集，AOF 文件的大小一般会比 RDB 文件大。

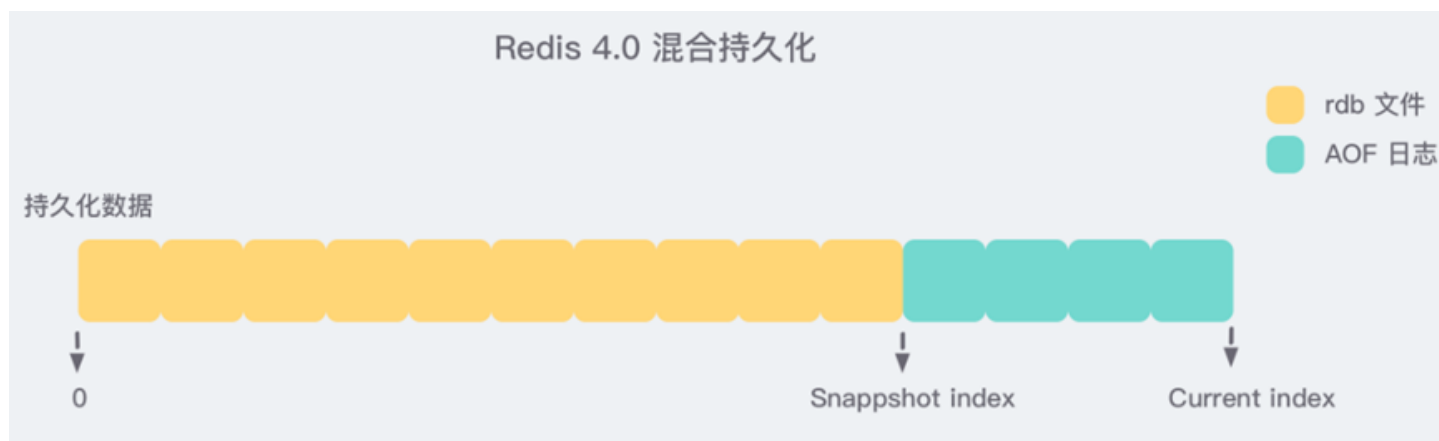
根据所使用的 fsync 策略，AOF 的速度可能会比 RDB 慢。通常 fsync 设置为每秒一次就能获得比较高的性能，而关闭 fsync 可以让 AOF 的速度和 RDB 一样快。

- AOF 可能会因个别命令的原因，导致 AOF 文件在重新载入时，无法将数据恢复到原样。

虽然这种 bug 在 AOF 文件中并不常见，但是相较而言，RDB 几乎是不可能出现这种 bug 的。

4.5.11 如何理解Redis的混合持久化？

混合持久化并不是一种全新的持久化方式，而是对已有方式的优化。混合持久化只发生于 AOF 重写过程。使用了混合持久化，重写后的新 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。



开启：混合持久化的配置参数为 aof-use-rdb-preamble，配置为 yes 时开启混合持久化，在 redis 4 刚引入时，默认是关闭混合持久化的，但是在 redis 5 中默认已经打开了。

关闭：使用 `aof-use-rdb-preamble no` 配置即可关闭混合持久化。混合持久化本质是通过 AOF 后台重写（`bgrewriteaof` 命令）完成的，不同的是当开启混合持久化时，fork 出的子进程先将当前全量数据以 RDB 方式写入到新的 AOF 文件，然后再将 AOF 重写缓冲区（`aof_rewrite_buf_blocks`）的增量命令以 AOF 方式写入到文件，写入完成后通知主进程将新的含有 RDB 格式和 AOF 格式的 AOF 文件替换旧的 AOF 文件。

优点：结合 RDB 和 AOF 的优点, 更快的重写和恢复。

缺点：AOF 文件里面的 RDB 部分不再是 AOF 格式，可读性差。

4.5.12 Redis为什么要AOF重写？

AOF持久化是通过保存被执行的写命令来记录数据库状态的，随着写入命令的不断增加，AOF文件中的内容会越来越多，文件的体积也会越来越大。如果不加以控制，体积过大的 AOF 文件可能会对 Redis 服务器、甚至整个宿主机造成影响，并且 AOF 文件的体积越大，使用 AOF 文件来进行数据还原所需的时间就越多。

举个例子，如果你对一个计数器调用了 100 次 `INCR`，那么仅仅是为了保存这个计数器的当前值，AOF 文件就需要使用 100 条记录。然而在实际上，只使用一条 `SET` 命令已经足以保存计数器的当前值了，其余 99 条记录实际上都是多余的。为了处理这种情况，Redis 引入了 AOF 重写，可以在不中断服务端处理请求的情况下，对 AOF 文件进行重建（`rebuild`）。

4.5.13 描述一下AOF重写的过程？

Redis中的AOF重写是生成新的 AOF 文件来代替旧 AOF 文件，这个新的 AOF 文件包含重建当前数据集所需的最少命令。具体过程是遍历所有数据库的所有键，从数据库读取键现在的值，然后用一条命令去记录键值对，代替之前记录这个键值对的多条命令。

命令：有两个 Redis 命令可以用于触发 AOF 重写，一个是 `BGREWRITEAOF`（底层fork子进程来重写）、另一个是 `REWRITEAOF` 命令(会阻塞主进程)；

开启：AOF 重写由两个参数共同控制，`auto-aof-rewrite-percentage` 和 `auto-aof-rewrite-min-size`，同时满足这两个条件，则触发 AOF 后台重写 `BGREWRITEAOF`。例如：

```
// 当前AOF文件比上次重写后的AOF文件大小的增长比例超过100
auto-aof-rewrite-percentage 100
// 当前AOF文件的文件大小大于64MB
auto-aof-rewrite-min-size 64mb
```

关闭：`auto-aof-rewrite-percentage 0`，指定0的百分比，以禁用自动AOF重写功能。

4.5.14 AOF 后台重写存在的问题？

AOF 使用子进程进行重写，解决了主进程阻塞的问题，但是仍然存在另一个问题，那就是子进程在进行 AOF 重写期间，服务器主进程还需要继续处理命令请求，新的命令可能会对现有的数据库状态进行修改，从而使得当前的数据库状态和重写后的 AOF 文件保存的数据库状态不一致。

4.6 项目及技术拓展

4.6.1 什么是最左前缀匹配原则？

最左前缀原则，就是最左优先，在创建多列索引时，要根据业务需求，where 子句中使用最频繁的一列放在最左边。例如，当我们创建一个组合索引的时候，如(k1,k2,k3)，相当于创建了 (k1)、(k1,k2)和(k1,k2,k3)三个索引，这就是最左匹配原则。

4.6.2 你们是如何优化慢查询的？

- 分析语句，是否加载了不必要的字段/数据。
- 分析 SQL 执行语句，是否命中索引等。
- 如果 SQL 很复杂，优化 SQL 结构。
- 如果表数据量太大，考虑分表。

4.6.3 如何排查并解决MySQL导致CPU飙升的问题？

排查过程：

- 使用 top 命令观察，确定是 mysqld 导致还是其他原因。
- 如果是 mysqld 导致的，show processlist，查看 session 情况，确定是不是有消耗资源的 sql 在运行。
- 找出消耗高的 sql，看看执行计划是否准确，索引是否缺失，数据量是否太大。

处理：

- kill 掉这些线程(同时观察 cpu 使用率是否下降)，
- 进行相应的调整(比如说加索引、改 sql、改内存参数)
- 重新跑这些 SQL。

其他情况：

也有可能是每个 sql 消耗资源并不多，但是突然之间，有大量的 session 连进来导致 cpu 飙升，这种情况就需要跟应用一起来分析为何连接数会激增，再做出相应的调整，比如说限制连接数等。

4.6.4 什么是分库分表？

分库分表（Sharding）是一种数据库水平扩展的技术，用于解决数据库存储和处理大规模数据的需求。

分库是将数据存储到多个数据库中，每个数据库存储数据的一部分。每个数据库相互独立，可以在不同的物理服务器上运行，由分库中间件或应用程序负责查询路由和数据的分发。

分表是将单个数据库中的数据拆分为多个表，每个表存储数据的一部分。分表可以按照某种策略（如范围分片、哈希分片等）将数据分散到不同的物理表中，以达到负载均衡和提高查询性能的目的。分表的策略可以根据应用程序的需求和数据分布进行选择。

4.6.5 为什么要进行分库分表？

当应用程序的数据量逐渐增长时，传统的单个数据库可能无法满足高性能和高可用性的要求。此时，可以采用分库分表的策略将数据分散到多个数据库或表中，以提高系统的吞吐量和扩展性。。此时我们通常会采用分库分表的方案，其好处主要包括：

- 可扩展性：将数据分散到多个数据库或表中，可以水平扩展系统的存储容量和处理能力。
- 高性能：将数据分片存储到不同的数据库或表中，可以避免单个数据库的性能瓶颈，提高系统的查询性能。
- 高可靠性：通过复制和备份数据，可以增加系统的容错能力和数据恢复能力。



总而言之，分库分表是一种数据库水平扩展的技术，通过数据的拆分和分散存储，提高系统的扩展性、性能和可靠性。分库分表也带来了一些挑战，如分布式事务、数据一致性、跨分片查询等问题需要额外的处理和考虑。

4.6.6 分库分表可能带来什么问题？

分库分表的优点包括可扩展性、高性能、高可靠性和节约成本。缺点包括分布式事务、数据一致性和跨分片查询等的处理复杂性。

4.6.7 如何保证分库分表的数据一致性？

数据一致性可以通过分布式事务、分布式锁、同步复制和异步复制等技术来保证。

4.6.8 什么时候进行分表以及可以解决的问题？

当数据量比较大（例如单表记录大于500万），并且插入和查询速度已经慢到影响到使用的时候，可以考虑分表。

通过分表可以提高系统的并发处理能力，磁盘I/O的性能，插入数据时重新建立索引的量也会减少。

4.6.9 你了解哪些分库分表的中间件？

ShardingSphere是一套开源的分布式数据库中间件解决方案组成的生态圈，它由Sharding-JDBC、Sharding-Proxy和Sharding-Sidecar（计划中）这3款相互独立的产品组成。它们均提供标准化的数据分片、分布式事务和数据库治理功能，可适用于如Java同构、异构语言、容器、云原生等各种多样化的应用场景。

4.6.10 什么是读写分离？

读写分离，基本的原理是让主数据库处理事务性增、改、删操作（INSERT、UPDATE、DELETE），而从数据库处理SELECT查询操作。数据库复制被用来把事务性操作导致的变更同步到集群中的从数据库。

4.6.11 为什么要进行读写分离？

因为数据库的“写”（例如写10000条数据到oracle可能要3分钟）操作是比较耗时的。但是数据库的“读”可能比较快（例如从oracle读10000条数据可能只要5秒钟）。**所以要读写分离，解决数据库的写入影响了查询的效率的问题。**

4.6.12 什么情况下适合读写分离？

当数据库读远大于写，就可以考虑主数据负责写操作，从数据库负责读操作，一主多重，从而把数据读写分离，最后还可以结合redis等缓存来配合分担数据的读操作，大大的降低后端数据库的压力。

4.6.13 描述一下主从复制的基本过程？

步骤一：主库的更新事件(update、insert、delete)被写到 binlog

步骤二：从库发起连接，连接到主库。

步骤三：此时主库创建一个 binlog dump thread，把 binlog 的内容发送到从 库。

步骤四：从库启动之后，创建一个 I/O 线程，读取主库传过来的 binlog 内容并 写入到 relay log

步骤五：还会创建一个 SQL 线程，从 relay log 里面读取内容，从 Exec_Master_Log_Pos 位置开始执行读取到的更新事件，将更新内容写入到 slave 的 DB 中。

5. 第五教学月

5.1 微服务初识

5.1.1 如何理解微服务？


微服务是一种架构风格，通过将一个大型应用拆分为一系列小而独立的服务，每个服务都有自己的业务逻辑和数据存储，相互之间通过轻量级的通信机制进行交互，以实现高内聚、低耦合和灵活扩展的目标。

微服务架构的核心概念是服务的独立性和自治性。每个微服务都是一个独立的应用，可以独立部署、独立扩展和独立升级，且具有清晰的业务边界。这使得团队可以根据业务需求独立开发和维护每个微服务，而不会影响其他服务的运行。

微服务之间通过轻量级的通信机制进行交互，常用的方式包括基于 HTTP/REST 的 API 调用、消息队列和事件驱动等。通过精心设计的接口和通信机制，微服务之间可以灵活协作，实现分布式系统的功能需求。

微服务架构还强调弹性和容错性。由于每个微服务都是独立的，当一个服务发生故障时，其他服务不会受到影响，整个系统可以保持部分可用。同时，由于微服务的小规模和独立性，可以更快地进行故障定位和处理，提高系统的可靠性和可维护性。

另外，微服务架构也提倡使用适合的技术栈和工具。由于每个微服务都是独立的，团队可以根据需求选择最适合的技术栈和工具，并且灵活地进行技术更新和升级，以满足不同的业务需求和技术挑战。

 总的来说，微服务是一种将大型应用拆分为一系列小而独立的服务的架构风格。它通过服务的独立性、自治性、轻量级的通信机制、弹性和容错性以及适合的技术栈和工具等特点，帮助开发团队构建高内聚、低耦合和灵活可扩展的分布式系统。

5.1.2 微服务架构诞生的背景？

微服务架构的诞生背景可以追溯到云计算、容器化和持续交付的趋势。

首先，云计算的兴起使得企业能够利用虚拟化技术将应用程序部署在云平台上，提供弹性和可扩展性。这使得传统的单体应用变得笨重、难以管理和部署。

其次，容器化技术的发展使得应用程序的打包、传输、部署和管理变得更加灵活和高效。容器可以快速构建、部署和复制，并且与操作系统和硬件无关，提供了一种更轻量级、可移植和可扩展的方式来

运行应用程序。

再者，持续交付的理念迫使开发人员频繁地进行代码提交、构建和部署，以保持软件产品的快速迭代和质量保证。而传统的单体应用往往需要整体构建和部署，导致开发和测试周期较长，无法满足快速迭代的需求。

基于上述背景，微服务架构应运而生。



服务大了太臃肿，要拆成若干个小系统，然后进行分而治之（例如北京一个火车站到多个火车站），同时解决因并发访问过大带来的系统复杂性(例如:业务,开发,测试,升级,可靠性等)。这样分了以后，可以把每个服务作为一个独立的开发项目，由团队进行快速开发、迭代升级。

5.1.3 微服务架构可能带来什么问题？

虽然微服务架构带来了很多好处，但也可能带来一些问题，包括以下几个方面：

1. 系统复杂性增加：由于微服务架构将应用程序拆分为多个服务，每个服务都需要独立开发、部署和维护，因此系统的整体复杂性增加。开发团队需要处理跨服务的通信、数据一致性、分布式事务等复杂性挑战，对开发和运维人员的要求更高。
2. 服务间通信开销：微服务架构依赖于轻量级的通信机制，如HTTP或消息队列。虽然这种通信方式很灵活，但也会带来一定的通信开销。大量的跨服务通信可能会导致网络延迟增加、系统性能下降。
3. 数据一致性问题：由于微服务架构中每个服务都有自己的数据库，数据一致性成为一个挑战。当多个服务需要共享数据时，需要采取一些机制来确保数据的一致性，如使用分布式事务、事件驱动架构等。但这些机制会增加系统的复杂性和开发成本。
4. 对运维的挑战：微服务架构使得系统的部署和维护变得更加复杂。需要考虑如何管理和监控多个微服务的运行状态、如何进行版本管理和回滚、如何保证整个系统的高可用性等。这对于运维团队来说可能是一项挑战。
5. 增加了开发和测试的复杂度：每个微服务都是独立的，都需要独立开发和测试。这意味着开发团队需要具备分布式系统开发和测试的能力，并且需要投入更多的时间和资源来进行集成测试和端到端的测试。



总而言之，微服务架构的引入对于大型和复杂的系统来说是有益的，但也需要权衡其带来的复杂性、开发成本和运维挑战。在决定采用微服务架构时，团队需要仔细评估其是否适合当前的业务需求和团队能力。

5.1.4 微服务架构中有哪些关键组件？

微服务架构设计时，需要一些关键组件和技术，包括：

- 1. 服务：**微服务架构将应用程序拆分为多个小型、自治的服务。每个服务都负责一个特定的业务功能，并且可以独立开发、部署、扩展和管理。每个服务都应该有明确定义的接口和清晰的边界。
- 2. 服务注册与发现：**微服务架构中的服务需要能够自动注册和发现其他服务的位置。服务注册与发现组件提供了服务的注册和发现功能，使得不同服务能够在运行时动态地发现和通信。
- 3. 负载均衡：**微服务架构中的服务通常是水平扩展的，可能会有多个实例运行。负载均衡组件可以根据请求的负载情况，将请求分发到不同的服务实例上，以实现负载均衡和高可用性。
- 4. 网关：**微服务架构中的服务可以非常细粒度地拆分，这可能导致客户端需要发起大量的请求才能完成一个功能。网关组件可以作为服务的统一入口，对外暴露简化的接口，将多个请求合并为一个或少量请求，减少客户端与服务的交互次数。
- 5. 分布式数据管理：**微服务架构中每个服务可能拥有自己的数据存储，数据一致性成为一个挑战。分布式数据管理组件可以帮助处理数据的复制、同步、共享和一致性问题，如分布式数据库、缓存和消息中间件等。
- 6. 容器化技术：**容器化技术如Docker，可以帮助打包、传输、部署和管理微服务应用。它提供了一种轻量级、可移植和可扩展的方式来运行应用程序，简化了部署和维护的过程。
- 7. 消息队列/事件驱动：**微服务架构中可以使用消息队列或事件驱动的方式进行服务间的通信和解耦。消息队列通过发布/订阅模型传递消息，而事件驱动则通过事件的触发和监听来实现服务间的解耦。



除了上述关键组件，还需要考虑监控和日志记录、安全和权限控制、自动化部署和持续交付等方面的工具和技术。微服务架构并没有固定的架构模式，组件选择和架构设计会根据具体的业务需求和技术栈进行调整。

5.1.5 如何理解Spring Cloud与Spring Cloud Alibaba？

Spring Cloud和Spring Cloud Alibaba是两个与微服务相关的项目，它们都是基于Spring Framework构建的，用于简化微服务架构开发和部署的工具集。

1. Spring Cloud：Spring Cloud是一个开源的微服务框架，它提供了一套丰富的组件和模块，用于解决微服务架构中的各种分布式系统开发和管理的需求。它包括服务注册与发现、负载均衡、熔断器、配置管理、消息总线、断路器等组件，可以帮助开发者快速构建和部署可伸缩、弹性和可靠的微服务应用。

2. Spring Cloud Alibaba: Spring Cloud Alibaba是在Spring Cloud基础上扩展出的一个项目，它与阿里巴巴的一些云原生开源产品进行了深度集成，为开发者提供了更多的选择和功能。Spring Cloud Alibaba包含了服务注册与发现、配置管理、服务熔断、分布式事务等组件，同时也集成了阿里云的消息队列、分布式数据库、分布式锁等云原生产品服务，使开发者能够更方便地构建在阿里云上运行的微服务应用。



总结来说，Spring Cloud和Spring Cloud Alibaba是两个在开发和管理微服务应用方面提供支持的框架。Spring Cloud是一个开放的微服务框架，而Spring Cloud Alibaba是在Spring Cloud基础上扩展并与阿里巴巴生态系统进行深度整合的一个微服务解决方案。开发者可以根据实际需求来选择适合的框架和组件来构建和部署微服务应用。

5.2 注册中心

5.2.1 什么是服务注册中心？

注册中心是分布式系统中常见的一种服务发现和管理组件。它充当了服务提供者和服务消费者之间的桥梁，用于管理和协调各个服务的注册、发现和通信。。它充当着服务实例的目录，记录了每个服务实例的网络位置（主机和端口），以便其他服务或客户端可以通过服务注册中心来发现和访问这些服务。

在微服务架构中，服务通常是水平扩展的，可能会有多个实例同时运行。服务注册中心提供了一个集中的位置，用于注册和存储服务实例的相关信息。当服务实例启动或关闭时，它会向服务注册中心注册或注销自己的信息。服务注册中心还可以监控服务的状态，并提供一些负载均衡的策略，以便其他服务可以在运行时选择合适的服务实例进行调用。

服务注册中心有多种实现方式，比如使用ZooKeeper、Consul、Eureka和etcd等。它们提供了各种功能和特性，如服务注册与发现、健康检查、负载均衡和故障恢复等。通过使用服务注册中心，微服务架构可以实现服务实例的动态发现和弹性扩展，提供了更好的可靠性和可伸缩性。

5.2.2 服务注册中心诞生的背景？

注册中心的诞生背景可以追溯到分布式系统的发展和微服务架构的兴起。


在传统的单体应用架构中，应用程序通常是以单个单体应用的形式部署和运行的，所有的功能模块都集中在一个应用中。随着业务的增长和用户量的增加，单体应用面临性能、可扩展性和可靠性等方面

的挑战。

于是，微服务架构应运而生。微服务架构将应用拆分为一组小而独立的服务，每个服务负责一个小的业务功能。每个服务都可以独立进行开发、部署和扩展，通过轻量级的通信机制进行交互。这种架构风格带来了很多好处，如灵活性、松耦合、可伸缩性和快速开发等。

然而，随着微服务的增多，服务之间的通信和调用变得复杂。传统的硬编码方式不再适用。这就需要一种机制来管理和维护服务实例的信息，以便其他服务可以在运行时发现并调用这些服务。这就是服务注册中心的诞生背景。

服务注册中心提供了一种集中式的方式来注册、存储和查询服务实例的信息，以便其他服务可以根据需要动态地发现和调用这些服务。它解决了微服务架构中服务发现和负载均衡的问题，为分布式系统提供了便利和灵活性。

 总结来说，服务注册中心本质上就是存储服务信息的一个服务,也可以理解为一个中介(这里用到了中介者模式)。它的诞生是为了应对微服务架构中分布式系统的挑战，提供了一种机制来管理和维护服务实例的信息，实现服务的动态发现和调用。它是微服务架构中的关键组件，为分布式系统的可靠性和可伸缩性提供了支持。例如所有公司需要在工商局进行备案，淘宝也可以理解为买家和卖家的注册中心。

5.2.3 服务注册中心会提供哪些功能？

注册中心的出现，使得服务提供者和服务消费者之间解耦，简化了分布式系统的开发和部署。它提供的功能包括：

1. 服务注册：服务实例启动时向注册中心注册自己的信息，包括服务名称、网络地址和端口号等。
2. 服务发现：其他服务或客户端可以通过查询注册中心来获取可用的服务实例信息，从而进行服务调用。
3. 健康检查：注册中心可以定期检查服务实例的健康状态，包括检查服务是否存活、可用性如何等。这样可以及时发现故障或不可用的实例，并做出相应的处理。
4. 负载均衡：注册中心可以提供负载均衡的策略，根据服务实例的负载、性能等指标来选择合适的实例进行服务调用，从而实现负载均衡。

5. 故障恢复：注册中心可以监控服务实例的状态变化，当发现有实例宕机或故障时，可以及时将该实例从注册中心中剔除，避免其他服务调用到不可用的实例上。
6. 配置管理：注册中心可以用于集中管理服务的配置信息，包括环境变量、属性文件等配置内容。服务实例可以从注册中心获取配置信息，实现统一的配置管理。
7. 监控和统计：注册中心可以收集服务实例的运行状态、调用次数、响应时间等指标，并提供监控和统计功能，帮助开发人员进行性能优化和故障排查。



总的来说，服务注册中心提供了服务注册与发现、负载均衡、健康检查、故障恢复、配置管理等功能，为微服务架构提供了服务实例的管理和控制能力。它是实现微服务架构中分布式系统的可靠性、可伸缩性和弹性扩展的重要组件。不同的服务注册中心实现可能会提供略有差异的功能和特性。

5.2.4 服务注册中心你是如何选型的？

选择适合的服务注册中心需要考虑多个因素，包括以下几个方面：

1. 功能需求：根据项目或系统的实际需求，明确所需的服务注册中心功能，例如服务发现、负载均衡、健康检查等。
2. 生态系统支持：考虑选型的服务注册中心是否有丰富的生态系统和社区支持，例如是否有成熟的文档、教程、示例代码、第三方工具和插件等。
3. 可靠性和性能：评估注册中心的可靠性和性能，包括其在高负载条件下的吞吐量、延迟、容错机制等。
4. 扩展性和可伸缩性：考虑注册中心是否具备良好的扩展性和可伸缩性，能够应对系统的增长和变化。
5. 社区活跃度：考察注册中心的社区活跃度和项目的更新维护情况，判断其未来发展的可持续性。
6. 安全性：确保注册中心能够提供合适的安全措施，如身份认证、访问控制、数据加密等，保护服务实例和数据的安全。
7. 工具和集成支持：检查注册中心是否提供与其他常用工具和框架的集成支持，例如与容器编排工具（如Kubernetes）、配置中心（如Spring Cloud Config）等的兼容性。
8. 技术栈匹配：考虑选型的注册中心是否与项目使用的开发语言、框架和技术栈相匹配，以便更好地集成和开发。



最终的选型应该根据项目的具体需求和实际情况进行评估和权衡。常见的服务注册中心包括ZooKeeper、Consul、Eureka、etcd等，每个都有其特点和适用场景。同时也要考虑使用微

服务框架（如Spring Cloud、Dapr等）时的默认注册中心选择。

5.2.5 说说你对Nacos的理解？

Nacos是一个开源的动态服务发现、配置管理和服务管理平台。它提供了服务注册和发现、配置管理、动态DNS和服务治理等功能，是构建云原生应用和微服务架构的重要组成部分。

Nacos以可靠性、可扩展性和易用性为设计目标，具有以下特点：

1. 服务注册和发现：Nacos支持多种服务注册和发现机制，包括基于DNS、HTTP和RPC的服务注册。它提供了统一的注册接口和查询接口，能够动态地注册和发现服务实例，并提供多种负载均衡策略来实现服务调用。
2. 配置管理：Nacos支持动态的配置管理，可以将应用的配置信息存储在注册中心中，并根据需要动态地获取和更新配置。它提供了配置推送、配置监听和配置分组等功能，方便进行配置的管理和更新。
3. 动态DNS：Nacos支持将服务名称映射到IP地址的动态DNS功能，从而实现了服务的动态发现和访问。通过使用动态DNS，服务消费者可以使用服务名称来访问服务实例，而不需要关注具体的IP地址和端口号。
4. 服务治理：Nacos提供了一些服务治理的功能，包括健康检查、流量管理、服务降级和限流等。这些功能可以帮助开发人员更好地管理和保护服务，提高系统的可靠性和稳定性。
5. 可扩展性：Nacos支持水平扩展，可以根据需要增加节点来提高性能和可用性。它使用了基于Raft算法的分布式一致性协议，保证了数据的一致性和可用性。



总之，Nacos是Alibaba基于SpringBoot技术实现的一个功能强大的服务注册和配置管理平台，为构建云原生应用和微服务架构提供了便捷而可靠的解决方案。它具有良好的社区支持和广泛的用户基础，并且持续更新和改进，能够满足不同规模和复杂度的分布式系统的需求。

5.2.6 Nacos如何检测服务状态？

Nacos使用心跳机制来检测服务的状态。具体来说，Nacos通过以下方式进行服务状态的检测：

1. 心跳注册：服务实例在启动时会向Nacos注册中心发送心跳注册信息，包括实例的IP地址、端口号、健康检查地址等。注册成功后，Nacos会保存这些信息。
2. 心跳保活：注册成功后，服务实例会周期性地向Nacos发送心跳请求，证明自己还活着。这个心跳请求的频率由实例配置的心跳间隔时间决定。

3. 心跳检测：Nacos在接收到服务实例的心跳请求后会更新实例的心跳时间戳，并记录最近一次心跳的时间。如果一段时间内没有收到实例的心跳请求，Nacos会认为该实例不可用。
4. 服务下线：如果服务实例正常停止或异常终止，并没有发送下线通知，Nacos会根据心跳检测的结果，超过一定时间后将该实例标记为离线或不可用。



通过心跳机制，Nacos可以实时监测服务实例的状态。当注册中心发现服务实例状态的变化（上线、下线或不可用），它会通知其他相关服务或客户端，以便它们可以及时更新服务的可用列表，保持服务调用的正确性和稳定性。同时，Nacos还提供了可视化的界面，实时展示服务的健康状态和变化情况，方便监控和管理。

5.2.7 Nacos注册中心有哪些常用配置？

我们可以在项目的application.yml中进行服务的注册配置，例如：

```
1 spring:
2   application:
3     # 定义服务名称(注册到nacos中的服务)
4     name: xxxx
5   cloud:
6     nacos:
7       discovery:
8         # 可以配置多个，逗号分隔
9         server-addr: localhost:8848
10        # nacos客户端向服务端发送心跳的时间间隔，时间单位其实是ms
11        heart-beat-interval: 5000
12        # 服务端没有接收客户端心跳请求就将其设为不健康的时间间隔，默认为15s
13        # 注：推荐值该值为15s即可，如果有的业务线希望服务下线或者出故障时希望尽快被发现，i
14        heart-beat-timeout: 20000
15        # 客户端在启动时是否读取本地配置项(一个文件)来获取服务列表
16        # 注：推荐该值为false，若改成true。则客户端会在本地的一个文件中保存服务信息，
17 # 当下次宕机启动时，会优先读取本地的配置对外提供服务。
18        naming-load-cache-at-start: false
```

5.3 服务调用分析

5.3.1 如何理解服务调用？

服务调用是指在分布式系统中，一个服务通过网络请求调用另一个服务提供的功能或资源。通常情况下，服务提供者和服务消费者分别运行在不同的进程或机器上，并通过网络进行通信。

在服务调用的过程中，服务消费者需要知道服务提供者的地址和接口信息。一般来说，服务提供者会将自己的服务注册到服务注册中心中，服务消费者通过服务注册中心查询到服务提供者的信息，然后通过网络进行请求和响应。

服务调用可以是同步的或异步的。在同步调用中，服务消费者发送请求后会一直等待服务提供者的响应，直到收到响应或超时。在异步调用中，服务消费者发送请求后不需要等待响应，可以继续处理其他业务逻辑，当服务提供者完成处理后，会通过回调或消息通知的方式将结果返回给服务消费者。

服务调用通常涉及到一些重要的概念和技术，如负载均衡、服务发现、熔断器、容错机制等，用于提高系统的可用性、性能和稳定性。



总之，通过服务调用，不同的服务可以相互协作，实现更复杂的业务逻辑。它是实现分布式系统的核心机制之一，也是微服务架构中的重要组成部分。

5.3.2 服务调用的基本过程是怎样的？

服务调用的过程大致可以分为以下几个步骤：

1. 服务发现：服务调用方会从服务注册中心获取服务提供方的可用实例列表。服务注册中心是一个集中管理服务实例信息的组件，服务提供方会将自己的网络地址、服务信息等注册到服务注册中心，而服务调用方可以通过查询服务注册中心来获取服务的可用地址。
2. 负载均衡：在获取到服务提供方的可用地址列表后，服务调用方需要选择一个实例来发送请求。这涉及到负载均衡算法的选择，负载均衡算法可以根据一定的策略，如轮询、随机等，在可用实例列表选择一个实例进行调用。负载均衡可以实现服务请求的分发，以避免某个实例负载过高，提高系统的可用性和性能。
3. 请求发送：一旦选择了要调用的服务实例，服务调用方会构造请求（包含请求方法、请求路径、请求体等）并将其发送到目标服务实例。请求可以基于标准的HTTP协议，也可以基于其他的RPC框架如gRPC，具体的协议和实现方式可以根据实际情况和需求确定。
4. 请求处理：服务提供方接收到请求后，会根据请求的信息进行相应的处理。这包括执行业务逻辑、读取数据库、调用其他服务等。处理完成后，服务提供方会将处理结果封装在响应中返回给服务调用方。

5. 响应接收：服务调用方接收到服务提供方返回的响应后，会对响应进行解析和处理。根据响应状态码、响应头和响应体等，服务调用方可以判断请求是否成功，如果成功则可以获取返回的数据，并进行相应的后续操作。如果请求失败或发生异常，服务调用方需要根据具体情况进行适当的处理，如重试、使用备份数据等。

值得注意的是，服务调用过程中还需要考虑各种异常情况的处理，如网络故障、服务不可用、超时等，这些异常情况可能会对服务调用的可靠性和性能造成影响。因此，服务调用方通常会使用熔断、降级、重试等机制来增加系统的容错性和稳定性。



总的来说，服务调用是分布式系统中实现服务协作的核心机制，通过服务发现、负载均衡、请求发送、请求处理和响应接收等步骤，不同的服务实例可以相互调用，实现系统的业务逻辑。

5.3.3 说说什么是Dubbo?

Dubbo是一个高性能的服务框架，用于提供分布式系统中的可靠的RPC（远程过程调用）通信和服务治理。它最初是由阿里巴巴公司开发的，后来成为了Apache的顶级项目。

Dubbo提供了一整套分布式服务框架的解决方案，包括服务注册与发现、负载均衡、容错机制、服务调用、服务监控等。它的目标是简化分布式系统的开发和部署，提高系统的灵活性、可扩展性和性能。

Dubbo在设计上注重了可扩展性和可定制性，它提供了丰富的扩展点和插件机制，可以根据不同的业务需求进行定制和扩展。同时，Dubbo还支持多种协议和序列化方式，可以与各种服务框架和语言进行集成。

在Dubbo中，服务提供者将自己的服务注册到注册中心，服务消费者通过注册中心获取到服务提供者的地址和接口信息，然后通过网络进行调用。Dubbo提供了基于配置的服务调用和基于注解的服务调用两种方式，可以根据具体的需求选择使用。



总之，Dubbo是一个功能强大的分布式服务框架，可以帮助开发者构建可靠的分布式系统，并提供了丰富的功能和扩展点，使得系统的开发、部署和维护变得更加简单和高效。

5.3.4 说说Dubbo的服务请求过程?

Dubbo的服务请求流程可以简要概括为以下几个步骤：

1. 服务提供者启动，将自己的服务注册到注册中心。注册中心负责维护服务提供者的地址信息。
2. 服务消费者启动，从注册中心获取可用的服务提供者地址列表。
3. 服务调用过程：
 - 服务消费者选择一个合适的负载均衡策略，从服务提供者地址列表选择一个可用的提供者。
 - 服务消费者将请求封装成Invocation对象，并发送给服务提供者。
 - 服务消费者和服务提供者之间通过网络进行通信，一般使用Dubbo的RPC框架进行远程调用。
 - 服务提供者接收到请求后，进行相应的处理，并返回结果。
 - 服务消费者接收到结果后，将结果返回给调用方。
4. 容错和熔断：
 - 如果服务调用过程中发生错误，Dubbo提供了多种容错策略来保证服务的可靠性。
 - 如果某个提供者多次调用失败，Dubbo会自动开启断路器，避免继续调用失败的提供者。




总体来说，Dubbo的服务请求流程通过服务注册和发现、负载均衡、远程调用等机制，将服务消费者和服务提供者连接起来，并提供了容错和熔断机制来保障服务的可靠性。

5.3.5 Dubbo支持哪些协议？

Dubbo支持以下几种协议：

1. Dubbo协议：Dubbo协议是Dubbo RPC框架默认使用的协议。它基于TCP长连接，采用自定义的通信协议，具有较高的性能和可靠性。
2. RMI协议：RMI（Remote Method Invocation）是Java标准库中提供的远程调用协议。Dubbo支持使用RMI协议进行远程调用。
3. Hessian协议：Hessian是一种二进制RPC协议，它使用较少的字节来序列化对象，并支持压缩。Dubbo支持使用Hessian协议进行远程调用。
4. HTTP协议：Dubbo支持使用HTTP协议进行远程调用。通过HTTP协议，Dubbo可以与其他语言的RPC框架进行互操作。
5. Webservice协议：Dubbo支持使用Webservice协议进行远程调用。Webservice是一种跨平台、松耦合的远程调用协议，支持多种数据格式和传输协议。

除了这些协议，Dubbo还支持自定义的协议扩展。用户可以通过实现Dubbo的SPI接口来自定义协议，并在Dubbo中进行注册和使用。


 注意：在Dubbo的最新版本中，默认推荐使用Dubbo协议，因为它在性能和功能上都有更好的表现。其他协议仍然存在，但可能在未来版本中逐渐淘汰或限制使用。

5.3.6 Dubbo支持哪些负载均衡策略？

Dubbo支持多种负载均衡策略，可以根据实际场景和需求选择适合的策略。常用负载均衡策略如下：

1. 随机(Random)：随机选择一个可用的服务提供者进行调用。
2. 轮询(RoundRobin)：按照顺序依次选择可用的服务提供者进行调用。
3. 最少活跃数(LeastActive)：选择活跃请求数最少的服务提供者进行调用。活跃请求数表示当前正在处理的请求量，越少越好。
4. 一致性哈希(ConsistentHash)：根据请求的参数或标识，通过哈希算法选择相应的服务提供者进行调用。
5. 加权随机(WeightRandom)：根据服务提供者的权重，按照权重随机选择一个可用的服务提供者进行调用。
6. 加权轮询(WeightRoundRobin)：根据服务提供者的权重，按照权重顺序选择可用的服务提供者进行调用。

此外，Dubbo还支持自定义的负载均衡策略。用户可以通过实现Dubbo的LoadBalance接口来自定义负载均衡算法，并在Dubbo中进行注册和使用。


 需要注意的是，Dubbo在配置上也能对每个服务进行单独的负载均衡策略配置，可以根据具体的业务需求灵活选择适合的负载均衡策略。

5.3.7 Dubbo支持哪些容错策略？

Dubbo支持多种容错策略，这些策略用于处理服务调用失败或超时时的方式。下面是Dubbo支持的常用容错策略：

1. Failover容错策略：如果服务调用失败，Failover策略会自动切换到另一个可用的服务提供者进行重试，直到调用成功或达到最大重试次数。这是Dubbo的默认容错策略。
2. Failfast容错策略：Failfast策略是一种快速失败的策略。它在服务调用失败后立即返回异常，不进行重试。适用于对服务调用的响应时间敏感的场景。

3. Failsafe容错策略：Failsafe策略在服务调用失败时，直接忽略错误，不进行任何处理。适用于日志记录、监控统计等不重要的业务场景。
4. Failback容错策略：Failback策略在服务调用失败后，会记录失败请求，并定期重发。适用于对服务调用的可靠性要求较高的场景。
5. Forking容错策略：Forking策略会同时向多个服务提供者发起调用请求，只要有一个服务提供者成功返回结果，就会立即返回。适用于对服务调用的响应时间要求较高的场景。
6. Broadcast容错策略：Broadcast策略会向所有可用的服务提供者发送调用请求，然后将结果合并返回。适用于广播通知等场景。


 用户可以在Dubbo的服务提供者和消费者配置中指定所需的容错策略，并根据具体的业务需求进行选择和调整。

5.3.8 说说Dubbo的代理策略？

Dubbo的代理策略是指在服务消费者端与服务提供者端之间进行远程调用时，Dubbo的代理生成方式和调用方式。Dubbo支持两种代理策略：JDK动态代理和CGLIB动态代理。

1. JDK动态代理：当服务接口有实现类时，Dubbo会使用JDK动态代理来生成服务接口的代理对象。JDK动态代理要求被代理的对象实现一个或多个接口，在运行时生成代理对象。JDK动态代理是基于接口的，它通过反射机制实现代理，具有较高的性能和较小的内存开销。
2. CGLIB动态代理：当服务接口没有实现类时，Dubbo会使用CGLIB动态代理来生成服务接口的代理对象。CGLIB动态代理可以生成一个目标类的子类作为代理类，不需要目标类实现接口。CGLIB动态代理是基于类的，通过继承目标类并重写其中的方法来实现代理。

Dubbo会根据代理策略以及服务接口是否有实现类来决定使用JDK动态代理还是CGLIB动态代理。用户可以在Dubbo的服务提供者和消费者配置中指定所需的代理策略，默认情况下Dubbo会自动根据条件选择合适的代理策略。

 需要注意的是，对于不同的代理策略，Dubbo在调用方式上也有所差异。对于JDK动态代理，Dubbo使用同步调用方式进行远程调用；而对于CGLIB动态代理，Dubbo使用异步调用方式进行远程调用。用户可以根据具体的业务需求选择合适的代理策略和调用方式。

5.3.9 注册中心挂了，服务还能调用吗？

可以，因为刚开始初始化的时候，consumer会将需要的所有提供者的地址等信息拉取到本地缓存，所以注册中心挂了可以继续通信。但是provider挂了，那就没法调用了。

5.3.10 说说Feign和OpenFeign?

Feign和OpenFeign都是基于Java的声明式、模板化的HTTP客户端框架，用于简化服务间的HTTP通信。它们可以与Spring Cloud等微服务框架集成，帮助开发者快速、方便地进行服务调用。

Feign最初是由Netflix开发的，它在RestTemplate的基础上做了进一步的封装。旨在简化微服务架构中的服务调用。它基于注解和接口定义的方式，使得开发者可以通过简单的接口定义和注解配置，实现对远程服务的调用。Feign底层使用了Ribbon作为负载均衡的组件，并且集成了Hystrix来实现容错和熔断机制。

OpenFeign是在Feign的基础上进行改进的项目，由Spring Cloud团队维护和推进。OpenFeign在保持Feign原有的简单和易用性的基础上，增加了更多的功能和扩展点。相对于Feign，OpenFeign更加灵活和可扩展，可以通过自定义注解、拦截器等方式实现更多的定制化需求，并且支持更多的配置选项。

Feign和OpenFeign都遵循了声明式、模板化的设计理念，简化了服务调用的过程，开发者只需要定义接口和注解，在运行时会自动生成实现逻辑，使得服务调用的代码更加清晰和简洁。同时，它们还支持负载均衡、熔断器等重要的功能，提高了系统的可靠性和性能。



总结来说，Feign和OpenFeign都是方便的HTTP客户端框架，用于简化微服务架构中的服务调用。OpenFeign在Feign的基础上增加了更多的功能和扩展点，更为灵活和可定制。开发者可以根据具体的需求选择使用。

5.3.11 说说Feign和Dubbo的异同点?

Feign和Dubbo都是用于构建分布式应用程序和实现微服务架构的开源框架，其相同点和不同点如下：


相同点：

- 都提供了便捷的服务调用方式，通过接口代理实现远程服务的调用，隐藏了底层通信细节。
- 都支持负载均衡功能，可根据配置的负载均衡策略将请求分发到不同的服务实例上。

不同点：

- 技术栈和生态系统：Feign与Spring框架和Spring Cloud的生态系统紧密集成。而Dubbo有自己独立的技术栈和生态系统，与Spring没有强依赖关系。

- 协议和序列化：Feign使用HTTP协议作为默认的远程通信协议，采用基于Restful风格的接口调用，具有简单、轻量、易用等特点。而Dubbo默认使用自定义的Dubbo协议，支持多种通信协议，如dubbo、rmi、hessian等。另外，Dubbo还提供了多种序列化机制可选，适合大规模分布式系统，具有高性能、丰富的配置、容错机制等特点。
- 注册中心：Feign依赖于Spring Cloud的服务注册与发现组件。而Dubbo有自己的注册中心，可以选择使用Zookeeper或Nacos等作为注册中心。
- 熔断机制：Feign默认集成了Netflix的Hystrix来实现熔断机制，可防止分布式系统由于服务之间的故障导致的服务雪崩。而Dubbo有自己的熔断机制，通过限流和降级来保护系统的可用性。

 总之，Feign 和 Dubbo 都是优秀的微服务架构下的远程调用框架，各有特点。Feign 适合调用 RESTful API 接口，具有简单、轻量、易用等特点。Dubbo 适合大规模分布式系统，具有高性能、丰富的配置、容错机制等特点。在实际应用中，还需要根据具体业务需求和技术栈进行权衡和选择，以便最终达到最优的性能和效果。

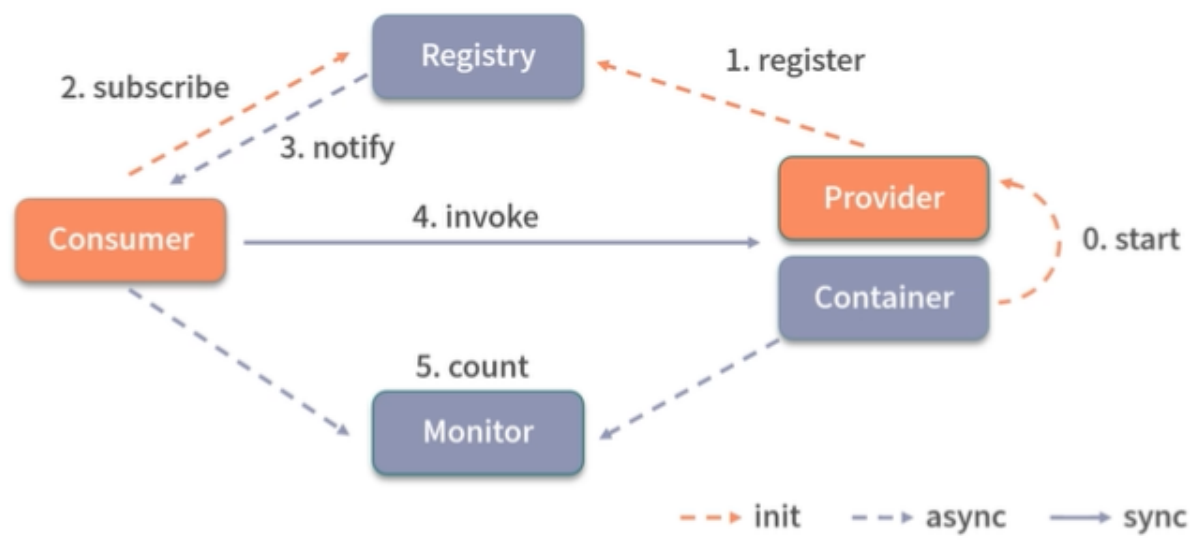
5.3.12 说说你对负载均衡的理解？

负载均衡是一种将网络流量、请求或工作负载均匀分配给多个服务器或计算资源的技术。它旨在提高系统的性能、可靠性和可扩展性。常见的负载均衡算法有：

1. 轮询（Round Robin）算法：这是一种最简单的负载均衡算法，按照轮询的方式将请求依次分发给每个服务器。每个请求都会按顺序发送到下一个服务器，从而实现请求的均衡分配。
2. 最少连接（Least Connection）算法：该算法会将请求发送到当前连接数最少的服务器上。通过动态地选择连接数最少的服务器，可以更加均衡地分配请求，避免过载。
3. 源IP哈希（Source IP Hash）算法：该算法通过对请求的源IP地址进行哈希计算，将请求分发到相应的服务器。这样相同的IP地址的请求总是会被分发到同一台服务器上，可以实现一定程度的会话保持。
4. 动态权重（Weighted Round Robin）算法：该算法会为每个服务器分配一个权重值，根据权重值决定分发请求的比例。具有较高权重的服务器将接收到更多的请求，从而实现根据服务器性能分配负载的目的。
5. IP散列（IP Hash）算法：该算法通过对客户端IP地址进行哈希计算，将请求分发到特定的服务器上。这样相同的客户端IP地址的请求总是会被分发到同一台服务器上，可以实现一定程度的会话保持。
6. 加权最少连接（Weighted Least Connection）算法：该算法结合了最少连接算法和动态权重算法。它会根据服务器的连接数和权重来进行决策，将请求发送到当前连接数最少且具有较高权重的服务器上。

这些负载均衡算法各有特点，可以根据系统的需求和实际情况选择适合的算法来实现负载均衡。

5.3.13 Dubbo架构是怎样的？



5.4 配置中心部分

5.4.1 什么是配置中心？


配置中心是一种用于集中管理和动态调整系统配置的工具或服务。在分布式系统中，不同服务通常有各自的配置参数，包括数据库连接信息、服务器地址、日志级别等。而配置中心的作用就是集中管理这些配置信息，通过动态调整配置的方式，实现系统各服务配置统一、分布式配置的协同更新和配置的动态调整。

5.4.2 为什么需要配置中心？

配置中心的存在有以下几个主要原因：

- 1. 集中管理：**在分布式系统中，不同的组件和模块有各自的配置参数，如果每个组件都需要自行管理配置，容易导致配置信息分散、难以管理和维护。配置中心可以集中管理和统一配置信息，提高配置的可维护性。
- 2. 动态调整：**系统中的配置信息通常需要根据实际情况进行动态调整，例如数据库连接信息、缓存配置、日志级别等。如果每次调整配置都需要重新部署或重启整个系统，会造成服务中断和影响系统的稳定性。而配置中心可以通过动态更新配置并通知组件进行调整，实现配置的动态调整和实时生效，提高系统的灵活性和可用性。
- 3. 配置共享和复用：**不同的系统或模块可能会有一些共同的配置信息，例如数据库连接地址、认证信息等。通过配置中心，可以将这些共享的配置信息集中管理，提高配置的复用性，避免重复配置和错误配置。


4. **版本控制和回滚：**配置中心通常支持配置的版本控制，可以记录每次配置变更的历史，并支持回滚和回退。这样在出现配置错误或不符预期时，可以方便地回滚到之前的版本，减少风险和影响。
5. **安全性和权限控制：**配置中心可以提供安全性和权限控制机制，保护配置信息的机密性和完整性，并给予不同角色或用户不同的访问和修改权限。这样可以确保配置信息的合规性和安全性。

 总之，配置中心可以有效简化系统的配置管理，提高系统的稳定性和灵活性，同时也方便了运维人员对系统的监控和管理。常见的配置中心产品包括Apollo、Spring Cloud Config、Zookeeper等。

5.4.3 你在配置中心配置过什么内容？

配置中心可以配置包括但不限于以下内容：

1. 服务参数配置：包括服务的端口号、超时时间、线程池大小等。
2. 数据库连接信息：包括数据库的地址、端口号、用户名、密码等。
3. 缓存配置：例如缓存的类型、过期时间、最大缓存数等。
4. 日志级别和日志输出配置：包括日志的打印级别、输出路径等。
5. 服务注册和发现：指定服务的注册中心地址、协议等。
6. 授权认证配置：例如接口的访问权限、令牌等。
7. 环境配置：包括开发环境、测试环境、生产环境的配置信息。
8. 其他业务相关配置：例如邮件服务器配置、消息队列的配置等。

 总的来说，配置中心可以配置系统中各个组件和模块所需的各项参数和配置信息，以满足不同环境和需求下的灵活配置和调整。具体可以根据实际需求和系统架构进行灵活的扩展和定制。

5.4.4 为什么要定义bootstrap.yml文件？

在使用配置中心时，定义bootstrap.yml文件是为了在系统启动阶段先加载配置中心的配置，确保配置中心的配置能够被应用到系统中。bootstrap.yml是Spring Boot框架提供的用于系统初始化的配置文件。与application.yml相比，bootstrap.yml的加载顺序更早，因此可以在系统启动前加载配置中心的配置。

配置中心的配置通常包括系统的基本配置，例如注册中心地址、配置中心地址、敏感配置等。将这些配置放在bootstrap.yml中，可以确保这些配置能够在系统初始化阶段就可用，避免系统启动后才加载配置中心的配置，导致系统在启动阶段无法使用某些关键配置。

另外，bootstrap.yml也可以用来配置一些启动参数，例如系统相关的启动环境和配置文件的位置等。



总之，通过定义bootstrap.yml文件并在其中加载配置中心的配置，可以确保配置中心的配置能够在系统启动前被正确加载和使用，保证系统的正常运行。

5.4.5 配置中心宕机了，还可以读到配置信息吗？

配置中心的数据可以在服务本地存储一份，所以配置中心宕机了，可以从本地内存读取。假如是集群部署，如果某个节点宕机了，其他节点仍然可以继续提供配置信息的读取服务，因此当一个Nacos节点宕机后，仍然可以从其他节点读取到配置信息。

5.4.6 微服务如何感知配置中心的配置的变化？

1.4.x版本中的nacos客户端会基于长轮询机制从nacos获取配置信息。这个轮询可以这样去理解，我们骑着共享单车去火车站买票，但是票卖完了，一种方式是直接返回（这种方式称之为短轮询）。还有一种方式在火车站的椅子上等一会，看看是否还会放票，是否有人会退票等。那这种机制就是长轮询。

Nacos作为配置中心提供了订阅功能，服务可以通过订阅配置的方式来感知配置的变化。当配置中心的配置发生变化时，Nacos会主动推送变更通知给订阅了该配置的服务实例。服务收到变更通知后可以进行相应的更新操作。订阅机制可以更加高效地实现配置的实时更新，并减少了额外的网络开销。


5.4.7 Nacos的配置管理模型是怎样的？

Nacos的配置管理模型主要包括三个关键概念：数据ID、分组（Group）和命名空间（Namespace）。

数据ID (Data ID)： 数据ID是配置在Nacos中唯一标识某个配置的字符串，可以通过数据ID来唯一地找到一条配置数据。数据ID可以是任意字符串，在同一个分组 (Group) 下必须唯一。通常，我们可以使用应用程序的名称或模块名作为数据ID。

分组 (Group)： 分组是一种对配置进行逻辑分组的机制。一个分组可以包含多个配置项，用于更好地组织和管理配置数据。Nacos默认使用"default"作为默认分组，可以根据业务需求进行分组的划分。

命名空间 (Namespace)： 命名空间是为了支持配置和服务的多环境和多团队管理而提供的一种隔离机制。不同的命名空间之间的配置是相互隔离的，一些配置在某个命名空间中可用，但在其他命名空间中可能不存在或不可见。命名空间可以用于灰度发布、从一个环境到另一个环境的迁移等场景。

 总之，通过使用数据ID、分组和命名空间，Nacos提供了灵活的配置管理机制，可以对配置进行细粒度的控制和访问。在nacos中的配置管理模型中，一个namespace可以有多个分组，每个分组下又可以有多个服务实例。

5.4.8 Nacos配置中心基础配置有哪些？

在Nacos配置中心中，基础配置包括以下几个方面：

1. **服务器地址 (Server Address)：** 配置Nacos服务器的地址和端口。这是配置中心的访问入口，客户端需要知道Nacos服务器的地址才能与其进行通信。
2. **命名空间 (Namespace)：** 命名空间是为了支持配置和服务的多环境和多团队管理而提供的一种隔离机制。通过配置命名空间，可以对不同环境或不同团队的配置进行隔离和管理。
3. **分组 (Group)：** 分组是一种对配置进行逻辑分组的机制。可以使用分组来组织和管理不同的配置数据，方便查找和管理。默认情况下，Nacos使用"default"作为默认分组。
4. **Data ID：** 数据ID是配置在Nacos中唯一标识某个配置的字符串。通过数据ID，可以唯一地找到并访问某个具体的配置数据。
5. **配置内容 (Configuration Content)：** 配置内容指的是具体的配置数据，如JSON、Properties等格式。这是配置中心存储和管理的具体配置信息。

通过配置这些基础配置项，可以向Nacos注册配置，访问和管理配置数据，并通过命名空间和分组实现不同环境或团队的配置隔离和管理。

```
1 spring:
2   application:
3     name: xxxx
4   cloud:
5     nacos:
6       config:
7         server-addr: localhost:8848
8         file-extension: yml
9         # prefix: 文件名前缀，默认是spring.application.name
10        # 如果没有指定命令空间，则默认命令空间为PUBLIC
11        namespace: dev
12        # 如果没有配置Group，则默认值为DEFAULT_GROUP
13        group: DEFAULT_GROUP
14        # 从Nacos读取配置项的超时时间
15        timeout: 5000
16        # 长轮训超时时间
17        config-long-poll-timeout: 1000
18        # 重试时间
19        config-retry-time: 100000
20        # 长轮训重试次数
21        max-retry: 3
```

5.5 限流降级部分

5.5.1 什么是限流？

限流（Rate Limiting）是一种用于流量控制（flow control）的策略，其原理是监控应用流量的 QPS 或并发线程数等指标，当达到指定的阈值时对流量进行控制。这样可以在高负载或高并发的情况下，保护服务免受过多请求的影响。限流机制可以确保系统在资源有限的情况下，能够按照设定的阈值来处理请求，避免系统过载或崩溃。从而保障应用的高可用性。



QPS：每秒请求数，即在不断向服务器发送请求的情况下，服务器每秒能够处理的请求数量。

5.5.2 为什么要限流？

限流主要是因为请求量比较大,但是系统资源处理能力不足（类似车辆限号），所以我们进行限流，这样可以更好的：

1. 防止系统崩溃：通过限制系统的并发请求数量，可以避免过多的请求导致系统资源耗尽，进而导致系统崩溃或不可用。
2. 保护关键资源：对于某些关键资源（如数据库、缓存等），通过限流可以有效保护其被过多请求使用，从而避免资源被耗尽导致系统瘫痪。
3. 提高系统稳定性：限流可以帮助系统在高负载情况下保持响应速度的稳定性，避免因突发请求过多而导致系统响应变慢。



限流策略可以根据业务需求和系统负载情况进行灵活配置，以保护系统的稳定性和可用性。

5.5.3 你了解哪些限流组件

有多种限流组件可以用于实现限流策略，以下是一些常见的限流组件：

1. Sentinel：Sentinel是阿里巴巴开源的一款流量控制和熔断降级框架，支持多种限流规则和限流策略，并提供实时监控和动态流控规则管理等功能。
2. Hystrix：Hystrix是Netflix开源的一款容错库，支持熔断、限流、舱壁模式等功能，能够帮助构建弹性和可靠的分布式系统。
3. Nginx：Nginx是一款高性能的Web服务器和反向代理服务器，可以通过配置限流模块实现请求的限流。



以上列举的限流组件只是一部分，还有其他一些开源限流组件可供选择。具体选择哪个组件需要根据业务需求、技术栈和系统规模等因素进行评估和决策。

5.5.4 你了解哪些限流算法？

限流可以限制系统接收的请求数量或请求处理的速率。常见的限流算法包括：


1. 令牌桶算法：该算法基于令牌桶的概念，通过设定一个固定容量的令牌桶，每个请求需要消耗一个令牌才能通过。可以根据实际情况设置令牌生成速率和令牌桶的容量。
2. 漏桶算法：该算法基于漏桶的概念，通过固定的速率从请求放入一个漏桶中，若漏桶已满，则请求被丢弃或延迟处理。可以根据实际情况设置漏桶的容量和漏水速率。
3. 计数器算法：该算法简单粗暴，直接对请求进行计数，当请求数量达到阈值时进行限流。可以根据实际情况设置阈值。
4. 基于时间窗口的限流算法：该算法将请求按时间窗口划分，统计每个时间窗口内的请求数量，若超过阈值则进行限流。可以根据实际情况设置时间窗口的大小和阈值。

这些算法可以根据系统的负载情况，决定是否接受或拒绝请求，以保证系统资源的稳定分配。

5.5.5 基于Sentinel实现限流的过程是怎样的？

Sentinel的限流过程主要包括以下几个步骤：

1. 定义资源：首先，我们需要定义要进行限流的资源，资源可以是一个接口、一个方法等。在Sentinel中，我们可以通过注解或配置文件的方式来定义资源。
2. 配置限流规则：一旦资源定义完成，我们可以为每个资源配置相应的限流规则。Sentinel提供了多种限流规则，例如基于QPS（每秒钟的请求数量）、基于线程数等。我们可以通过注解或配置文件的方式来配置限流规则。
3. 监控流量：Sentinel会收集并统计每个资源的实时流量状况。可以通过Sentinel的监控控制台来监控流量数据，并实时了解每个资源的请求量、响应时间等指标。
4. 判断流量是否超过限制：在实际处理请求之前，Sentinel会根据配置的限流规则判断当前的请求是否超过了限制，即是否达到了限流的阈值。
5. 做出限流控制决策：如果判断流量超过了限制，Sentinel会根据配置的限流策略来做出相应的控制决策。常见的限流策略包括直接拒绝、排队等待、慢启动模式等。可以根据实际需求进行配置。
6. 执行请求处理：如果限流策略允许请求通过，Sentinel将继续将请求转发到相应的资源进行处理。

 通过以上步骤，Sentinel可以实现对请求的限流控制，保护系统免受过载的影响。同时，Sentinel还提供了动态配置和实时监控等功能，能够灵活地调整限流规则，并监控系统的流量状况。这使得Sentinel成为一个强大的流量控制和熔断降级框架，用于构建弹性和可靠的分布式系统。

5.5.6 Sentinel限流的阈值类型有哪些？

流控分为两种统计类型，分别是QPS和并发线程数。这两种统计类型在流控中起到不同的作用。

QPS，全称为Queries Per Second，即每秒请求数。它是指在单位时间内系统可以处理的请求数量。这个指标可以用来衡量系统的处理能力和性能。当系统的QPS达到设定的阈值时，流控会触发并采取相应的限制措施，例如拒绝请求或者延迟处理。

而并发线程数则是指同时处理请求的线程数量。当系统的并发线程数达到设定的阈值时，流控会限制进入系统的请求数量，以防止系统过载。并发线程数的控制可以有效地保护系统的稳定性和可用性。

从功能上来看，QPS主要用来衡量系统的处理能力和性能，而并发线程数则主要用来控制系统的负载和保护系统的稳定性。QPS是一种对系统性能的度量，而并发线程数则是一种对系统负载的控制手段。



总结起来，QPS和并发线程数是流控中两种不同的统计类型。QPS用来衡量系统的处理能力和性能，而并发线程数用来控制系统的负载和保护系统的稳定性。在实际应用中，我们需要根据具体情况来选择合适的统计类型，并进行相应的流控策略设置。

5.5.7 Sentinel出现限流时的异常类型是什么？

BlockException?

5.5.8 Sentinel中默认异常处理器是什么？

DefaultBlockExceptionHandler

5.5.9 Sentinel的流控模式有哪些？

- 1)直接模式：直接对请求url进行限流，也就是接口达到限流条件时，直接限流。
- 2)关联模式：一种霸权主义，要保证我(核心业务)先行。关联的资源达到阈值时，就限流自己。(例如我们可以优先保证创建订单可以成功、查询订单可以被限流。)
- 3)链路模式：对同一个资源的访问有多条链路，然后对指定链路进行限流。（例如对Google的访问）

5.5.10 @SentinelResource注解的作用是什么？

定义限流切入点方法,底层可以基于aop方式对请求链路进行限制（一般应用于链路限流、热点数据限流）。对应的切面（SentinelResourceAspect）。

5.5.11 Sentinel常见的限流效果是什么？

快速失败,warm up（预热方式）,排队等待。

快速失败

默认流量控制方式，当QPS超过任意规则的阈值后，新的请求就会被立即拒绝，拒绝方式为抛出FlowException。

warm up

即预热/冷启动方式。当系统长期处于低水位的情况下，当流量突然增加时，直接把系统拉升到高水位可能瞬间把系统压垮。通过"冷启动"，让通过的流量缓慢增加，在一定时间内逐渐增加到阈值上限，给冷系统一个预热的时间，避免冷系统被压垮。

注意：这一效果只针对QPS流控，并发线程数流控不支持。

预热底层是根据令牌桶算法实现的，源码对应得类在

com.alibaba.csp.sentinel.slots.block.flow.controller.WarmUpController。

算法中有一个冷却因子 `coldFactor`，默认值是3，即请求 QPS 从 `threshold(阈值) / 3` 开始，经预热时长逐渐升至设定的 QPS 阈值。


排队等待

匀速排队方式会严格控制请求通过的间隔时间，也即是让请求以均匀的速度通过，对应的是漏桶算法。源码对应得类：`com.alibaba.csp.sentinel.slots.block.flow.controller.RateLimiterController`。

5.5.12 Sentinel热点数据如何限流？

热点数据通常可以理解为高频访问的数据，-例如文章、视频、图片等

Sentinel 利用 LRU 策略统计最近最常访问的热点参数，然后在 `SentinelResourceAspect` 切面中结合令牌桶算法来进行参数级别的流控。

 注意：热点参数限流只针对QPS。

5.5.13 什么是服务熔断和降级？


熔断和降级是在系统遇到异常情况时进行流量控制和保护的两种策略。

1. 熔断：

熔断是一种流量控制策略，用于保护系统免受故障的影响。当某个服务或接口出现异常或不可用时，可以将该服务切换到一个预设的错误处理逻辑，而不是等待超时或产生其他问题。熔断可以快速响应并防止故障在整个系统内部蔓延。常见的熔断策略有三个状态：关闭状态（允许请求通过）、打开状态（直接拒绝请求）、半开状态（逐渐放行请求进行测试）。

2. 降级：

降级是一种保护系统稳定性的策略，用于在系统遇到高压或异常情况下保持核心功能的可用性。当系统资源紧张或请求量激增时，可以主动关闭部分功能，减少对资源的依赖以保证核心功能的稳定性。降级可以通过调整功能的复杂度或直接返回预先设置的默认值或错误信息来实现。降级的目的是保证关键业务的可用性，避免整个系统的崩溃。

 简而言之，熔断处理的是异常情况，快速切换到备用逻辑保证系统可靠性；而降级处理的是高压或异常流量，主动关闭部分功能以保证核心功能的稳定性。两者都是为了保护系统免受异常或过载的影响，并提高系统的可用性和稳定性。

5.5.14 熔断和降级的区别是什么？

降级和熔断都是在异常情况下进行流量控制和保护的策略，但它们的目的和应用场景略有不同：

- 熔断处理的是异常情况，快速切换到备用逻辑保证系统可靠性，防止故障扩散和资源浪费。
- 降级处理的是高压或异常流量，主动关闭部分功能以保证核心功能的稳定性，以避免整个系统的崩溃。



总之，Sentinel 通过统计和监控系统状态，根据预设的阈值和规则来实现熔断和降级功能。它能够帮助开发者保护系统免受异常请求的影响，提高系统的稳定性和可靠性。

5.5.15 Sentinel实现熔断和降级的原理是怎样的？

Sentinel 是阿里巴巴开源的一款流量控制和系统保护的组件，它可以用于实现熔断和降级等功能。Sentinel 的实现原理如下：

1. 计数统计：Sentinel 通过对请求的响应时间、成功次数、异常次数等进行统计，来评估系统的状态。
2. 状态转换：根据统计的结果，Sentinel 利用有限状态机来进行状态转换。通常有三个状态：关闭状态、打开状态、半开状态。
 - 关闭状态：允许请求通过，系统运行正常。
 - 打开状态：直接拒绝请求，不再执行后续的处理。
 - 半开状态：逐渐放行请求进行测试，如果发生异常则继续保持打开状态，否则切换回关闭状态。
3. 触发条件：当某个接口或服务的响应时间、异常比例等超过预设的阈值时，Sentinel 将触发熔断或降级操作。
4. 策略配置：Sentinel 提供了丰富的配置选项，可以根据实际情况配置熔断和降级的规则。可以设置触发熔断或降级的阈值、时间窗口、熔断时长、恢复时间等参数。
5. 信息反馈：Sentinel 还能在触发熔断或降级时，及时记录日志、生成报警，并提供实时的监控和管理界面，方便运维人员进行查看和调整配置。



总之，Sentinel 通过统计和监控系统的状态，根据预设的阈值和规则来实现熔断和降级功能。它能够帮助开发者保护系统免受异常请求的影响，提高系统的稳定性和可靠性。

5.6 API网关部分

5.6.1 什么是API网关？

网关（Gateway）是一种在系统架构中用于承担请求分发、协议转换和路由转发等功能的中间层组件。它作为系统的入口，负责接收来自客户端的请求，并将请求转发给后端的服务。

5.6.2 为什么需要API网关？

使用API网关的好处和原因如下：

1. 统一入口：API网关提供了对外的统一入口，客户端只需与API网关进行通信，而不需要直接与后端的多个微服务进行交互。这简化了客户端的请求配置和管理，并提供了一个单一的入口点。
2. 请求路由与负载均衡：API网关可以根据路由规则将请求转发到正确的后端微服务。这样可以根据请求的不同特征（如路径、主机名、HTTP方法等）将请求合理地分发给相应的微服务实例，并实现负载均衡，在后端微服务之间平均分配负载。
3. 安全认证和授权：API网关可以处理请求的安全问题，包括身份认证和授权。它可以验证请求的凭据（如令牌、API密钥等），并根据事先定义的策略进行访问控制，以保护后端微服务免受未经授权的访问。
4. 降低复杂性：通过使用API网关，可以将各个微服务的内部结构和实现细节隐藏起来，对外部客户端提供一致的API接口。这降低了客户端和后端微服务之间的耦合性，提高了系统的可维护性和可扩展性。
5. 缓存和数据聚合：API网关可以缓存来自后端微服务的响应结果，并在下次相同请求时直接返回缓存的响应。这提高了系统的性能和响应速度。另外，网关还可以从多个微服务中聚合数据，形成更完整的响应结果。
6. 监控和指标收集：API网关可以收集和记录请求的各项指标和性能数据。这有助于系统的监控和故障排查，帮助开发团队了解系统的运行状态和性能情况。



综上所述，使用API网关可以提供统一的入口、请求路由与负载均衡、安全认证和授权、降低复杂性、缓存和数据聚合、监控和指标收集等多项好处。它为微服务架构提供了一种中心化的管理和控制机制，促进了系统的可扩展性、安全性和性能。

5.6.3 API网关主要功能？

API网关可以实现的主要功能包括：

1. 请求路由：网关可以根据请求的路由规则将请求转发给不同的后端服务。通过配置路由规则，可以实现请求的动态转发，达到灵活的请求分发和负载均衡的效果。
2. 协议转换：网关可以将不同的协议转换为后端服务所支持的协议。例如，将 HTTP 请求转换为 RPC 请求，或者将 WebSocket 协议转换为 HTTP 协议等。
3. 安全控制：网关可以集中处理身份认证、权限控制和访问控制等安全相关的功能。通过在网关层进行统一的安全控制，可以简化后端服务的安全处理逻辑。
4. 缓存加速：网关可以充当缓存层，将经常请求的数据缓存在网关上，以加速对后端服务的访问。这样可以减少后端服务的负载，提高系统的性能和响应速度。
5. 限流熔断：网关可以根据请求的流量情况进行限流熔断，保护后端服务免受恶意请求或高并发压力的影响。通过限制请求并拒绝超出阈值的流量，可以保持系统的稳定性和可用性。



总之，网关在系统架构中承担着请求分发、协议转换、安全控制、缓存加速等功能，为系统提供了统一入口和基础服务支持，提高了系统的可扩展性、安全性和性能。

5.6.4 API网关都可以配置什么？

Spring Cloud Gateway是一个基于Spring Framework的轻量级API网关，提供了丰富的配置选项来满足不同需求。以下是Spring Cloud Gateway主要可以配置的内容：

1. 路由配置：可以通过配置路由规则来定义API的路由，包括请求路径、请求方法、断言条件等。可以通过配置多个路由规则来支持不同的请求转发。
2. 过滤器配置：可以通过配置过滤器来对请求进行全局或特定的处理。过滤器可以用于请求转发前的预处理（如修改请求头、添加认证信息等）、请求转发后的后处理（如修改响应内容、添加响应头等）等。
3. 限流和熔断配置：可以通过配置限流和熔断规则来保护后端微服务免受过载的影响。可以设置并发请求限制、QPS限制、错误率阈值等，并定义失败时的处理策略，如返回默认响应或进行降级处理。
4. 路径重写配置：可以通过配置路径重写规则来修改请求的路径。可以将请求路径的一部分或全部替换为新的路径，实现路径的重定向和重写。

5. 响应重写配置：可以通过配置响应重写规则来对后端微服务返回的响应进行修改。可以修改响应头、响应体内容等，以适应特定的需求。
6. 安全认证和授权配置：可以配置网关进行身份认证和授权的相关规则，如验证请求的凭据、生成和解析访问令牌、进行访问控制等。
7. 监控和指标配置：可以配置网关进行监控和指标收集，记录请求的各项指标和性能数据，支持与监控系统（如Prometheus、Elasticsearch等）集成。

上述配置项可以通过Java代码或配置文件的方式进行配置，并结合Spring Cloud Gateway的自动配置机制，实现功能的定制和扩展。可以根据具体的需求和场景来选择配置内容，以满足系统的要求。

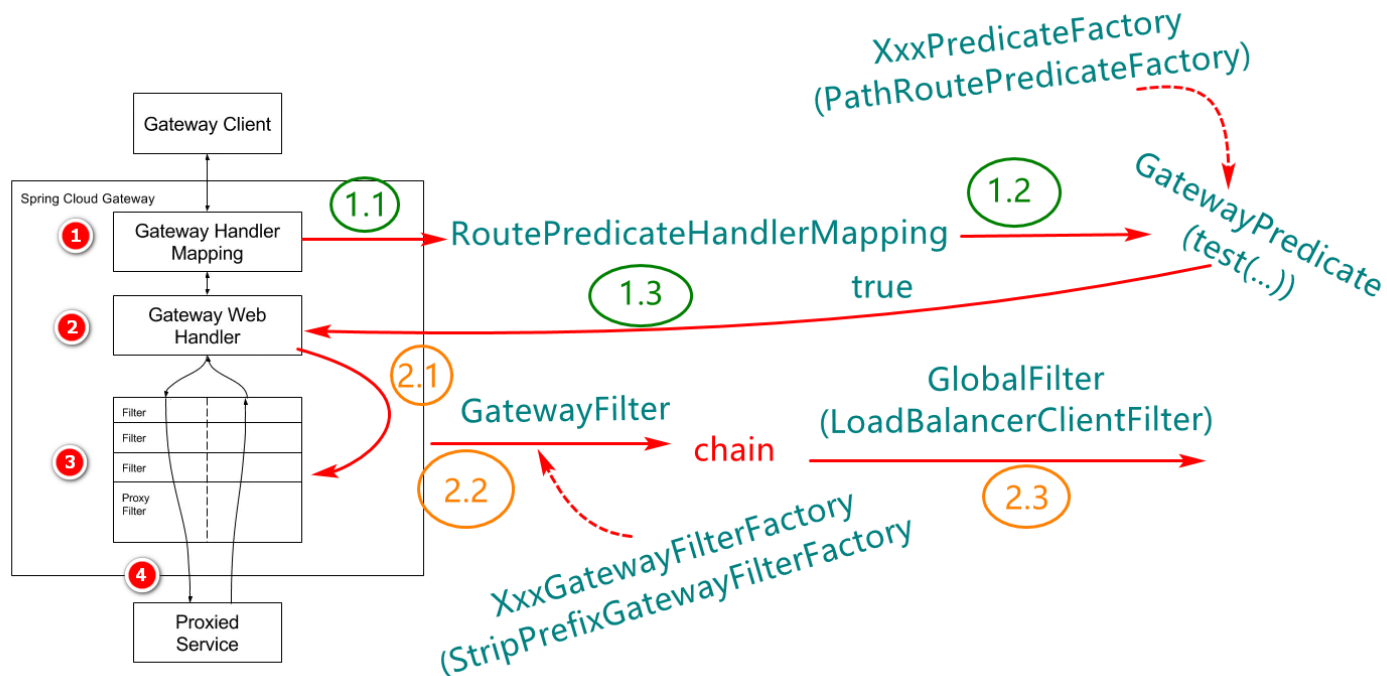
5.6.5 API网关中的负载均衡如何实现？

在Spring Cloud Gateway中，负载均衡的实现主要通过集成Spring Cloud LoadBalancer来完成。Spring Cloud LoadBalancer是Spring Cloud提供的负载均衡框架，可以与多种负载均衡器（如Ribbon、Nacos、Consul等）集成，实现请求的负载均衡。

```
1  spring:
2    cloud:
3      gateway:
4        routes:
5          - id: service-route
6            uri: lb://service-name
7            predicates:
8              - Path=/api/**
```

5.6.6 SpringCloud Gateway处理请求的基本流程？

- 1)客户端向Spring Cloud Gateway发出请求。
- 2)基于GatewayHandlerMapping调用谓词predicates(predicates)的集合判定请求与路由(Routers)是否匹配，不匹配则抛出异常。
- 3)将请求其发送到Gateway Web Handler。此对象基于路由配置调用过滤链中的过滤器（也就是所谓的责任链模式）进一步的处理请求。
- 4)将请求转发到具体的服务。



5.6.7 网关中谓词(Predicate)的作用是什么？

对请求url、请求数据进行校验，符合规则再交给过滤器去处理。

5.6.8 你知道哪些谓词对象(Predicate)对象？

1. Path相关的
2. 日期时间相关的
3. IP相关的
4. Cookie相关的
5. 请求参数相关的
6. 请求方式相关
7. 请求头相关的
8. 上传文件大小相关的
9. ...

5.6.9 网关中的过滤器是如何分类的？

- 1) 全局过滤器(不需要配置)，作用于所有路由。
- 2) 局部过滤器（这个需要针对具体路由进行配置），只作用于具体路由。

5.6.10 你知道Gateway中的哪些过滤器？

1. 负载均衡相关的
2. 请求转发相关的
3. 限流相关的
4. 请求前缀、后缀相关的
5. ...

5.6.11 Gateway如何基于Sentinel进行限流？

在Spring Cloud Gateway中，可以集成阿里巴巴的Sentinel来实现请求的限流功能。Sentinel是一个强大的流量控制和熔断器组件，可以对服务进行细粒度的流量控制、熔断降级和系统保护。

要在Spring Cloud Gateway中基于Sentinel实现限流，可以按照以下步骤进行：

1. 引入Sentinel依赖：在项目的pom.xml文件中添加spring-cloud-starter-alibaba-sentinel依赖。
2. 配置Sentinel资源规则：在Sentinel的配置文件中，指定需要进行限流的资源和限制条件。可以配置URL、方法、参数等作为资源，并设置相应的限流阈值和策略。
3. 配置Sentinel资源过滤器：在Spring Cloud Gateway的配置中，通过添加Sentinel的资源过滤器来实现限流。可以指定需要进行限流的资源名称和对应的Sentinel配置文件。

以下是一个使用Sentinel实现限流的Spring Cloud Gateway配置示例：

```
1  spring:
2    cloud:
3      gateway:
4        discovery:
5          locator:
6            enabled: true
7            lower-case-service-id: true
8        routes:
9          - id: sentinel-route
10            uri: lb://service-name
11            predicates:
12              - Path=/api/**
13            filters:
14              - name: Sentinel
15                args:
16                  resourceName: my-resource
17                  blockHandler: fallbackHandler
18                  blockExceptionHandler: com.example.MyBlockHandlerClass
```

在上述配置中，通过filters配置项指定了使用Sentinel的资源过滤器，并配置了资源名称为my-resource。此外，还可以配置限流的降级处理方法，以及自定义的熔断降级处理类。

需要注意的是，使用Sentinel进行限流时，还需要启动Sentinel的控制台来管理资源规则和实时监控。通过控制台，可以动态地调整限流规则和查看限流情况。

使用Sentinel实现限流可以帮助保护后端服务免受异常流量的影响，提高服务的可用性和稳定性。

5.7 分布式事务

5.7.1 什么是分布式事务？

分布式事务是分布式架构下的一种事务处理方式，是指位于不同的节点之上的事务参与者，执行一系列操作时，要确保这些操作要么都执行成功，要么都执行失败。以保证数据的一致性和完整性。

传统的单机事务（例如在关系型数据库中使用的ACID事务）可以通过数据库的事务管理机制来实现，但在分布式环境中，涉及到多个独立的服务或数据库实例，无法使用单一的事务管理机制来完成分布式事务的管理。因此，需要使用特殊的技术和方法来实现分布式事务。

5.7.2 CAP定理及对分布式事务的影响？

在分布式系统中，有三个指标分别是：

- 1)一致性(Consistency)是指在分布式系统中的多个副本或节点之间，无论用户如何进行读取操作或更新操作，最终都能保证数据的一致性。即系统要么返回最新的数据（强一致性），要么返回过去一段时间内的最新数据（最终一致性）。
- 2)可用性(Availability)：是指在分布式系统中，系统能够持续地保持对外提供服务，即使在某些节点或副本出现故障的情况下也能够继续正常运行。
- 3)分区容错性(Partition tolerance)是指在分布式系统中，系统能够正常工作并保持一致性或可用性，即使在网络中存在分区通讯中断的情况下。这也是分布式架构系统中必须要满足的一个指标,在分布式架构下的一台服务器出了问题,其它服务必须依旧可以持续提供业务服务。

这三个指标不可能同时满足，这个定理就叫CAP定理。在分布式系统中，通常分区容错性是必须满足的，可用性(AP)和一致性(CP)只能选择其一。

因此，在设计和实现分布式系统并进行分布式事务处理时，需要根据具体的业务需求和系统的特点，权衡在一致性、可用性和分区容错性之间的取舍。常见的做法是在系统中进行适当的折中，根据业务

的重要性和数据的一致性要求，选择合适的分布式架构和技术方案。

5.7.3 解释一下什么是BASE理论？

BASE理论是对分布式系统设计原则的一个概括。BASE是指：基本可用（Basically Available）、软状态（Soft state）和最终一致性（Eventually Consistent）。

基本可用性（Basically Available）是指在分布式系统中，尽管系统可能会遇到故障或部分节点不可用的情况，但系统仍然能够保持基本的可用性，即能够继续处理和响应用户的请求。

软状态（Soft state）是指在分布式系统中的节点之间，并不要求数据的强一致性，允许存在一段时间的数据不一致，即允许数据的状态是“软”的。这样的设计思路可以提高系统的可用性和性能。

最终一致性（Eventually Consistent）是指在分布式系统中，虽然数据会存在一定的时间窗口内的一致性，但最终数据会达到一致的状态。系统会通过一定的机制，例如异步复制和定时同步等，保证数据的最终一致性。

BASE理论相对于ACID（原子性、一致性、隔离性和持久性）事务模型，提供了一种更加宽松和灵活的设计思路，可以满足分布式系统中的高可用性和性能要求。在BASE理论中，系统设计者可以根据对系统的要求和业务场景，合理地权衡一致性和性能，选择适合的一致性模型。

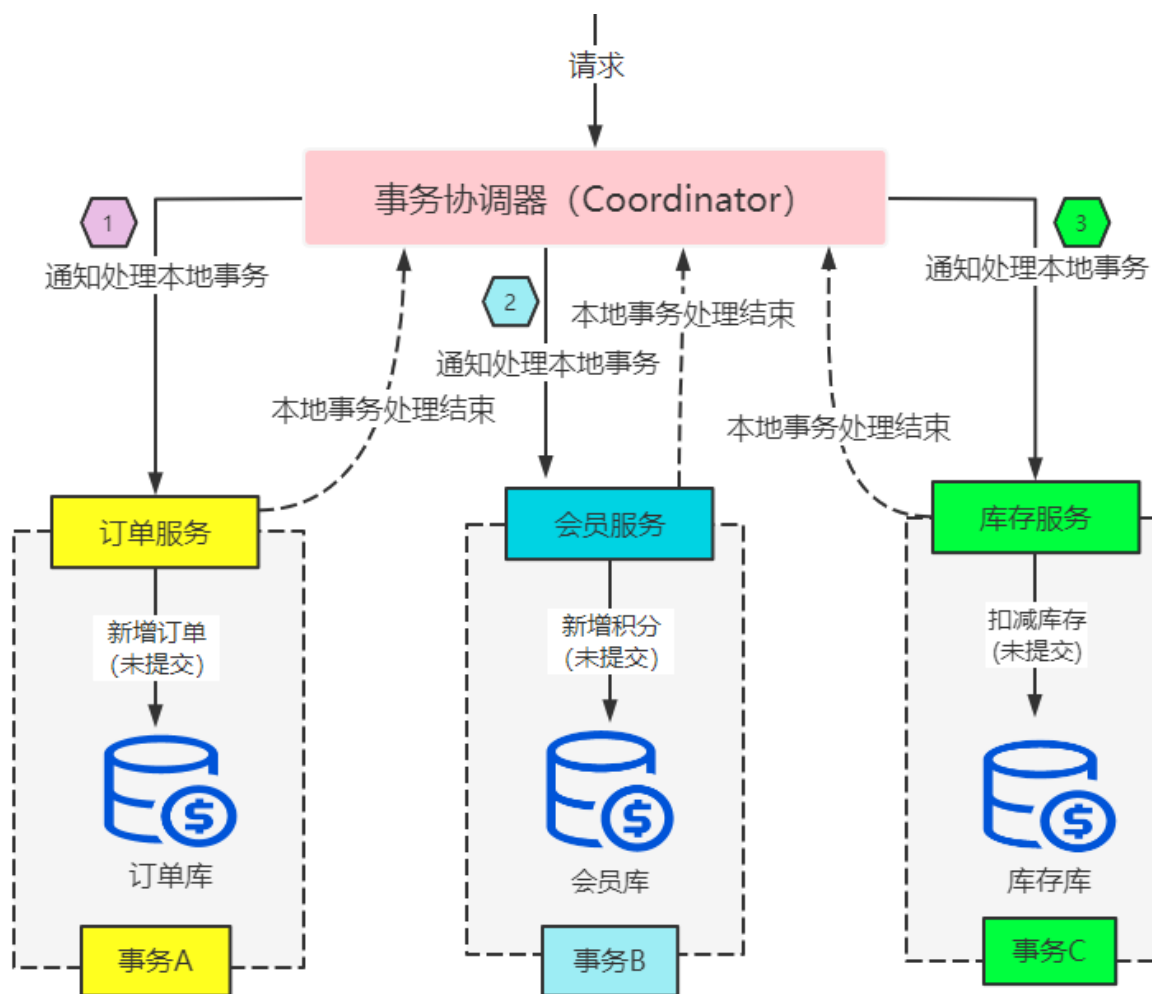


需要注意的是，BASE理论并不是说一致性不重要，而是在实际的分布式系统中，根据业务需求和系统特点，通过一系列的机制和策略来实现最终一致性。这样既可以保证系统的可用性，又可以满足用户对数据一致性的要求。

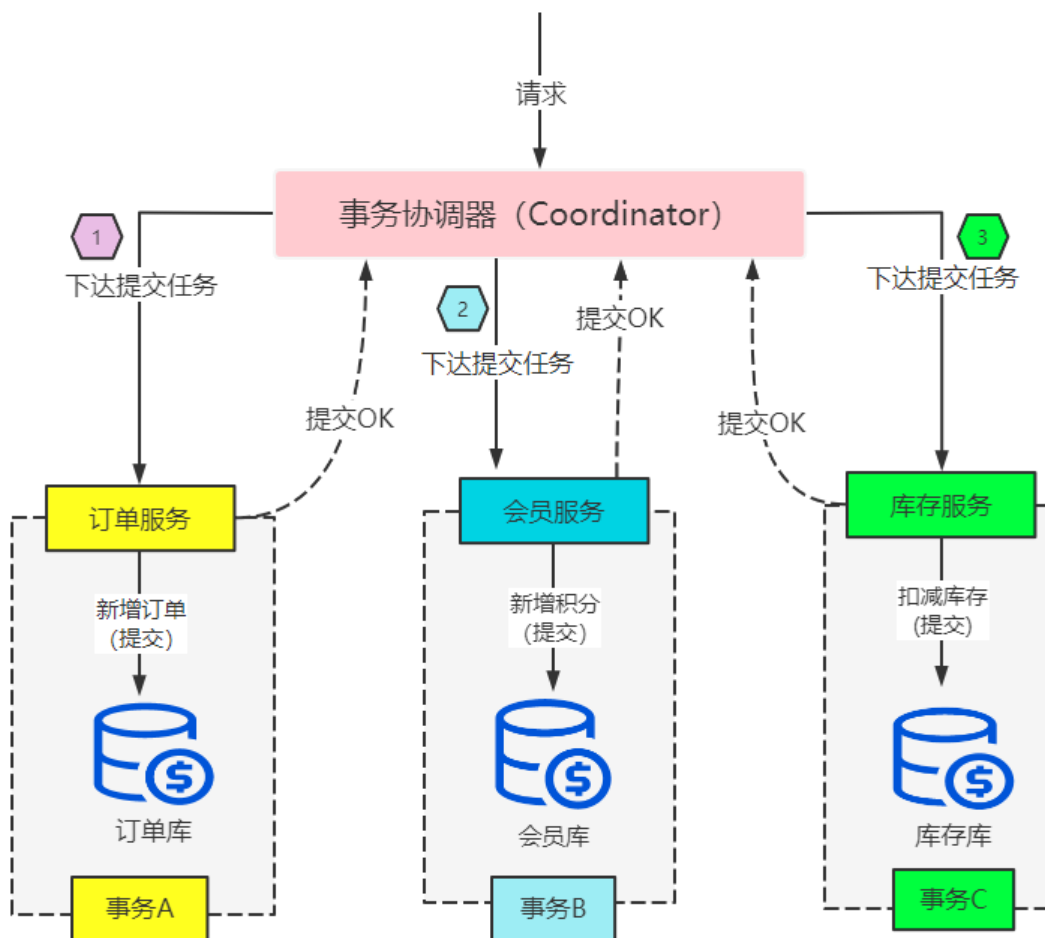
5.7.4 解释一下两阶段提交协议？

两阶段提交是一种分布式事务处理的协议。它包含两个阶段：准备阶段和提交阶段。在准备阶段，协调者节点向所有参与者节点发送要执行的事务，并询问它们是否准备好提交事务。参与者节点会执行事务并在准备好的情况下向协调者发送“准备就绪”消息。在提交阶段，协调者（Coordinator）节点根据参与者节点的反馈来决定是提交还是回滚事务。

当商城应用订单创建时，首先事务协调者会向各服务下达“处理本地事务”的通知，所谓本地事务就是每个服务应该做的事情，如订单服务中负责创建新的订单记录；会员服务负责增加会员的积分；库存服务负责减少库存数量。在这个阶段，被操作的所有数据都处于未提交（uncommit）的状态，会被排它锁锁定。这个阶段也是二阶段提交中的第一阶段（也称之为预处理阶段），如图所示：



当本地事务都处理完成后，会通知事务协调者“本地事务处理完毕”。当事务协调者陆续收到订单、会员、库存服务的处理完毕通知后，便进入“阶段二：提交阶段”。

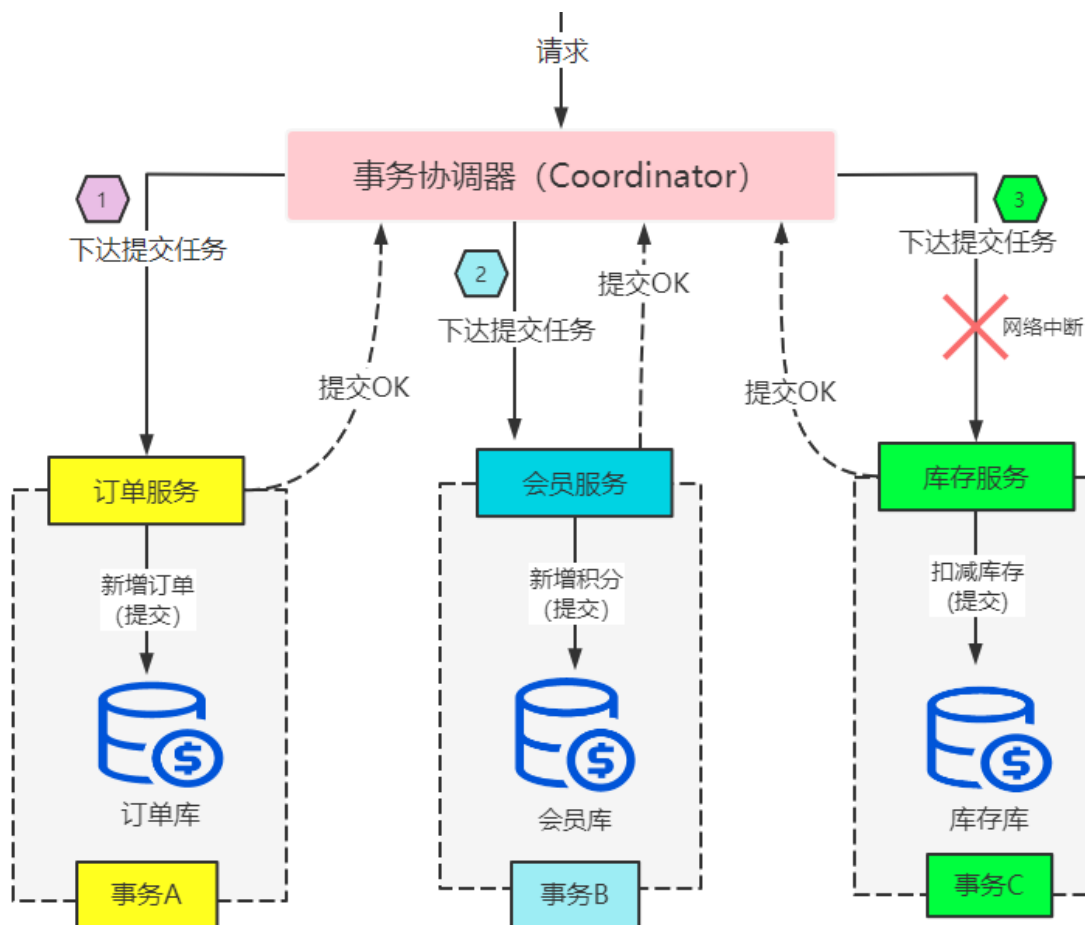


在提交阶段，事务协调者会向每一个服务下达提交命令，每个服务收到提交命令后在本地事务中对阶段一未提交的数据执行 Commit 提交以完成数据最终的写入，之后服务便向事务协调者上报“提交成功”的通知。当事务协调者收到所有服务“提交成功”的通知后，就意味着一次分布式事务处理已完成。

这便是二阶段提交的正常执行过程，但假设在阶段一有任何一个服务因某种原因向事务协调者上报“事务处理失败”，就意味着整体业务处理出现问题，阶段二的操作就自动改为回滚（Rollback）处理，将所有未提交的数据撤销，使数据还原以保证完整性。

5.7.5 分布式事务二阶段提交有什么缺陷？

两阶段提交协议存在阻塞问题，即如果任何一个参与者节点或协调者节点宕机或网络中断，整个协议会处于阻塞状态。此外，两阶段提交协议在故障恢复和扩展方面也存在一些复杂性和性能开销。



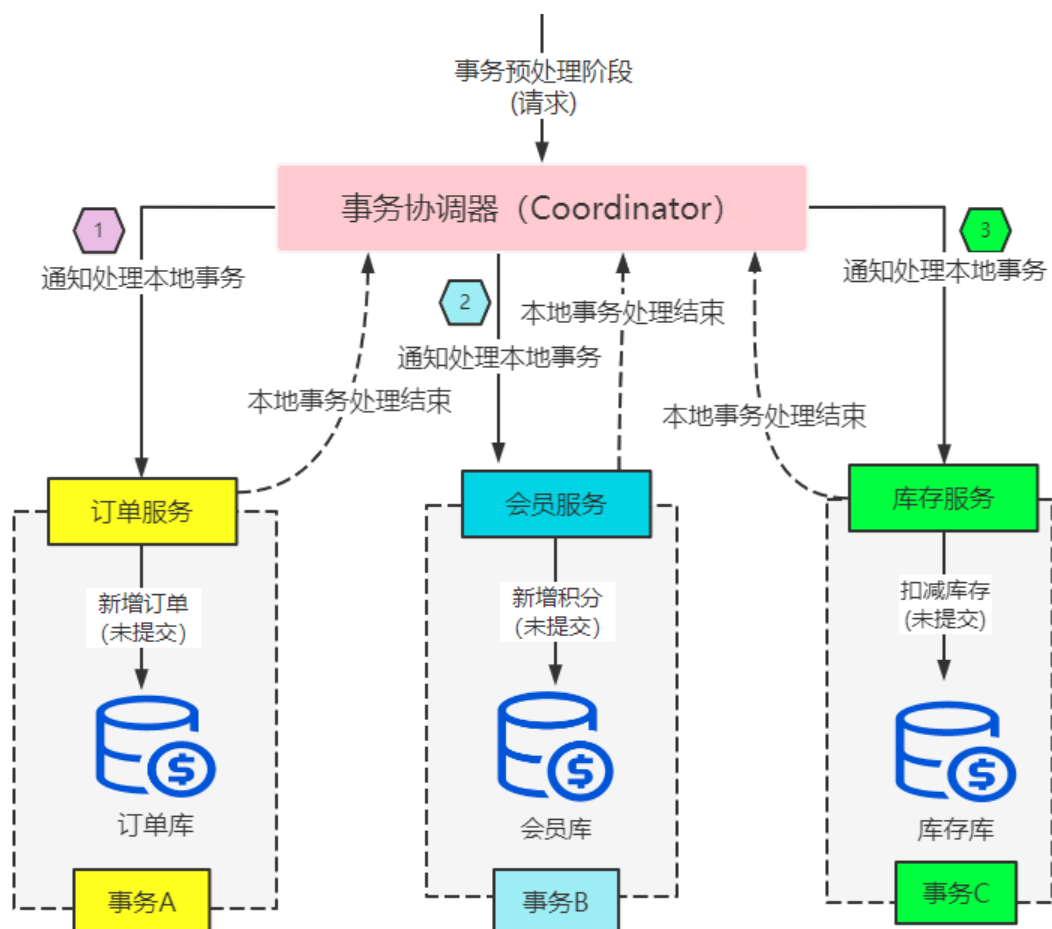
以上图为例，假如在提交阶段，库存服务实例与事务协调者之间断网。提交指令无法下达，这会导致商品库存记录会长期处于未提交的状态，因为这条记录被数据库排他锁长期独占，之后再有其他线程要访问“飞科剃须刀”库存数据，该线程就会长期处于阻塞状态，随着阻塞线程的不断增加，库存服务会面临崩溃的风险。

那这个问题要怎么解决呢？其实只要在服务这一侧增加超时机制，过一段时间被锁定的“飞科剃须刀”数据因超时自动执行提交操作，释放锁定资源。尽管这样做会导致数据不一致，但也比线程积压导致服务崩溃要好，出于此目的，三阶段提交（3PC）便应运而生。

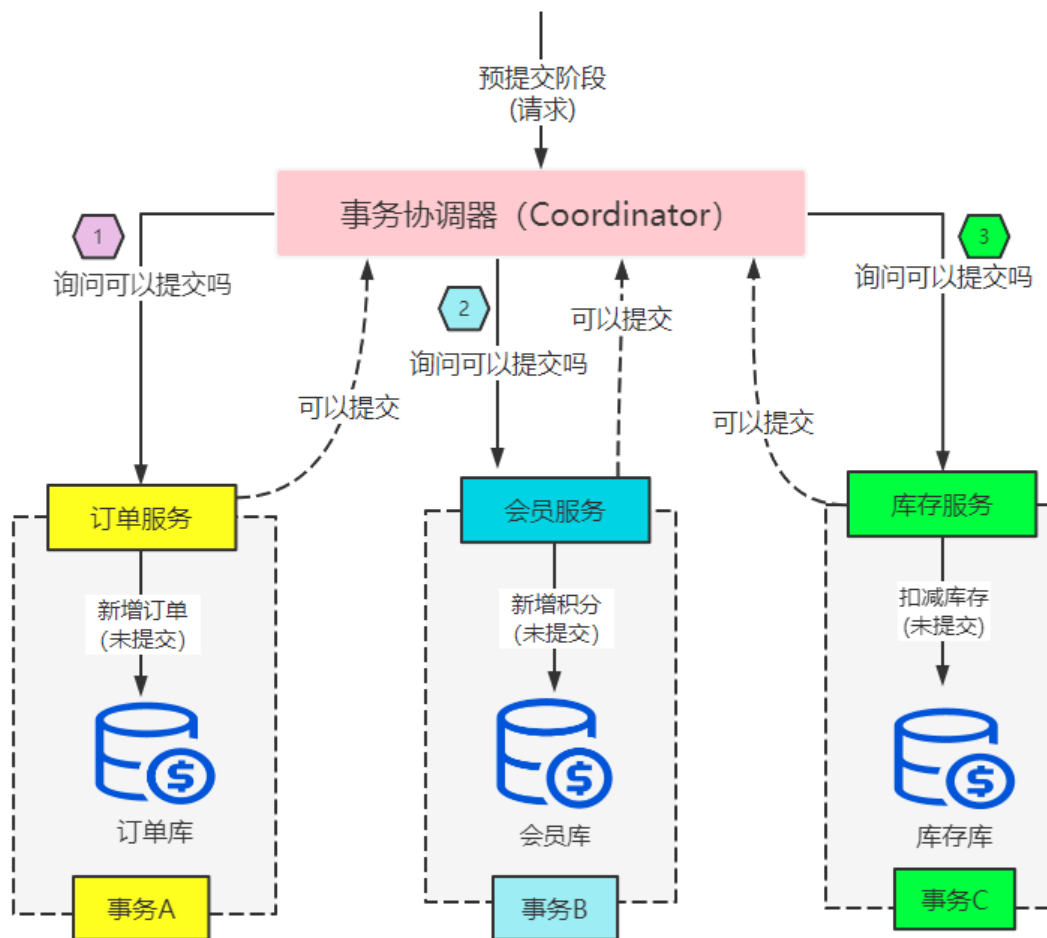
5.7.6 分布式事务中的三阶段提交？

三阶段提交实质是将二阶段中的提交阶段拆分为“预提交阶段”与“提交阶段”，同时在服务端都引入超时机制，保证数据库资源不会被长时间锁定。下面是三阶段提交的示意流程：

阶段1：事务预处理阶段，3PC 的事务预处理阶段与 2PC 是一样的，用于处理本地事务，锁定数据库资源，例如：



阶段2:当所有服务返回成功后，进入阶段二。预提交阶段只是一个询问机制，以确认所有服务都已准备好，同时在此阶段协调者和参与者都设置了超时时间以防止出现长时间资源锁定。



当阶段二所有服务返回“可以提交”，进入阶段三“提交阶段”。

3PC 的提交阶段与 2PC 的提交阶段是一致的，在每一个数据库中执行提交实现数据的资源写入，如果协调者与服务通信中断导致无法提交，在服务端超时后在也会自动执行提交操作来保证资源释放。

三阶段提交本质上是二阶段提交的优化版本，主要通过加入预提交阶段引入了超时机制，让数据库资源不会被长期锁定，但这也会带来一个新问题，数据一致性也很可能因为超时后的强制提交被破坏，对于这个问题各大软件公司都在各显神通，常见的做法有：增加异步的数据补偿任务、更完善的业务数据完整性的校验代码、引入数据监控及时通知人工补录这些都是不错的补救措施。

5.7.7 解释一下柔性事务 (Saga) 模式？

柔性事务是一种通过将事务拆分为多个小的原子操作，并通过补偿机制来保证最终一致性的分布式事务模式。每个原子操作都有自己的提交和回滚逻辑，当一个原子操作发生失败时，系统可以通过执行相应的补偿操作来回滚事务。

5.7.8 解释一下Seata是什么？

Seata是一款开源的分布式事务解决方案，提供高性能和高可靠性的分布式事务支持。它能够帮助应用程序在分布式环境中实现原子性、一致性、隔离性和持久性（ACID）的事务行为。

它的官网是<http://seata.io/>。

5.7.9 Seata的主要组件有哪些？

Seata主要由三大组件组成：

- Transaction Coordinator（TC）：协调和管理分布式事务的全局事务状态。
- Transaction Manager（TM）：控制分布式事务的边界和提交/回滚请求的发起方。
- Resource Manager（RM）：管理本地资源，并与TC和TM进行通信以提交/回滚本地事务。

5.7.10 Seata的工作流程是怎样的？

1. 应用程序通过 Seata 提供的 API 开启全局事务。
2. Seata 事务协调器(TC)生成全局事务 ID，并将该 ID 分发给所有参与者。
3. 在分布式系统的各个节点上，事务管理器(TM)协调并管理分支事务的**生命周期**，包括事务的开始、提交和回滚。
4. 每个分支事务对应一个资源管理器(RM)，负责管理本地资源的事务操作。资源管理器(RM)与事务管理器(TM)进行通信，保证分支事务的一致性。
5. 全局事务执行结束时，应用程序提交全局事务。Seata 事务协调器(TC)根据全局事务ID，通知所有参与者提交分支事务。
6. 如果全局事务执行过程中出现异常，应用程序可以通过 Seata 提供的 API 发起全局事务的回滚操作。Seata 事务协调器(TC)根据全局事务ID，通知所有参与者回滚分支事务。

5.7.11 解释一下Seata中的两阶段提交？

Seata中的两阶段提交机制与传统的两阶段提交（2PC）协议类似。在第一阶段，TM向所有RM发送事务提交请求，并等待RM的响应。在第二阶段，TM根据所有RM的响应决定是否提交或回滚事务。

5.7.12 Seata是如何保证事务的一致性和可靠性？

Seata通过事务日志存储和强一致性策略来实现事务的一致性和可靠性。它使用日志记录所有的事务操作，并将事务日志存储在可靠的存储介质上。当发生故障或机器宕机时，Seata可以通过恢复之前的事务日志来保证事务的一致性。

5.7.13 Seata支持哪些分布式事务模式？

Seata支持AT（自动补偿型）、TCC（尝试/确认/取消型）、SAGA、XA四种分布式事务模式。

AT模式中业务操作在本地事务中进行。Seata通过事务协调器（Transaction Coordinator）记录事务日志，并在需要时回滚事务。当发生故障或异常时，Seata会根据事务日志中的补偿逻辑来执行事务的回滚操作，以确保数据的一致性。此模式是Seata主推的分布式事务解决方案，对业务无侵入，真正做到业务与事务分离

TCC（Try-Confirm-Cancel）模式，适用于有明确操作语义的场景。在TCC模式中，业务操作被拆分为三个阶段：尝试（Try）、确认（Confirm）和取消（Cancel）。Seata通过执行这三个阶段中的相应操作来实现事务的一致性。如果任何一个阶段失败，Seata会执行相应的取消操作来回滚事务。此模式对业务代码侵入性太强。没有AT模式全局锁，加锁逻辑需要根据业务自行实现。

SAGA模式中的业务操作是通过发送和接收消息来间接执行的。Seata使用消息队列作为基础设施，将业务操作封装成消息，并在事务提交前将其发送到消息队列。一旦提交事务，消息被消费，业务操作得以执行。如果发生故障，Seata会通过消息重试或者死信队列来修复和恢复事务。此模式的正向服务和补偿服务都需要手动实现，因此有很强的侵入性。能保证隔离性，不容易进行并发控制。

XA模式是一种广泛使用的分布式事务协议。在XA模式中，Seata通过和分布式资源管理器（ResourceManager）进行协调，保证所有资源的一致性。Seata使用XA协议来实现全局事务控制。此模式传统分布式强一致性的解决方案，性能较低，在实际业务中使用较少。

5.8 消息队列应用

5.8.1 解释一下消息队列？

消息队列简称MQ（Message Queue），通常被认为消息中间件，可以理解为一个使用队列来通信的组件。它的本质，就是个转发器，包含发消息、存消息、消费消息的过程。

5.8.2 消息队列的应用场景？

消息队列在分布式系统中具有广泛的使用场景，包括但不限于以下几个方面：

1. 异步通信：消息队列常用于异步通信场景，发送者将消息发送到队列中，而不需要等待接收者的即时响应。这对于解耦和提高系统的吞吐量非常有帮助。例如，一个订单系统可以将订单创建的消息发送到消息队列，而不需要等待库存系统立即处理。
2. 事件驱动架构：消息队列可以作为实现事件驱动架构的关键组件。不同的服务可以通过发布和订阅消息来实现解耦以及事件的通知和处理。当一个事件发生时，相关的服务可以接收到相应的消息并采取相应的操作。

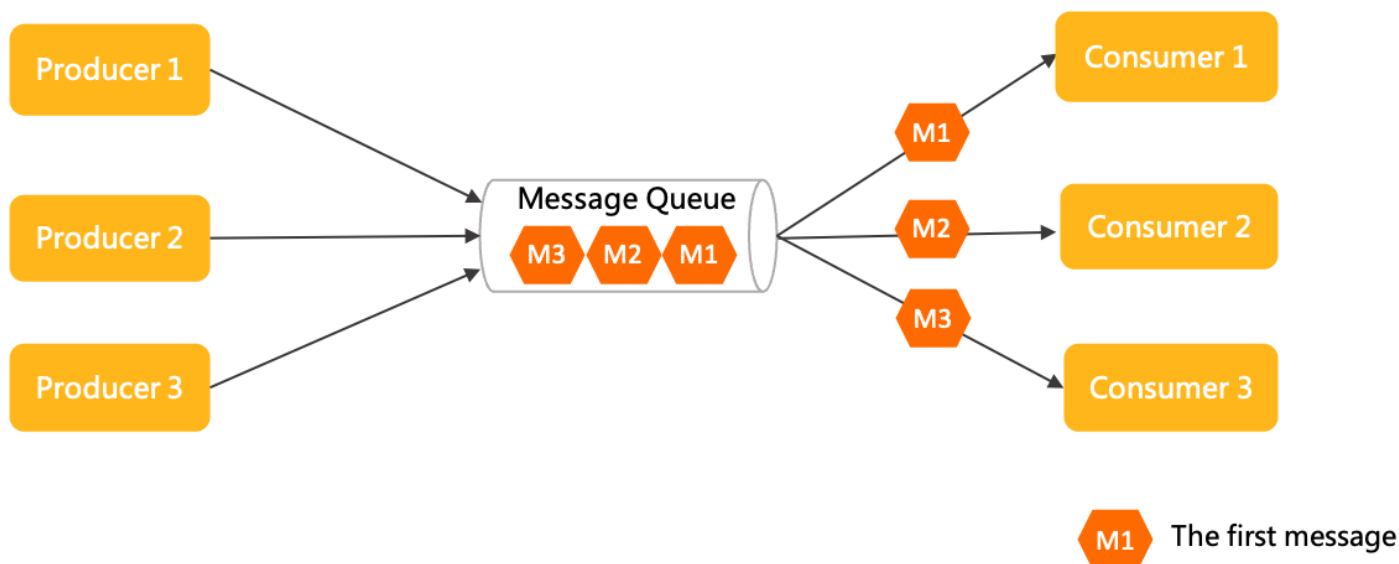
3. 流量削峰和负载均衡：通过消息队列，可以实现流量的削峰和负载均衡。当系统面临峰值流量时，可以将消息缓冲到队列中，然后按照系统的处理能力逐步处理。这可以防止系统超载和行为不一致。
4. 分布式事务以及最终一致性：消息队列可以在分布式事务中起到关键的作用，例如，通过消息队列实现的消息补偿机制可以实现最终一致性。在某些复杂的业务场景中，使用消息队列来跨系统和服务管理事务状态可以大大简化系统的复杂性。
5. 日志收集和数据分析：消息队列可以用于集中收集和传递日志信息，这对于系统监控和故障排查非常重要。通过将日志信息发送到消息队列，可以实现实时的日志处理和数据分析。

综上所述，消息队列具有广泛的使用场景，适用于许多与分布式系统相关的问题和需求。

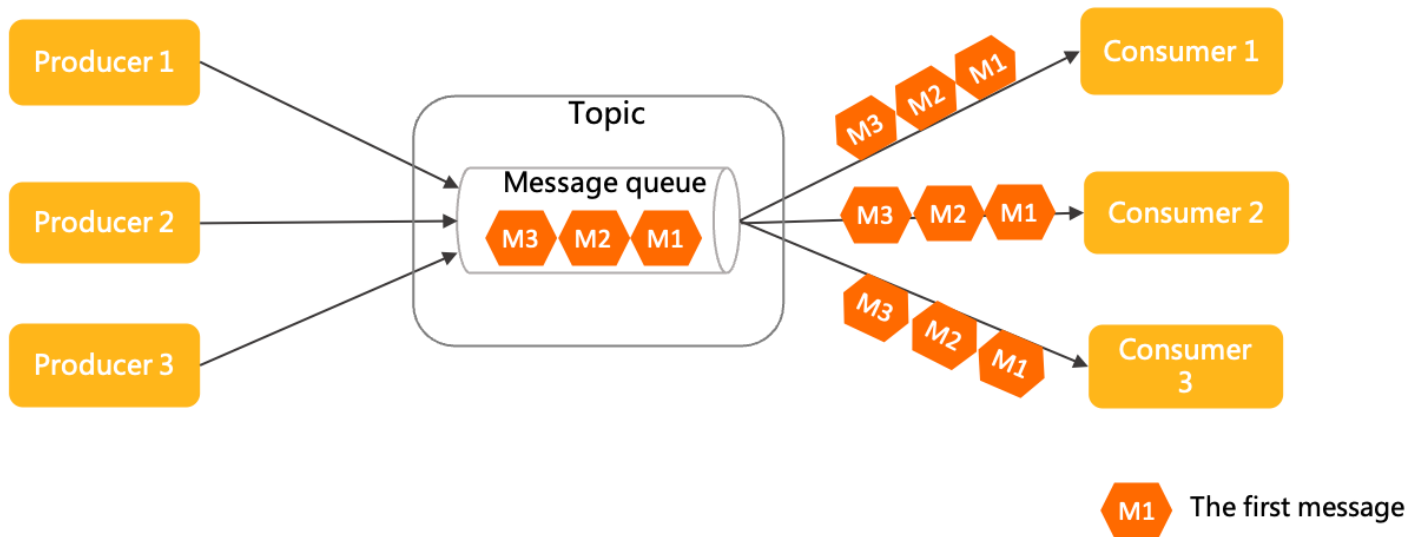
5.8.3 说说你对消息传输模型的理解？

主流的消息中间件的传输模型主要为点对点模型和发布订阅模型。

点对点模型(每一条消息都只会被唯一一个消费者处理。因此点对点模型只能实现一对一通信)，如图所示：



发布订阅模型(同一个主题内的消息可以被多个订阅组处理，可以实现一对多通信。)，如图所示：

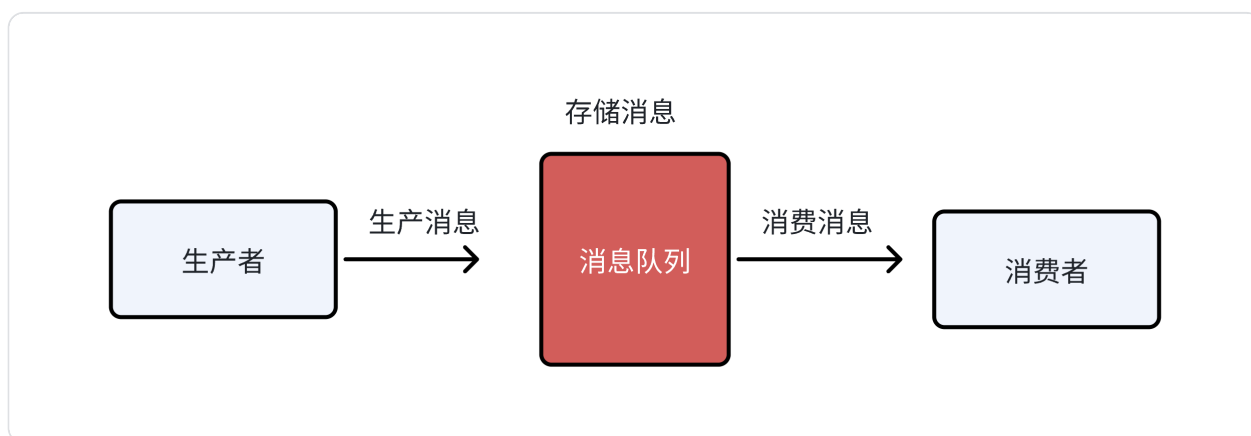


5.8.4 RocketMQ的基本架构是怎样的？

RocketMQ的架构主要包括消息生产者（Producer）、消息消费者（Consumer）、消息队列（Broker）和命名服务（Name Server）。Producer负责发送消息，Consumer负责接收和处理消息，Broker存储和转发消息，Name Server提供命名服务和路由管理。

5.8.5 消息队列如何保证消息可靠性？

一个消息从生产、存储、消费的基本过程，如图所示：



所以，假如保证消息的可靠性，只要保证生产者不丢消息，存储端不丢消息，消费端不丢消息即可。

5.8.6 RocketMQ如何保证生产端不丢消息？

如果是RocketMQ消息中间件，生产者要想发消息时保证消息不丢失，采用**同步方式**发送消息，send消息方法返回**成功**状态，就表示消息正常到达了存储端Broker。如果send消息**异常**或者返回**非成功**状态，可以**重试**。也可以使用事务消息，RocketMQ的事务消息机制就是为了保证零丢失来设计的。

5.8.7 RocketMQ如何保证队列端不丢消息？

保证存储端的消息不丢失需要确保消息**持久化**到磁盘，RocketMQ刷盘可分**同步刷盘**和**异步刷盘**：

- 生产者消息发过来时，只有持久化到磁盘，RocketMQ的存储端Broker才返回一个成功的ACK响应，这就是**同步刷盘**。它保证消息不丢失，但是影响了性能。
- 异步刷盘的话，只要消息写入PageCache缓存，就返回一个成功的ACK响应。这样提高了MQ的性能，但是如果这时候机器断电了，就会丢失消息。

Broker一般是**集群部署**的，有master主节点和slave从节点。消息到Broker存储端，只有主节点和从节点都写入成功，才反馈成功的ack给生产者。这就是**同步复制**，它保证了消息不丢失，但是降低了系统的吞吐量。与之对应的就是**异步复制**，只要消息写入主节点成功，就返回成功的ack，它速度快，但是会有性能问题。

5.8.8 RocketMQ消费者如何保证消息不丢？

RocketMQ会为每个消费者实例记录消费位移，即消费者已经消费到的消息位置。消费者在获取消息后会将当前消费的位置提交到Broker，然后由Broker记录消费位移。当消费者实例重启或者新的消费者实例加入时，会根据消费位移从上次消费的位置继续消费消息，保证消息的不丢失。

5.8.9 RocketMQ如何保证消费的顺序？

消息的有序性，就是指可以按照消息的发送顺序来消费。有些业务对消息的顺序是有要求的，比如**先下单再付款，最后再完成订单**等。RocketMQ通过以下机制保证消息的消费顺序：

1. 消息队列的有序存储：RocketMQ中每个Topic包含多个消息队列，每个消息队列有一个消费者线程负责消费。消息队列中的消息按照顺序存储，保证了消息按照发送的顺序存储在队列中。
2. 每个消费者线程按照消息队列顺序消费：RocketMQ使用Pull模式拉取消息，每个消费者线程独立拉取消息，并按照消息队列的顺序进行消费。这样可以确保每个消费者线程按照顺序消费所拉取的消息。
3. 单线程消费：在同一个消费组中，RocketMQ只会将消息发送到同一个消费者线程中进行消费。这保证了消息可以按照顺序由同一个消费者线程消费，避免了多个消费者之间可能产生的顺序混乱的情况。




需要注意的是，消息的顺序消费是相对于消息队列而言的，即同一个消息队列上的消息是有顺序的。如果在业务层面需要保证跨队列的消息消费顺序，可以将相应的消息发送到同一个队列中。同时，消费者在处理消息时，也需要保证业务逻辑的顺序性。

5.8.10 如何避免消息被重复消费？

消息队列出现消息的重复消费，可能是：

- 生产端为了保证消息的可靠性，它可能往MQ服务器重复发送消息，直到拿到成功的ACK。
- 消费端消费消息一般是这个流程：**拉取消息、业务逻辑处理、提交消费位移**。假设业务逻辑处理完，事务提交了，但是需要更新消费位移时，消费者却挂了，这时候另一个消费者就会拉到重复消息了。

 避免消息的重复消费，可以从**幂等处理重复消息**说起，简单来说，就是搞个本地表，带**唯一业务标记**的，利用主键或者唯一性索引，每次处理业务，先校验一下就好啦。又或者用redis缓存下业务标记，每次看下是否处理过了。

5.8.11 如何处理消息积压问题？

消息积压是因为生产者的生产速度，大于消费者的消费速度。遇到消息积压问题时，我们需要先排查，是不是有bug产生了。如果不是bug，我们可以**优化一下消费的逻辑**，比如之前是一条一条消息消费处理的话，我们可以确认是不是可以优为**批量处理消息**。如果还是慢，我们可以考虑水平扩容，增加Topic的队列数和消费组机器的数量，提升整体消费能力。

对于RocketMQ的消息积压问题，可以从以下几个方面进行处理：

1. 消费者线程的数量：增加消费者线程的数量可以提高消息的消费速度，减少消息积压。可以根据生产者的消息发送速率和消费者的消费能力来调整消费者线程的数量。
2. 增加消费者实例：如果消费者线程无法满足消费需求，可以增加消费者实例，将消息分摊到多个消费者实例中进行消费。
3. 调整消息拉取的间隔和批量大小：通过调整消息拉取的间隔和批量大小，可以控制消费者每次拉取消息的数量，以适应消费者的消费能力。
4. 并发消费：对于可以并发处理的消息，可以使用多线程并发消费的方式提高消费速度，减少积压。可以将相互没有依赖关系的消息分配给不同的消费者线程来并发处理。
5. 调整消费者组和消费者偏移量：通过调整消费者组和消费者的偏移量，可以重新消费未被消费的消息，以减少积压。
6. 监控和告警：使用RocketMQ提供的监控工具和告警机制，可以实时监控消费者消费的情况，及时发现消息积压的问题，并进行相应的处理。

需要根据具体的业务情况和系统负载来选择合适的方法来处理消息积压问题。同时，还可以根据实际情况来调整RocketMQ的配置参数，如消息存储策略、消息堆积阈值等，以满足业务需求。

5.8.12 Kafka/RocketMQ/RabbitMQ的异同点

当涉及Kafka、RocketMQ和RabbitMQ这几个消息队列时，它们都有各自的特点和适用场景，下面是它们的异同点：

1. Kafka：

- Kafka 是一个高吞吐量、低延迟的分布式消息系统，主要用于大规模数据处理和高并发场景。
- Kafka的设计目标是支持高容量的实时数据流，具备持久性和良好的可伸缩性。
- 提供了持久性存储、高可用性、分布式部署和数据复制等特性，支持海量数据的持久化存储和离线处理。
- Kafka的消费模型以分区为基础，采用消费者拉取消息的方式，具备较低的延迟和高并发能力。
- 适用于大数据处理、日志收集、事件驱动架构以及流处理等场景。

2. RocketMQ：

- RocketMQ 是一个开源的分布式消息队列系统，最初由阿里巴巴集团开发并开源。
- RocketMQ非常擅长处理消息的有序性，能够保证消息严格按照发送顺序进行处理。
- 具备较高的可用性、容错性和可伸缩性，支持主备切换和水平扩展。
- RocketMQ提供了多种消息模式（点对点和发布/订阅）、可靠消息传递机制、顺序消息支持和广播等功能。
- 适用于金融、电商、物流以及其他对消息顺序和高可用性有特殊要求的场景。

3. RabbitMQ：

- RabbitMQ 是一个开源的消息队列系统，实现了高级消息队列协议（AMQP）标准。
- RabbitMQ提供了灵活的消息路由和交换机机制，支持多种消息模式和交换机类型。
- 具备较高的可靠性和稳定性，支持持久化存储和可靠消息传递机制。
- RabbitMQ的社区非常活跃，提供了多种语言的客户端SDK，易于开发和使用。
- RabbitMQ适用于各种场景，包括任务调度、RPC、发布/订阅模式等，并且具备良好的可扩展性。



在选择时，可以根据具体业务需求和场景选择最适合的消息队列。同时，也可以考虑结合使用多个消息队列来满足不同的需求。

5.9 鲁班上门项目

5.9.1 师傅入住的必要流程是怎样的？

1. 前台注册
2. 前台登录
3. 前台上传图片
4. 前台入住提交
5. 后台审核列表
6. 后台审核详情
7. 后台审核提交
8. 前台师傅详情

5.9.2 师傅抢单的必要流程是怎样的？

1. 查询需求单列表
2. 抢单提交
3. 查询订单列表
4. 查询到订单详情
5. 签到
6. 施工图片上传
7. 确定施工图片绑定
8. 师傅完成订单
9. 异步结算模拟打款
10. 异步修改订单完成

5.9.3 如何保证一个订单只有一名师傅抢单成功？

基于CAS乐观锁,在修改需求单为抢单之前,先查询字段version的值,更新时携带这个值,值相同修改成功,抢单成功,version值不同,修改失败,抢单失败。

5.9.4 师傅入驻的幂等性是如何设计的？

师傅入驻可能重复提交,当第一次提交审核驳回后,会需要再次提交,所以需要设计成幂等。在入驻新增之前,使用userId删除已有师傅信息,实现幂等。

5.9.5 师傅上传的图片但没有入住怎么处理？

师傅没入住，假如图片绑定没有完成 则图片服务器会定时删除超时未绑定数据。

图片上传和 数据与图片做业务绑定有时间间隔. 导致上传完图片没有做数据绑定,图片数据是冗余的.也可以根据创建图片时间和绑定空数据为条件，定时刷表格(buzType=0 buzId=0 create_time 超过30分钟 1小时 2小时)。