【编程导航】后端经典面试题合集

> 分类汇总 90 道经典后端面试题目 + 题解
>
> 作者：编程导航知识星球 + 星球鱼友们

Java 基础

1、JDK 和 JRE 和 JVM 分别是什么，有什么区别?

官方解析

1. JDK（Java Development Kit）：JDK 是 **Java 开发工具包**，包含了编写、编译、调试和运行 Java 程序所需的所有工具和组件，比如编译器（javac）、Java API、调试工具等。JDK 是针对 Java 开发人员的，**它包含了 JRE**，还有编译器和其他工具，可以用来编写和调试 Java 程序。
2. JRE（Java Runtime Environment）：JRE 是 **Java 运行时环境，包括了 Java 虚拟机（JVM）和 Java 标准类库（Java API）**。JRE 是针对 Java 应用程序的，它**提供了在计算机上运行 Java 应用程序所需的最小环境**。
3. JVM（Java Virtual Machine）：JVM 是 **Java 虚拟机**，是 Java 程序运行的环境。JVM 负责将 Java 代码解释或编译为本地机器代码，并在运行时提供必要的环境支持，比如内存管理、垃圾回收、安全性等。JVM 的主要作用是**将 Java 代码转换为可以在计算机上运行的机器码，并负责程序的执行**。

综上所述，JDK、JRE 和 JVM 之间的区别可以总结如下：

JDK 是 Java 开发工具包，包括了编译器、Java API、调试工具等，用于开发 Java 应用程序。

- **JRE 是 Java 运行时环境**，包括了 Java 虚拟机和 Java 标准类库，用于在计算机上运行 Java 应用程序。
- **JVM 是 Java 虚拟机**，是 Java 程序运行的环境，负责将 Java 代码转换为可以在计算机上运行的机器码，并提供必要的环境支持。

鱼友的精彩回答

萧芝雉随的回答

Java 虚拟机（JVM）是运行 Java 字节码的虚拟机。JVM 有针对不同系统的特定实现（Windows，Linux，macOS），目的是使用相同的字节码，它们都会给出相同的结果。字节码和不同系统的 JVM 实现是 Java 语言“一次编译，到处运行”的关键所在。

JVM 并不是只有一种！只要满足 JVM 规范，每个公司、组织或者个人都可以开发自己的专属 JVM。

JDK 是 Java Development Kit 缩写，它是功能齐全的 Java SDK。它拥有 JRE 所拥有的一切，还有编译器（javac）和工具（如 javadoc 和 jdb）。它能够创建和编译程序。

JRE 是 Java 运行时环境。它是运行已编译 Java 程序所需的所有内容的集合，包括 Java 虚拟机（JVM），Java 类库，java 命令和其他的一些基础构件。但是，它不能用于创建新程序。

如果你只是为了运行一下 Java 程序的话，那么你只需要安装 JRE 就可以了。如果你需要进行一些 Java 编程方面的工作，那么你就需要安装 JDK 了。

L深渊绘卷的回答

搞清楚这个问题，我觉得首先你要清楚 Java 程序是如何在计算机上运行的。

1.编写Java源代码；--------需要用到JDK,**Java Development KitJava** (Java开发工具包)

2.将Java源代码编译成字节码；---------需要用到JAVA开发工具包中的编译器（javac.exe），生产.class文件

3.Java虚拟机（JVM）将字节码转换成机器码；-------需要用到 **Java Virtual Machine** (Java虚拟机)

4.操作系统执行机器码，完成程序的运行。---------需要用到Java Runtime Environment (Java 运行环境)

JDK 全称 **Java Development KitJava** (Java开发工具包)，就是包含了开发Java 程序时所需的文件包。

JRE 全称 **Java Runtime Environment** (Java 运行环境)，就是运行Java 程序时所需的文件包。

JVM 全称 **Java Virtual Machine** (Java虚拟机)，是可运行Java代码的假想计算机。在Java虚拟机上生成可运行的字节码，移植到在其他平台上时可不加修改地运行。

总结：

Java开发工具包=Java 运行环境+Java开发工具

Java 运行环境=Java虚拟机+Java核心类库

一切总会归于平淡的补充

JDK + JRE + JVM 的 概念我看到有人发了，我就发一下大家可能在各个阶段对JDK + JRE + JVM的理解吧。

- 如果你是初级 Java 开发工程师可能会觉得 JDK、JRE 和 JVM 都是 Java 的一部分，但对它们的具体区别可能理解不够深入。他们可能只知道需要在本地安装 JDK 或 JRE 才能运行 Java 程序，并且需要使用 JVM 来解释执行 Java 代码。
- 如果你是中级开发工程师应该能够区分 JDK、JRE 和 JVM 的功能，并知道它们之间的关系，这对于开发 Java 应用程序以及排查相关问题都是非常重要的。此外，中级开发工程师还应该能够使用 JDK 提供的工具来诊断和解决问题，例如 jstack、jmap 和 jstat 等命令行工具。
- 如果你已经成为高级 Java 开发工程师，那你对 JDK、JRE 和 JVM 的区别应该非常清楚，并且能够更深入地理解它们的作用。首先，你了解到 JDK 是 Java 开发工具包，包含JRE和开发工具，如编译器、调试器、文档生成器等。它为开发者提供了创建 Java 应用程序所需的一切工具和环境。
    - 其次，你知道 JRE 是 Java 运行时环境，它包含了 JVM 和 Java 类库，能够在计算机上运行 Java 程序，但没有开发工具。它为用户提供了运行 Java 程序所需的环境，但不包括开发 Java 程序的工具。
    - 最后，你了解到 JVM 是 Java 虚拟机，是 Java 程序运行的基础。JVM 可以在各种平台上运行 Java 程序，并提供了 Java 程序运行所需的环境。JVM 能够将 Java 字节码翻译成本地机器代码，使得 Java 程序可以在不同的操作系统和硬件平台上运行。
    - 除此之外，你还能够深入了解 JDK、JRE 和 JVM 的内部结构和工作原理，例如类加载机制、垃圾回收机制、字节码优化等。你也会使用一些高级工具和技术来优化 Java 应用程序的性能和稳定性，例如 JIT 编译器、性能分析工具、线程池、断路器等。

2、什么是字节码？采用字节码的最大好处是什么？

官方解析

字节码是 Java 程序编译后的中间代码，是一种可移植的二进制代码，可以在任何支持 Java 虚拟机（JVM）的平台上运行。字节码通过将 Java 源代码编译为字节码指令序列，使得 Java 程序可以跨平台运行，即使是在不同的操作系统和硬件平台上也可以运行。

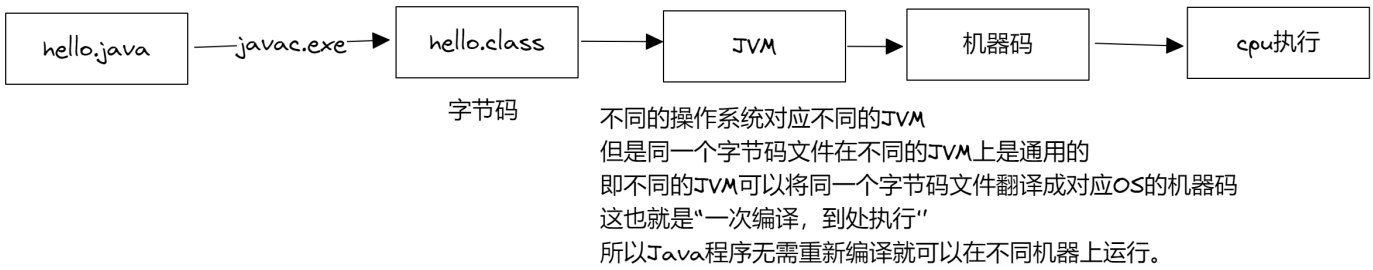
字节码采用中间代码的形式，相比于直接将程序编译为特定平台上的机器码，有以下几个好处：

1. **可移植性**：由于字节码是中间代码，所以可以在任何支持 JVM 的平台上运行，使得 Java 程序具有很好的可移植性。这也是 Java 跨平台的重要特性之一。
2. **安全性**：由于字节码需要在 JVM 中运行，所以可以对字节码进行安全检查，以确保程序不会对系统造成威胁。
3. **性能**：由于字节码是一种紧凑的二进制格式，相比于直接编译为机器码，可以更快地加载和传输，同时也可以在运行时进行动态优化，提高程序的执行效率。
4. **可读性**：相比于直接编译为机器码，字节码具有更好的可读性，可以方便地进行反汇编和调试。

因此，采用字节码作为中间代码的最大好处是提高了 Java 程序的可移植性、安全性、性能和可读性。这也是 Java 跨平台和安全性等特性的基础。

鱼友的精彩回答

行云的回答



3、Java 和 C++、Go 语言的区别，各自的优缺点？

官方解析

Java、C++ 和 Go 都是流行的编程语言，它们各自有着自己的特点和优缺点。

Java 是一种面向对象的编程语言，最初被设计用于开发嵌入式系统，现在已广泛应用于企业应用、Web 应用和移动应用开发等领域。Java 的优点包括：

- **平台性强(跨平台)**：Java 虚拟机（JVM）可以在不同的操作系统上运行 Java 程序，使得开发者可以编写一次代码并在多个平台上运行；
- **安全性高**：Java 有很多安全特性，例如自动内存管理、类的访问权限控制、异常处理等，可以有效避免一些常见的安全漏洞；

- **生态系统完备**：Java 的生态系统非常完善，有很多成熟的框架和工具可以帮助开发者快速构建复杂的应用。C++ 是一种系统级编程语言，最初被设计用于操作系统和系统软件的开发，现在也广泛应用于游戏、图形图像处理、嵌入式系统等领域。C++ 的优点包括：
- **性能高**：C++ 是一种编译型语言，可以生成高效的机器代码，所以在处理大量数据和对计算效率有要求的应用场景下具有优势；
- **控制能力强**：C++ 具有非常高的控制能力，可以直接操作计算机的硬件和内存等资源；
- **应用领域广泛**：C++ 适用于开发大型系统、高性能应用和底层软件，如操作系统、数据库、游戏引擎等。Go 是一种开源的编程语言，由 Google 公司开发，被设计用于开发高并发、分布式系统。Go 的优点包括：
- **并发能力强**：Go 具有轻量级的线程（goroutine）和基于消息传递的通道（channel），可以方便地实现并发编程；
- **内存管理高效**：Go 的内存管理采用自动垃圾回收机制，使得开发者可以更加专注于业务逻辑；
- **语言特性简单**：Go 的语法简单易懂，代码易于阅读和维护，同时提供了丰富的标准库和工具支持。

不同的编程语言适用于不同的应用场景，开发者可以根据项目的需求、自己的经验和偏好选择合适的语言。

鱼友的精彩回答

一切总会归于平淡的回答

JAVA

- 优点
  - 简单易学，代码可读性强
  - 跨平台，一次编写可以在多个操作系统上运行
  - 面向对象，支持继承、多态等特性
  - 丰富的类库，可以快速开发应用程序
  - 自动内存管理，减少了内存泄漏的可能性
- 缺点:
  - 由于JVM的存在，运行速度相对较慢
  - 对于实时性要求较高的场景，Java的表现可能不如C++和Go

C++

- 优点：
  - 速度快，适合编写需要高性能的应用程序
  - 应用广泛，特别是在游戏开发、操作系统和嵌入式系统开发方面
  - 灵活性高，可以直接访问硬件和内存
- 缺点：
  - 学习难度较高，需要掌握指针、内存管理等底层知识
  - 容易出现内存泄漏和指针错误等问题
  - 编写代码过程中需要更多的手动管理，相比 Java 更容易出错

Go:

- 优点：
  - 高并发，天生支持协程，能够轻松编写高效的并发程序
  - 简单易学，语法简洁，上手容易
  - 静态类型语言，可以避免一些潜在的运行时错误
  - 快速编译，可以快速构建和部署应用程序
- 缺点：
  - 缺乏丰富的类库，与 Java 和 C++ 相比有些不足
  - 在一些性能要求极高的场景中可能不如 C++ 表现
  - 语言本身还比较年轻，相关生态和工具还需要进一步完善
    使用场景：

JAVA

- 适合开发企业级应用程序、后端服务等。

C++

- 适合开发需要高性能和高可靠性的应用程序，特别是在游戏开发、操作系统和嵌入式系统开发方面。

GO

- 适合开发高并发的后端服务、微服务、容器化应用程序等.

当然，每种语言都有其独特的优势和适用场景，具体应根据项目需求和开发团队的技术背景来选择合适的语言。

如果当面试官问你这个问题时，他们可能想要了解你对不同编程语言的了解和理解程度，以及你是否能够根据不同的项目需求和特点选择合适的编程语言。

此外，面试官可能还想知道你是否有跨语言的经验和能力，以及你是否能够评估和解决跨语言集成和兼容性问题。

最后，通过你的回答，面试官还可以了解你对软件开发生态系统的了解程度，以及你是否能够在行业发展和趋势方面保持敏锐的洞察力。

4、JDK 动态代理和 CGLIB 动态代理的区别是什么?

官方解析

JDK 动态代理和 CGLIB 动态代理都是 Java 中动态代理的两种实现方式，它们的区别主要在以下几个方面：

- 实现方式：JDK 动态代理是通过反射实现的，而CGLIB动态代理是通过继承目标类来实现的。
- 目标类限制：JDK 动态代理要求目标类必须要实现接口，而CGLIB动态代理则没有这个限制。
- 性能：JDK 动态代理相对于 CGLIB 动态代理来说，因为实现方式不同，生成的代理类的效率会低一些。
- 对象类型：JDK 动态代理只能代理实现了接口的类，CGLIB 通过继承实现，不能代理 final 类。
- 依赖库：JDK 动态代理是 Java 自带的库，不需要额外的依赖，而 CGLIB 动态代理需要依赖 cglib 库。

在使用动态代理时，可以根据需要和具体的场景选择合适的实现方式，JDK 动态代理适用于**接口代理**的场景，而 CGLIB 动态代理适用于**类代理**的场景。

鱼友的精彩回答

解州扯面的回答

JDK和CGLib动态代理区别

1、JDK动态代理具体实现原理：

- 通过实现 InvocationHandler 接口创建自己的调用处理器；
- 通过为 Proxy 类指定 ClassLoader 对象和一组 interface 来创建动态代理；
- 通过反射机制获取动态代理类的构造函数，其唯一参数类型就是调用处理器接口类型；
- 通过构造函数创建动态代理类实例，构造时调用处理器对象作为参数参入；

JDK 动态代理是面向接口的代理模式，如果被代理目标没有接口那么 Spring 也无能为力，Spring 通过 Java 的反射机制生产被代理接口的新的匿名实现类，重写了其中 AOP 的增强方法。

2、CGLib动态代理：

利用 ASM 开源包，对代理对象类的 class 文件加载进来，通过修改其字节码生成子类来处理。

3、两者对比：

JDK 动态代理是面向接口的。

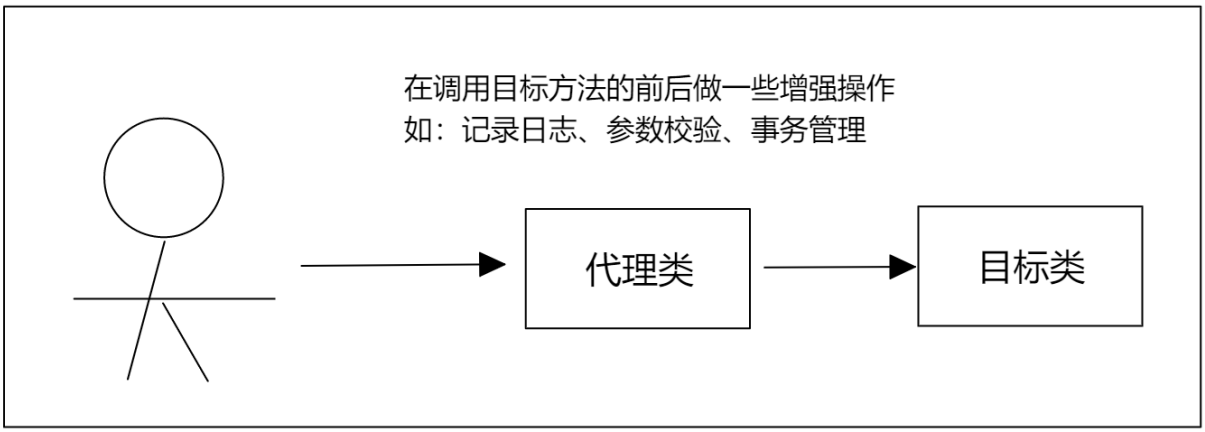
CGLib 动态代理是通过字节码底层继承要代理类来实现，因此如果被代理类被 final 关键字所修饰，会失败。

如果要被代理的对象是个实现类，那么 Spring 会使用 JDK 动态代理来完成操作（Spirng 默认采用 JDK 动态代理实现机制）；

如果要被代理的对象不是个实现类，那么 Spring 会强制使用 CGLib 来实现动态代理。

行云的补充

代理模式



代理模式的实现方式：

静态代理：手动编写代理类代码

JDK动态代理：运行时通过反射自动生成代理对象

实现：

1、动态代理类实现InvocationHandler接口，通过重写invoke()实现增强操作

2、通过getInstance()传入目标类，返回值即为代理类

要求：目标类必须要有接口

CGLIB动态代理：即Code Generation Library,是一个开源的第三方工具库，

其原理是继承，去生成目标类的子类对象，这样对目标类的功能进行增强。

但是要求：目标类不能用final修饰，目标类中的方法也不能被final修饰。

5、Java 中 final 关键字有什么用?

官方解析

在 Java 中，final 关键字用于表示一个不可变的常量或一个不可变的变量。

在 Java 中，final 关键字可以修饰类、方法和变量，作用如下：

1. final 修饰类，表示该类不能被继承。final 类中的方法默认都是 final 的，不能被子类重写。
2. final 修饰方法，表示该方法不能被子类重写。
3. final 修饰变量，表示该变量只能被赋值一次。final 修饰的变量必须在声明时或构造函数中初始化，且不能再被修改。常用于定义常量。

另外，使用 final 修饰的变量在编译时就已经确定了其值，因此在运行时访问时比非 final 变量更快。

使用 final 关键字可以带来一些好处，例如：

1. 安全性：将变量声明为 final 可以防止它被改变，从而提高安全性。
2. 可读性：将常量声明为 final 可以提高代码的可读性，因为常量的值不会被修改。
3. 优化：final 变量在编译时被转换成常量，这可以提高程序的性能。

总之，final 关键字在 Java 中非常有用，可以提高程序的安全性、可读性和性能。

鱼友的精彩回答

迷。的回答

1. 定义常量：在变量声明前加上 final 关键字，可以将变量定义为常量。常量是不可变的，一旦初始化后其值不能再被改变。例如，final int MAX_NUM = 100; 将 MAX_NUM 定义为一个常量，其值不能再被改变。
2. 防止方法被重写：在方法声明前加上 final 关键字，可以防止该方法被子类重写。例如，final void doSomething() { ... } 将 doSomething 方法定义为一个不可重写的方法。
3. 防止类被继承：在类声明前加上 final 关键字，可以防止该类被继承。例如，final class MyClass { ... } 将 MyClass 类定义为一个不可继承的类。
4. 确保对象引用不可变：在对象引用声明前加上 final 关键字，可以确保该引用指向的对象不能被改变，但是该对象的内容可以被改变。例如，final MyClass obj = new MyClass(); 将 obj 声明为一个不可变引用，指向一个可变的 MyClass 对象。

需要注意的是，final 关键字不同于 static 关键字，final 关键字用于定义常量或限制重写或继承等操作，而 static 关键字用于定义静态变量或方法，表示这些成员属于类，而不是属于对象。

黑的回答

1、**final可以用来修饰的结构：类、方法、变量**

2、**final用来修饰一个类：此类不能被其它类继承。**

当我们需要让一个类永远不被继承，此时就可以用final修饰，但要注意：final类中所有的成员方法都会隐式的定义为final方法。

比如：String类、System类、StringBuffer类

3、**final 用来修饰方法： 表明此方法不可以被重写**

作用：

(1) 把方法锁定，以防止继承类对其进行更改。

(2) 效率，在早期的java版本中，会将final方法转为内嵌调用。但若方法过于庞大，可能在性能上不会有多大提升。因此在最近版本中，不需要final方法进行这些优化了。

final方法意味着“最后的、最终的”含义，即此方法不能被重写。
比如：Object类中的getClass( )

4、**final 用来修饰变量 ，此时变量就相当于常量**

- final用来修饰属性：可以考虑赋值的位置有：显式初始化、代码块中初始化、构造器中初始化
- final修饰局部变量：尤其是使用final修饰形参时，表明此形参是一个常量。当我们调用此方法时，给常量形参赋一个实参，一旦赋值之后，就只能在方法体内使用此形参的值，不能重新进行赋值。

- 如果final修饰一个引用类型时，则在对其初始化之后便不能再让其指向其他对象了或者说他的地址不能发生变化了（因为引用的值是一个地址，final要求值，即地址的值不发生变化），但该引用所指向的对象的内容是可以发生变化的。本质上是一回事。

5、**使用 final 关键字声明类、变量和方法需要注意以下几点：**

- final 用在变量的前面表示变量的值不可以改变，此时该变量可以被称为常量。
- final 用在方法的前面表示方法不可以被重写（子类中如果创建了一个与父类中相同名称、相同返回值类型、相同参数列表的方法，只是方法体中的实现不同，以实现不同于父类的功能，这种方式被称为方法重写，又称为方法覆盖。这里了解即可，教程后面我们会详细讲解）。
- final 用在类的前面表示该类不能有子类，即该类不可以被继承。

final 修饰变量

final 修饰的变量即成为常量，只能赋值一次，但是 final 所修饰局部变量和成员变量有所不同。

- final 修饰的局部变量必须使用之前被赋值一次才能使用。
- final 修饰的成员变量在声明时没有赋值的叫“空白 final 变量”。空白 final 变量必须在构造方法或静态代码块中初始化。

注意：final 修饰的变量不能被赋值这种说法是错误的，严格的说法是，final 修饰的变量不可被改变，一旦获得了初始值，该 final 变量的值就不能被重新赋值。

> 作者：编程导航知识星球 + 星球鱼友们

6、Java 中 hashCode 和 equals 方法是什么？它们和 == 各有什么区别?

官方解析

在 Java 中，hashCode 和 equals 方法都是 Object 类的方法。它们的作用分别如下：

- hashCode 方法返回对象的哈希码，用于支持基于哈希表的集合，如 HashMap、HashSet 等。如果两个对象的 equals 方法返回 true，则它们的 hashCode 方法必须返回相同的值，反之则不需要。
- equals 方法用于比较对象是否相等。默认情况下，equals 方法使用的是 == 操作符，即只有两个对象引用指向同一个对象时才会返回 true。但是，大部分情况下，我们需要重写 equals 方法来实现自己定义的相等规则。

两者之间的区别在于，hashCode 方法返回的是一个 int 类型的数值，而 equals 方法返回的是一个 boolean 类型的值。

hashCode 方法用于快速比较两个对象是否不同，因为如果它们的哈希码不同，那么它们肯定不相等。equals 方法则用于判断两个对象是否真正相等，这个判断比较复杂，需要根据对象的实际情况来定义。

另外，需要注意的是，== 操作符用于比较两个对象的引用是否相等，即它们是否指向同一个对象。而 equals 方法则用于比较两个对象的值是否相等。

在 Java 中，对象的值比较往往需要根据对象的实际情况来定义，因此一般需要重写 equals 方法。

鱼皮补充：注意，因为hashCode 可能会出现 hash 碰撞，所以导致不相等的两个对象 hash 码可能相等。

林风补充：String类对equals()方法进行了重写用户比较字符串的值是否相等。

鱼友的精彩回答

苏打饼干的回答

	类型	返回类型	本质
hashCode	Object类中定义的方法	`int`	返回一个代表对象哈希码的整数值 必须遵循equals`方法被认定为相等的两个对象，必须有相同的哈希码
equals	Object类中定义的方法	`boolean`	默认通过 `==` 比较两个对象 可在类中通过重写此方法定义其他相等性逻辑
==	运算符	`boolean`	对于基本类型：比较值是否相等 对于引用类型：比较内存地址是否相等

潜的回答

hashCode和equals方法是Object类的两个重要方法，用于判断对象的相等性和生成对象的哈希值。

hashCode：

- 可以这样理解：它返回的就是根据对象的内存地址换算出的一个值。

==：

== 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

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：

- 如果两个对象equals相同,hashCode一定相同
- 如果两个对象equals不同,hashCode不一定不同
- 如果两个对象的hashCode相同，它们的equals并不一定相同
- 如果两个对象的hashCode不相同，它们的equals一定不相同

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

7、什么是反射机制？说说反射机制的优缺点、应用场景？

官方解析

Java 反射机制是指在运行时动态地获取类的信息、创建对象以及调用对象的属性和方法的机制。Java 反射机制提供了运行时检查 Java 类型信息的能力，让 Java 程序可以通过程序获取其本身的信息。

Java 反射机制的优点：

- 可以动态地获取类的信息，不需要在编译时就知道类的信息。
- 可以动态地创建对象，不需要在编译时就知道对象的类型。
- 可以动态地调用对象的属性和方法，可以在运行时动态地改变对象的行为。

Java 反射机制的缺点：

- 由于反射是动态的，所以它的运行效率较低，不如直接调用方法或属性。
- 由于反射是动态的，所以它会破坏 Java 的封装性，可能会使代码变得复杂和不稳定。

Java 反射机制的应用场景：

- 动态代理。动态代理可以使用反射机制在运行时动态地创建代理对象，而不需要在编译时就知道接口的实现类。
- 单元测试。JUnit 等单元测试框架可以使用反射机制在运行时动态地获取类和方法的信息，实现自动化测试。
- 配置文件加载。许多框架（如 Spring）使用反射机制来读取和解析配置文件，从而实现依赖注入和面向切面编程等功能。

> 鱼皮补充：这里如果能像题解的最后一点一样提到哪些框架或者你自己的哪些项目用到了反射，会很加分

鱼友的精彩回答

林寻的回答

反射就是 Reflection，Java 的反射是指程序在运行期可以拿到一个对象的所有信息。
正常情况下，如果我们要调用一个对象的方法，或者访问一个对象的字段，通常会传入对象实例

➢在运行时判断任意一个对象所属的类

➢在运行时构造任意一个类的对象

➢在运行时判断任意一个类所具有的成员变量和方法

➢在运行时获取泛型信息

➢在运行时调用任意一个对象的成员变量和方法

➢在运行时处理注解

➢生成动态代理

优点:

➢可以实现动态创建对象和编译，体现出很大的灵活性

缺点:

➢对性能有影响.使用反射基本上是-种解释操作，我们可以告诉JVM，我们希望做什么并且它满足我们的要求。这类操作总是慢于直接执行相同的操作。

yes.的回答

Java的反射机制是指在运行时获得类的信息，创建类的对象，调用其中的方法和属性。 反射机制的优点：

- 可以动态的获取类信息。
- 可以动态的创建类对象。
- 可以动态的调用类对象的方法以及属性。
  反射机制的缺点：
- 不安全，反射能够获取所有类对象，包括私有的，破坏了程序的封装性。
- 效率低，反射的效率较低。

应用场景：

- 动态代理，因为不确定需要代理的类，所以需要通过反射动态的获取
- RPC 框架，RPC 框架就是动态的生成类对象，然后调用方法的。

> 鱼皮评论：rpc 这里提的很好

8、Java 访问修饰符 public、private、protected，以及无修饰符（默认）的区别

官方解析

在 Java 中，访问修饰符指的是控制类、接口、方法、属性等成员的访问范围。Java 提供了四种访问修饰符，分别为 public、private、protected 和默认（无修饰符）。

- public：可以被任何类或对象访问。
- private：只能被定义该成员的类访问，其他类无法访问。
- protected：可以被当前类、子类和同一个包中的类访问。
- 默认（无修饰符）：可以被同一个包中的类访问。

下面简要总结一下各个访问修饰符的特点：

- public 可以被任何类或对象访问，因此其访问范围最大，但也可能会存在安全问题。
- private 限制了访问范围，可以有效保护数据的安全，但是可能会增加代码的耦合度。
- protected 提供了一种在继承中使用的访问控制方式，但是可能会导致模块间的耦合。
- 默认（无修饰符）访问范围比 protected 更小，只能被同一个包中的类访问，可以减小模块间的耦合。

访问修饰符的选择需要根据具体情况来考虑，不能一概而论。通常情况下，应该尽可能地将成员设置为 private，只在需要的情况下使用 public 或 protected。

需要注意的是，在同一个类中，成员可以直接访问其他成员，无论其访问修饰符是什么。

鱼友的精彩回答

码不矫情的回答

> 鱼皮评论：一图胜千言

9、String 和 StringBuffer、StringBuilder 的区别是什么?

官方解析

String 和 StringBuffer/StringBuilder 是 Java 中两种不同的字符串处理方式，主要的区别在于 **String 是不可变的**（immutable）对象，而 **StringBuffer 和 StringBuilder 则是可变的**（mutable）对象。

String 对象**一旦被创建，就不可修改**，任何的字符串操作都会返回一个新的 String 对象，这可能导致**频繁的对象创建和销毁**，影响性能。而 StringBuffer 和 StringBuilder 允许进行修改操作，提供了一种更高效的字符串处理方式。

StringBuffer 和 StringBuilder 的主要区别在于**线程安全性和性能**方面。**StringBuffer 是线程安全的**，所有方法都是同步的，因此可以被多个线程同时访问和修改。而 **StringBuilder 不是线程安全的**，适用于单线程环境下的字符串处理，但是相比于 StringBuffer，StringBuilder 具有更高的性能。

因此，当字符串处理需要频繁修改时，建议使用 StringBuffer 或 StringBuilder；而当字符串处理不需要修改时，可以使用 String。

> 鱼皮补充：这道题是一个高频考点，感兴趣的同学可以实际测试下 StringBuffer 和 StringBuilder 的性能，加深印象

鱼友的精彩回答

林风的回答

总的来说这三者都是用来**操作字符串**的类。

String 是**不可变类**，在 Java 当中提供了多种操作字符串的方法，比如字符串的拼接、字符串的裁剪等。**每次这些操作的发生都会产生新的对象**。

这样来看的话似乎 String 并不适合去做字符串的各种操作，这种不可变的特性也带来了一定的好处。

1、免去了后续 String 字符串的创建。

由于 Strint Pool 的存在，当第一次字符串创建了之后，那么就可以根据引用直接在 String Pool 中获取到相应数据。

如果 String 不是不可变的，那么这个引用就会一直改变，不利于维护字符串常量池。

2、保证安全性

String 经常作为参数，String 不可变性可以保证参数不可变。

例如在作为网络连接参数的情况下如果 String 是可变的，那么在网络连接过程中，String 被改变，改变 String 的那一方以为现在连接的是其它主机，而实际情况却不一定是。

String 不可变性天生具备**线程安全**，可以在多个线程中安全地使用。

StringBuffer 解决 String 拼接产生太多中间对象的问题.append 或者 add 方法，把字符串添加到已有序列的末尾或者指定位置。

线程安全的可修改字符序列，它保证了线程安全，也随之带来了额外的性能开销

StringBuilder 是 Java 1.5 中新增的，在能力上和 StringBuffer 没有本质区别，但是它去掉了线程安全的部分，有效减小了开销，是绝大部分情况下进行字符串拼接的首选。

苏打饼干的回答

	可变性	线程安全	应用
String	不可变的每次修改都会创建一个新对象	√	字符串常或减少修改次数时
StringBuffer	可变的修改可在原对象上进行	√（同步）	频繁修改且在多线程中访问，需保证线程安全
StringBuilder	可变的	×	频繁修改且在单线程中访问，速度快

10、什么是 Java 内部类？ 内部类的分类有哪些？ 内部类有哪些优点和应用场景?

官方解析

内部类是定义在另一个类中的类。Java 中内部类主要分为**成员内部类、静态内部类、局部内部类和匿名内部类**四种。

1. **成员内部类**：定义在另一个类的内部，并且与其它成员变量和方法平级，可以访问外部类的所有成员变量和方法。使用方式：

```
Outer.Inner inner = new Outer().new Inner()。
```

2. **静态内部类**：定义在另一个类的内部，但是要用 **static** 修饰。只能访问外部类的静态成员变量和方法。使用方式：

```
Outer.Inner inner = new Outer.Inner()。
```

3. **局部内部类**：定义在**方法**中，作用域仅限于方法内部。与局部变量类似，不能使用访问控制符修饰。使用方式：在方法中直接实例化。
4. 匿名内部类：没有名字的内部类。使用方式：new 接口或者抽象类() { } 或 new 父类() { }。

内部类的优点：

- 可以访问外部类的私有成员变量和方法。
- 可以隐藏实现细节。
- 便于编写和维护，提高代码的可读性和可维护性。
- 内部类可以很好地解决Java中单继承的问题。

内部类的应用场景：

- 需要访问外部类的私有成员变量和方法。
- 需要定义一个回调函数或者监听器。
- 需要实现多重继承。
- 需要对外部类进行扩展。

鱼友的精彩回答

苏打饼干的回答

顾名思义，内部类是指定义在某一个类中的类，主要分为**成员内部类，静态内部类，局部内部类和匿名内部类**四种。

分类	定义	作用范围	应用
成员内部类	与其外部类的其他他成员变量和方法平级的内部类	可以访问外部类的所有成员变量和方法 可以被外部类访问	为外部类提供服务的场景
静态内部类	用 `static` 修饰的内部类	只能访问外部类的静态成员变量和方法 只能被外部类的静态方法访问	与外部无关或者需要单例模式的场景
局部内部类	定义在方法中的内部类 在方法中直接实例化	可以访问外部类或代码块中声明为 `final` 的变量 作用域仅限于方法内 不能使用访问控制符修饰	执行特定任务或逻辑操作场景
匿名内部类	没有名字的内部类 使用 `new接口` 或 `抽象类(){}` 或 `new 父类(){}`	没有名字、没有构造器、只能使用一次 必须继承一个父类型（接口或抽象类）或实现一个接口的局部内部类	创建回调函数、事件监听器、线程等临时性对象

创建与获取

```
/ 1、私有内部类 => 在外部类中编写方法，对外提供内部类对象
// 定义方法（外部类中）
public Inner getInstance(){
  return new Inner();
}
// 使用方法
Outer o = new Outer();
Object i = o.getInstance();

// 2. 非私有内部类 => 直接创建成员内部类
// 外部类名.内部类名 对象名 = new 外部类对象.new 内部类对象;
Outer.Inner oi = new Outer().new Inner();
```

优点

1. 可以隐藏实现细节。
2. 便于编写和维护，提高代码的可读性和可维护性。
3. 使用内部类解决 Java单继承问题
4. 可以更换的对外部类进行扩展

注：JDK16 之前成员内部类里不能定义静态变量

爱吃鱼蛋的回答

内部类是指定义在另一个类内部的类，它可以访问外部类的成员变量和方法。

内部类的分类有以下几种：

- **成员内部类**：定义在类内部，但在方法外部的类。**它可以访问外部类的所有成员变量和方法**；
- **静态内部类**：定义在类内部，但使用 static 修饰的类。**它只能访问外部类的静态成员变量和方法**；
- 局部内部类：定义在**方法内部的类**。它只能访问方法内部的 final 变量和方法参数；
- 匿名内部类：**没有类名的内部类**，通常用于创建只需要使用一次的类。

它们主要有以下区别：

- 成员内部类和静态内部类都可以定义在外部类中，成员内部类需要外部类实例化才能使用，而静态内部类不需要。
- 成员内部类和静态内部类都可以定义静态成员，但成员内部类只能在非静态方法中使用，而静态内部类可以在任何地方使用。
- 局部内部类只能定义在方法中，不能定义静态成员，也不能访问外部类成员。
- 匿名内部类没有类名，通常用于创建一个实现某个接口或继承某个类的对象，可以访问外部类成员。

类型	定义位置	是否需要外部类实例化	是否可以定义静态成员	是否可以访问外部类成员
成员内部类	外部类中	是	是	是
静态内部类	外部类中	否	是	否
局部内部类	方法中	否	否	否
匿名内部类	方法中	否	否	是

内部类的合理使用存在以下优点：

- 通过内部类变相地实现多继承，使得一个类可以假性继承多个父类；
- 内部类可以隐藏实现细节，从而简化代码的编写；
- 内部类可以访问外部类的私有成员变量和方法，从而增加了程序的灵活性和安全性。

内部类常见的使用场景如下：

- 可用于实现**回调函数**提供给外部类使用；
- 通过内部类中定义静态成员变量，利用内部类的加载机制，从而实现**单例模式**；
- 有多继承需求时可使用内部类实现**假性多继承**。

> 作者：编程导航知识星球 + 星球鱼友们

MySQL 数据库

1、什么是数据库事务？讲一下事务的 ACID 特性?

官方解析

数据库事务是指数据库管理系统（DBMS）中的一个操作序列，这些操作必须作为一个不可分割的单元执行，即要**么全部执行成功，要么全部失败回滚**。事务通常涉及到对数据库中的数据进行读写操作。

事务的 **ACID** 特性指四个关键特征：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）。

1. **原子性**（Atomicity）：事务是一个原子操作，要么全部提交，要么全部回滚。当一个事务执行期间发生故障，操作系统会自动将其回滚到事务执行之前的状态，保证数据的一致性。
2. **一致性**（Consistency）：事务执行结束后，数据必须保持一致性状态。在事务执行期间，数据库中的数据可以处于中间状态，但在事务完成时必须保证数据的一致性。
3. **隔离性**（Isolation）：数据库系统必须保证事务之间相互隔离，不会互相干扰。隔离级别不同，会影响到事务的并发性和数据一致性，比如出现脏读、不可重复读、幻读等问题。
4. **持久性**（Durability）：一旦事务提交，其所做的修改必须永久保存到数据库中。即使系统发生故障或宕机，数据也能够保持不变。

ACID 特性是保证事务正确性和数据一致性的重要手段。在设计数据库应用程序时，应该根据具体的业务需求和数据安全性要求，选择合适的隔离级别和事务提交策略，保证事务的可靠性和数据的一致性。

鱼友的精彩回答

叁觀同学的回答

ACID特性：

数据库管理系统中事务(transaction)的四个特性（分析时根据首字母缩写依次解释）：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）

所谓事务，它是一个操作序列，这些操作要么都执行，要么都不执行，它是一个**不可分割的工作单位**。（执行单个逻辑功能的一组指令或操作称为事务）

1. **原子性**

原子性是指事务是**一个不可再分割的工作单元**，事务中的操作要么都发生，要么都不发生。可采用“A向B转账”这个例子来说明解释。

在 DBMS 中，默认情况下一条 SQL 就是一个单独事务，事务是自动提交的。只有显式的使用 start transaction 开启一个事务，才能将一个代码块放在事务中执行。

2. **一致性**

一致性是指在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。这是说数据库事务不能破坏关系数据的完整性以及业务逻辑上的一致性。

如 A 给 B 转账，不论转账的事务操作是否成功，其两者的存款总额不变（这是业务逻辑的一致性，至于数据库关系约束的完整性就更好理解了）。

3. **隔离性**

多个事务并发访问时，事务之间是隔离的，一个事务不应该影响其它事务运行效果。

在并发环境中，**当不同的事务同时操纵相同的数据**时，每个事务都有**各自的完整数据空间**。由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。事务查看数据更新时，数据所处的状态要么是另一事务修改它之前的状态，要么是另一事务修改它之后的状态，**事务不会查看到中间状态的数据**。

事务最复杂问题都是由事务隔离性引起的。完全的隔离性是不现实的，完全的隔离性要求数据库同一时间只执行一条事务，这样会严重影响性能。

有四种隔离级别：

第一种隔离级别：**Read uncommitted(读未提交)**

- 解决了更新丢失，但还是可能会出现脏读

第二种隔离级别：**Read committed(读提交)**

- 解决了更新丢失和脏读问题

第三种隔离级别：**Repeatable read(可重复读取)**

- 解决了更新丢失、脏读、不可重复读、但是还会出现幻读

第四种隔离级别：**Serializable(可序化)**

- 解决了更新丢失、脏读、不可重复读、幻读(虚读)

4. 持久性

这是最好理解的一个特性：**持久性**，意味着在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。（完成的事务是系统永久的部分，对系统的影响是永久性的，该修改即使出现致命的系统故障也将一直保持）

yes.的补充

InnoDB中：

- 通过 undo log 支持事务回滚、当前读（多版本查询）
- 通过 redo log 实现持久性
- 通过两阶段提交实现一致性
- 通过当前读、锁实现隔离性

2、MySQL 日志有了解过吗？binlog、redolog、undolog 分别有什么作用、有什么区别?

官方解析

MySQL 是一款流行的关系型数据库，其日志是其关键功能之一。MySQL 包括三种类型的日志，分别是 binlog、redolog 和 undolog，它们分别有不同的作用和特点。

- binlog ，binlog（Binary log）是 MySQL 中的二进制日志文件，用于记录 MySQL 服务器上的所有更新和修改操作。它可以记录所有的 DDL（Data Definition Language）和 DML（Data Modification Language）操作，包括对表结构的更改、数据的插入、修改、删除等等。binlog是在事务提交后生成的，因此可以用于恢复数据库。
- redolog ，redolog（Redo log）用于恢复数据，保证数据的一致性和持久性。当 MySQL 发生修改时，redolog 会将这些操作记录下来，并写入磁盘。这样，当 MySQL 发生宕机或崩溃时，通过重放 redolog 就可以恢复数据。
- undolog，undolog（Undo log）用于回滚操作。当 MySQL 发生事务回滚时，undolog 会记录这些操作并将其写入磁盘。这样，当 MySQL 需要回滚时，通过重放 undolog 就可以回滚事务。

区别：

- binlog 和 redolog 都是 MySQL 中的二进制日志，但是它们的作用和实现方式有所不同。binlog 是 MySQL 记录所有的操作，而 redolog 则是用于保证数据的一致性和持久性。此外，binlog 是逻辑日志，redolog 是物理日志。binlog 记录的是SQL语句，而 redolog 记录的是数据页的修改，所以 binlog 可以跨平台使用，而 redolog 不能。undolog 和 redolog 的区别是，undolog 是用于回滚操作的，而 redolog 是用于恢复数据的。

鱼友的精彩回答

星河的回答

binlog：即存档日志，是 Server 层生成的的日志，主要用于数据备份和主从复制；

redolog：即重做日志，是 Innodb 存储引擎层的日志，是 Mysql 实现原子性和一致性的重要保证，主要用于事务回归和 MVCC；

uodolog：即回滚日志，是 Innodb 存储引擎层的日志，是 Mysql 实现持久性的重要保证，主要用于数据库事故故障恢复；

下面我们详细说说三种日志：

首先当我们执行一条增删改 sql 语句（没用begin）时，mysql 会隐式开启事务执行该条语句，在执行完毕后 mysql 会自动提交事务，我们就能看见增删改的实际结果，但是如果在事务执行中，mysql 在没提交事务的时候崩溃了，那么数据有问题了，此时就需要回滚到事务之前的数据，它本质就是用于撤销回退的日志（ctrl+z），在事务没提交之前，mysql 会记录更新前的数据到 undolog 中，当事务回滚时，就利用 uodolog 日志进行回滚：

- 在**插入**一条记录时，要把这条记录的主键值记下来，这样之后回滚时只需要把这个主键值对应的记录删掉；
- 在**删除**一条记录时，要把这条记录中的内容都记下来，这样之后回滚时再把由这些内容组成的记录插入到表中；
- 在**更新**一条记录时，要把被更新的列的旧值记下来，这样之后回滚时再把这些列更新为旧值。

一条记录的 uodolog 格式由 trx_ 事务 id + roll_pointer 指针串成一个链表，即**版本链**。此外**在读提交和可重复读隔离级别中，它和快照 readView 实现了 MVCC**。

然后是 redolog，它是为了防止机器故障导致数据丢失的物理日志，它用于记录数据页做了什么修改，每当执行一条事务就会产生这样的一条或多条物理日志，然后通过**顺序写**写入磁盘。它和 uodolog 区别是：**redolog 记录了此次事务完成后的数据状态，记录的是更新之后的值；undolog 记录了此次事务开始前的数据状态，记录的是更新之前的值**；它保证了数据的持久性。

binlog 是逻辑日志，在完成一条更新操作后，server 层会生成 binlog，等事务提交时，将此事务运行中产生的所有 DDL 记录通过追加写统一写入 binlog 日志文件，保存的是全量的日志，用于备份恢复和主从复制。

3、数据库索引是什么，有什么作用，什么场景适合使用索引?

官方解析

数据库索引是一种数据结构，用于提高数据库表的查询效率。

索引可以帮助数据库快速定位和检索存储在表中的数据，从而加快数据查询的速度。在数据量比较大时，使用索引可以极大地提高数据检索的效率。

索引的作用是通过构建一个额外的数据结构（B-tree、哈希表等）来加速数据的检索。它是在数据库表上创建的一种数据结构，它包含一些指向表中数据的指针，可以快速地定位到满足查询条件的数据行，从而提高查询效率。索引可以包含一个或多个列，可以使用单列索引、组合索引、全文索引等多种方式来创建。

适合使用索引的场景包括：

- 频繁查询的列，如主键、外键等。

- 经常作为查询条件的列，如 WHERE、ORDER BY、GROUP BY 等语句中的列。
- 经常需要连接的列，如多表联合查询时的列。
- 数据量较大的表，通过索引可以加快数据检索速度。

索引的优点是可以提高数据库的查询速度，缩短数据检索的时间，提高系统的性能。但是索引也有一些缺点，包括：

- 占用额外的存储空间，增加了存储成本。
- 建立索引需要时间，增加了系统的开销。
- 数据库的更新操作（增删改）会导致索引的重建，影响系统的性能。

因此，需要根据实际情况进行索引的创建和使用，避免过度索引导致系统性能下降。

> 鱼皮补充：这题如果能举例你自己是如何在项目中应用索引的（比如检索用户消息）、或者说什么情况下你没有选择用索引的（比如性别字段），会很加分

鱼友的精彩回答

林风的回答

数据库索引是一种数据结构，就想书的目录一样，它可以帮助我们快速定位到想要的数据。

优点

- 可以提高数据检索的效率，降低数据库的IO成本，类似于书的目录。
- 通过索引列对数据进行排序，降低数据排序的成本，降低了CPU的消耗。
  缺点
- 需要占用物理空间，数量越大，占用空间越大；
- 索引虽然会提高查询效率，但是会降低更新表的效率。比如每次对表进行增删改操作，MySQL不仅要保存数据，还有保存或者更新对应的索引文件。

索引不是万能钥匙，它也是根据场景来使用的

什么场景适合使用索引?

- 频繁使用的列，主键、外键
- 字段有唯一性限制的，比如商品编码，可以使用唯一索引
- 经常用于 WHERE 查询条件的字段，这样能够提高整个表的查询速度，如果查询条件不是一个字段，可以建立联合索引。
- 经常用于 GROUP BY 和 ORDER BY 的字段，这样在查询的时候就不需要再去做一次排序了，因为我们都已经知道了建立索引之后在 B+Tree 中的记录都是排序好的

还有一点要注意的是索引失效的场景，本来使用索引是为了加快检索速度的，一旦索引失效，不仅起不了作用还占用空间。

索引失效

- 使用函数、计算、类型转换

使用这些以后，索引中存的不是函数计算后的值，自然用不到索引

- like %xxx %xxx%

不符合最左匹配原则

- or 只有一边用上了索引，也会导致索引失效
- 联合索引的最左匹配原则，在遇到范围查询（>、<、between、like 包括like '林%'这种）的时候，就会停止匹配

什么时候不适合创建索引?

- **WHERE 条件，GROUP BY，ORDER BY 里用不到的字段**，索引的价值是快速定位，如果起不到定位的字段通常是不需要创建索引的，因为索引是会占用物理空间的。
- **字段中存在大量重复数据**，不需要创建索引，比如性别字段，只有男女，如果数据库表中，男女的记录分布均匀，那么无论搜索哪个值都可能得到一半的数据。在这些情况下，还不如不要索引，因为 MySQL 还有一个查询优化器，查询优化器发现某个值出现在表的数据行中的百分比很高的时候，它一般会忽略索引，进行全表扫描。
- **表数据太少**的时候，不需要创建索引；
- **经常更新**的字段不用创建索引，比如不要对电商项目的用户余额建立索引，因为索引字段频繁修改，由于要维护 B+Tree的有序性，那么就需要频繁的重建索引，这个过程是会影响数据库性能的。

4、你是怎么做 MySQL 数据备份的？比如怎么恢复半个月前的数据?

官方解析

MySQL 数据备份是一个非常重要的工作，保证数据的安全性和可靠性。备份数据的方式有很多种，以下是其中一种基本的备份和恢复方式：

一、备份 MySQL 数据库

我们可以使用 mysqldump 工具来备份 MySQL 数据库，该工具可以生成 SQL 脚本文件，包含数据库中所有表和数据的语句。在终端中运行以下命令：

```
mysqldump -u [username] -p [database_name] > [backup_file].sql
```

其中，[username] 是 MySQL 用户名，[database_name] 是需要备份的数据库名称，[backup_file].sql 是备份的文件名。

该命令会将 SQL 脚本文件导出到当前目录下。

二、恢复 MySQL 数据库

如果需要恢复之前备份的数据，可以运行以下命令：

```
mysql -u [username] -p [database_name] < [backup_file].sql
```

其中，[username] 是 MySQL 用户名，[database_name] 是需要恢复的数据库名称，[backup_file].sql 是备份的文件名。

该命令会将备份文件中的 SQL 语句执行，从而将数据恢复到指定的数据库中。

如果需要恢复半个月前的数据，可以选择备份文件中的某个时间点之前的数据，并使用以上方法进行恢复。

此外，还有其他的备份方式，如使用 MySQL 自带的 mysqlbinlog 工具进行增量备份，或使用第三方备份软件进行备份。根据实际需求选择合适的备份方式，并将备份文件存放在可靠的位置。

> 鱼皮补充：mysql 备份操作是可以写在简历上的，推荐大家自己实践试试

鱼友的精彩回答

解州扯面的回答

MySQL 数据备份有多种方法，包括**逻辑备份、物理备份、全备份和增量备份**。你可以根据你的需求和环境选择合适的方式。如果你想要恢复半个月前的数据，你需要先确保你有半个月前的备份文件，然后使用相应的工具或命令进行还原。例如，如果你使用 mysqldump 命令进行了逻辑备份，那么你可以使用 source 指令或者 mysql 命令来导入备份文件。

逻辑备份

逻辑备份是使用 SQL 语句来导出和导入数据库的数据的方法。它的优点是简单易用，可以跨平台和存储引擎，可以部分备份和恢复。它的缺点是速度慢，占用空间大，可能影响数据库性能。

MySQL 提供了一个原生的逻辑备份工具叫做 mysqldump。你可以使用它来备份整个数据库实例、单个数据库或单张表。例如，如果你想要备份一个叫 ytt 的数据库，你可以在命令行窗口输入：

```
mysqldump -u 用户名 -p 密码 --database ytt > ytt.sql
```

这样就会生成一个包含 ytt 数据库所有数据的 SQL 文件。

如果你想要恢复这个文件到数据库中，你可以在命令行窗口输入：

```
mysql -u 用户名 -p 密码 < ytt.sql
```

或者

```
mysql -u 用户名 -p 密码 source ytt.sql
```

这样就会执行 SQL 文件中的语句，将数据还原到数据库中。

物理备份

物理备份是直接拷贝数据库的数据文件、配置文件和日志文件。它的优点是速度快，占用空间小，不影响数据库性能。它的缺点是恢复复杂，需要关闭数据库服务，可能导致数据不一致。

MySQL 有一个开源的物理热备工具叫做 Percona XtraBackup。它可以在不锁表的情况下备份 InnoDB、XtraDB 和 MyISAM 存储引擎的表，并且支持增量备份。你可以使用它来备份整个数据库实例或单个数据库。

例如，如果你想要全量备份一个叫 ytt 的数据库，你可以在命令行窗口输入：

```
innobackupex --user=用户名 --password=密码 --databases="ytt" /backup
```

这样就会在 /backup 目录下生成一个包含 ytt 数据库所有数据文件的目录。

如果你想要恢复这个目录到数据库中，你需要先准备好数据文件，然后拷贝到对应的数据目录中，并更改权限和所有者。具体步骤如下：

```
innobackupex --apply-log /backup/2023-02-27_00-00-00 cp -r /backup/2023-02-27_00-00-00/* /var/lib/mysql chown -R mysql:mysql /var/lib/mysql
```

请注意，在恢复之前，你需要停止 MySQL 服务，并确保数据目录为空。

全备份和增量备份

全备份是指备份数据库中的所有数据，不管数据是否有变化。它的优点是恢复简单，不需要其他文件。它的缺点是占用空间大，耗时长，影响数据库性能。

增量备份是指只备份上次全备份或增量备份后发生变化的数据，需要开启 binlog 日志功能. 它的优点是占用空间小，耗时短，不影响数据库性能，它的缺点是恢复复杂，需要依赖 binlog 日志文件和全备份文件。

例如，如果你想要每天进行一次全备份和每小时进行一次增量备份，你可以使用 mysqldump 工具和 binlog 工具来实现。

首先，在 my.cnf 文件中开启 binlog 功能，并设置每天生成一个日志文件：

[mysqld] log-bin=/var/lib/mysql/mysql-bin expire-logs-days=7 max_binlog_size=100M

然后，在 crontab 中设置定时任务：

```
`# 每天凌晨 0 点进行一次全备份 0 0 * * * mysqldump -u root -p password --all-databases > /backup/full_$(date +%Y%m%d).sql
```

每小时进行一次增量备份

```
0 * * * * mysqlbinlog --read-from-remote-server --host=localhost --user=root --password=password --stop-never-slave-server-id=10 mysql-bin.000001 > /backup/incremental_$(date +%Y%m%d%H).sql`
```

这样就可以在 /backup 目录下生成全备份和增量备份文件。

如果你想要恢复这些文件到数据库中，你需要先停止 MySQL 服务，并确保数据目录为空。然后按照以下步骤操作：

```
`# 恢复最近一次的全备份文件 mysql -u root -p password < /backup/full_20230227.sql
```

恢复之后产生的所有增量备份文件

```
mysqlbinlog /backup/incremental_202302270*.sql | mysql -u root -p password`
```

请注意，在恢复之前，你需要确认 binlog 文件名和路径，并按照时间顺序恢复。

5、一条 SQL 语句在 MySQL 中的执行过程是怎样的?

官方解析

在 MySQL 中，一条 SQL 语句的执行过程通常可以分为以下几个步骤：

词法分析和语法分析：MySQL 的 SQL 解析器会对输入的 SQL 语句进行词法分析和语法分析，**以确定语句的结构和语法是否正确**。

查询优化：MySQL 会对 SQL 语句进行优化，**以确定最优的执行计划**。在这个过程中，MySQL 会考虑许多因素，例如**索引、表连接、统计信息**等，以找到执行查询的最有效方式。

查询执行：在查询优化后，MySQL 开始**执行查询，读取和处理数据**。在执行过程中，MySQL 会根据查询中所涉及的表和列等信息，从磁盘中读取相应的数据，并进行计算和过滤操作。

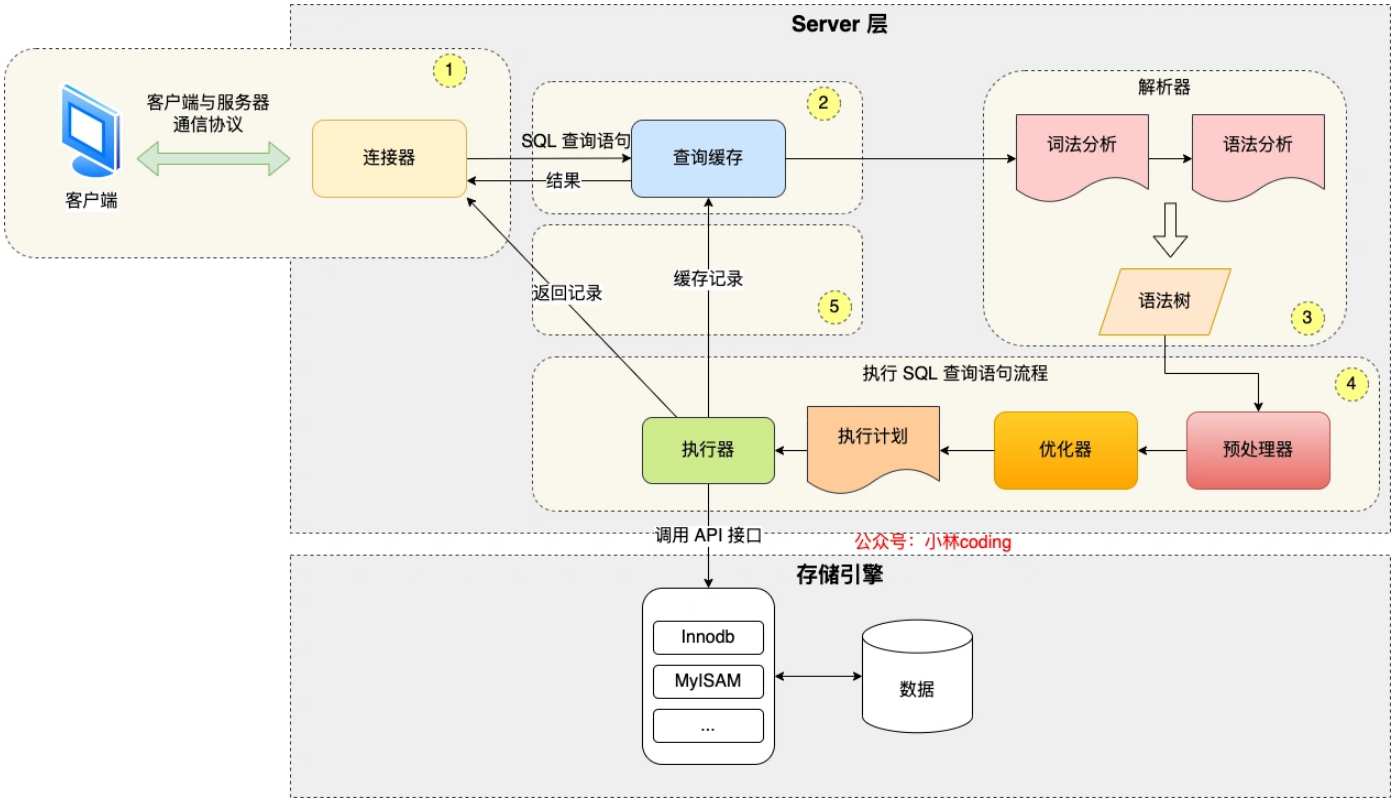
结果返回：最后，MySQL 会将查询结果返回给客户端，完成整个查询过程。

需要注意的是，实际的执行过程可能会因为多种因素而不同，例如数据量、硬件配置等。

另外，在并发环境下，多个查询可能会同时进行，需要使用**锁和事务**等机制来保证数据的一致性和正确性。

鱼友的精彩回答

Ming 的回答



第一步：连接器

过程

1. **建立连接**：与客户端进行 TCP 三次握手建立连接；
2. **校验密码**：校验客户端的用户名和密码，如果用户名或密码不对，则会报错；

3. **权限判断**：如果用户名和密码都对了，会读取该用户的权限，然后后面的权限逻辑判断都基于此时读取到的权限；

其他相关问题

如何查看 MySQL 服务被多少个客户端连接了?

```
mysql> show processlist;
```

空闲连接会一直占用着吗?

- 空闲连接的最大空闲时长，由 wait_timeout 参数控制的
- 查询命令

```
mysql> show variables like 'wait_timeout';
```

- 手动断开

```
mysql> kill connection +6;
```

MySQL 的连接数有限制吗?

MySQL 服务支持的最大连接数由 max_connections 参数控制。

MySQL 的连接也跟 HTTP 一样，有短连接和长连接的概念。

怎么解决长连接占用内存的问题?

- 定期断开长连接
- 客户端主动重置连接

第二步：查询缓存

过程

解析出 SQL 语句的第一个字段，看看是什么类型的语句。

如果 SQL 是查询语句（select 语句），MySQL 就会先去查询缓存（ Query Cache ）里查找缓存数据。

如果查询的语句命中查询缓存，那么就会直接返回 value 给客户端。

如果查询的语句没有命中查询缓存中，那么就要往下继续执行，等执行完后，查询的结果就会被存入查询缓存中。

缺点

- 更新比较频繁的表，查询缓存的命中率很低

版本变动

- MySQL 8.0 版本直接将查询缓存删掉了，也就是说 MySQL 8.0 开始，执行一条 SQL 查询语句，不会再走到查询缓存这个阶段了。 对于 MySQL 8.0 之前的版本，如果想关闭查询缓存，我们可以通过将参数 query_cache_type 设置成 DEMAND。

这里说的查询缓存是 server 层的，也就是 MySQL 8.0 版本移除的是 server 层的查询缓存，并不是 Innodb 存储引擎中的 buffer pool。

第三步：解析 SQL

过程

1. 词法分析
2. 语法分析
3. 语法不对，解析器就会给报错

注意：表不存在或者字段不存在，并不是在解析器里做的，解析器只负责构建语法树和检查语法，但是不会去查表或者字段存不存在。

第四步：执行 SQL

过程

1. prepare 阶段，也就是预处理阶段；
2. optimize 阶段，也就是优化阶段；
3. execute 阶段，也就是执行阶段；

1 预处理器

检查 SQL 查询语句中的表或者字段是否存在；

将 select * 中的 * 符号，扩展为表上的所有列；

2 优化器

优化器主要负责将 SQL 查询语句的执行方案确定下来

比如在表里面有多个索引的时候，优化器会基于查询成本的考虑，来决定选择使用哪个索引。

要想知道优化器选择了哪个索引，我们可以在查询语句最前面加个 explain 命令

3 执行器

执行器就会和存储引擎交互了，交互是以记录为单位的。

三种方式执行过程

- 主键索引查询
- 全表扫描
- 索引下推（MySQL 5.6 推出的查询优化策略）

特点

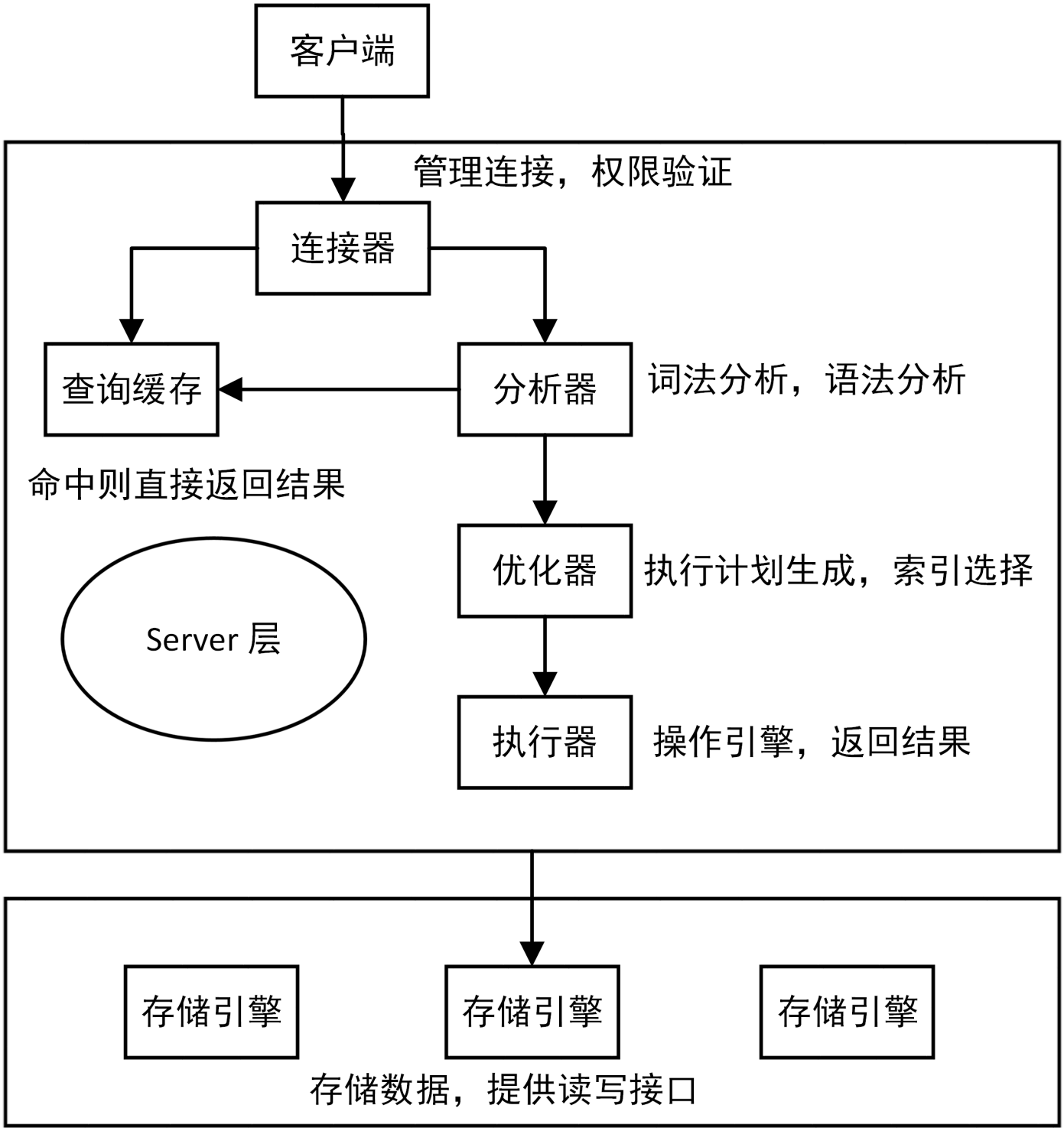
- 执行器查询的过程是一个 while 循环
- Server 层每从存储引擎读到一条记录就会发送给客户端，之所以客户端显示的时候是直接显示所有记录的，是因为客户端是等查询语句查询完成后，才会显示出所有的记录

总结

执行一条 SQL 查询语句，期间发生了什么?

- **连接器**：建立连接，管理连接、校验用户身份；
- **查询缓存**：查询语句如果命中查询缓存则直接返回，否则继续往下执行。MySQL 8.0 已删除该模块；
- **解析 SQL**，通过解析器对 SQL 查询语句进行词法分析、语法分析，然后构建语法树，方便后续模块读取表名、字段、语句类型；
- **执行 SQL**：执行 SQL 共有三个阶段：
    - **预处理阶段**：检查表或字段是否存在；将 select * 中的 * 符号扩展为表上的所有列。
    - **优化阶段**：基于查询成本的考虑，选择查询成本最小的执行计划；
    - **执行阶段**：根据执行计划执行 SQL 查询语句，从存储引擎读取记录，返回给客户端；

林风的回答



连接器：客户端首先进行身份验证，如果没通过直接返回。

查询缓存：如果身份验证不通过则查询缓存。缓存这个原理很容易理解，就是把常用的数据放到更高效的地方便于查询。 查询缓存也有缺点，就是每当数据更改的时候就要重新设置缓存，在 MySQL8.0 已经将查询缓存去掉。

分析器：分析器先会做“**词法分析**”。把你输入的内容进行识别，知道字符分别代表什么，有什么含义 然后进行**语法分析**，如果你的 SQL 语句不符合 MySQL 语法就会收到错误提醒。

优化器：优化器作用就是决定使用哪个索引，决定 join 表的连接顺序。 优化器会选择它自己认为最高效的方案，(也代表它不一定能选择出最优的方案)。

执行器：执行器还是先会判断有没有执行的权限，如果有权限的话才会执行下一步。遍历满足条件的行，并把组成的记录集作为结果集返回给客户端。

6、MySQL 中的索引是怎么实现的？B+ 树是什么，B 树和 B+ 树的区别，为什么 MySQL 要用 B+ 树?

官方解析

MySQL 中的索引是**通过 B+ 树实现**的。B+ 树是一种多叉树，它可以将数据按照一定的顺序组织起来，从而提高查询效率。

B+ 树与 B 树的区别在于，**B+ 树的所有数据都存储在叶子节点上，而非叶子节点只存储索引**，这样可以提高数据查询效率。B+ 树的**叶子节点之间使用指针相连**，这样可以实现**区间查找**，也就是说，可以快速定位某个区间内的数据。

MySQL 之所以采用 B+ 树作为索引的实现方式，主要是因为 B+ 树具有以下优点：

- 能够支持高效的范围查找和排序。
- 叶子节点之间使用指针相连，能够支持高效的区间查询。
- B+ 树具有较高的数据密度，可以减少磁盘 I/O 次数，提高查询效率。
- B+ 树对于插入和删除操作也比较高效。

在 MySQL 中，B+ 树的实现主要是通过 InnoDB 存储引擎来实现的。InnoDB 存储引擎中的索引主要有**聚簇索引**和**辅助索引**两种类型，**聚簇索引是根据主键创建的索引**，而**辅助索引是根据非主键列创建的索引**。

对于辅助索引，MySQL 中会同时创建一个对应的聚簇索引，这样可以提高查询效率。

鱼友的精彩回答

Gundam 的回答

MySQL 中的索引采用**B+树**实现。

B+树是一种**多路搜索树**，是对B树的一种改进。

B树和B+树的主要区别在于：

1. B+树只存储数据的索引信息，并且把叶子节点存储在同一层上，**中间节点只起到索引作用**，查询时只需要**遍历一次树**就能找到目标数据,因此B+树具有更好的查询性能和更少的磁盘I/O次数，**B+树的叶子节点都连接成了一个有序的链表**，因此可以很方便地进行**范围查询**等操作。适合于数据库等存储大量数据的场景。
2. B树是一种**平衡树**，它的每个节点都存储有序的关键字，并且每个节点的子节点数目介于m/2和m之间。B树中的每个节点既可以存储数据，也可以存储索引信息，因此B树可以减少I/O次数，适合于文件系统等存储大量数据的场景。但是B树的查询性能相对较低，因为每个节点都可能存储数据，查询时需要遍历多个节点才能找到目标数据。

在MySQL中使用B+树作为索引的数据结构，主要是因为B+树的**查询性能好**、**支持范围查询**、**支持数据分页**等操作，适用于存储海量数据的场景。此外，B+树的索引结构相对简单，易于实现和维护，能够满足高并发、高可用性的数据库要求。

维萨斯的回答

答案 + 解释

Mysql 中的索引是怎么实现的

- B+树、哈希索引、全文索引
  - 哈希索引 （ 一般用于等值操作，而且人工一般不能干涉 ）
  - 全文索引 （ 没有了解，因为比起使用 Mysql 的全文索引，用 ES 实现搜索更好 ）
- 根据存储引擎，b+树的具体实现有细微区别
  - MyISAM
    - 无聚簇索引
    - 叶子节点存的是 数据的地址
    - 每次查询都需要去回表
    - why？因为MyISAM的数据持久化方法是：数据和索引分别存储
  - InnoDB
    - **分聚簇索引和非聚簇索引**
    - 聚簇索引只有一个，它根据主键实现了b+树
    - 非聚簇索引有多个，它的叶子节点存的是索引 + 主键
    - why？ 因为InnoDB的数据持久化方法是：数据和索引一个文件

B+树 vs B树

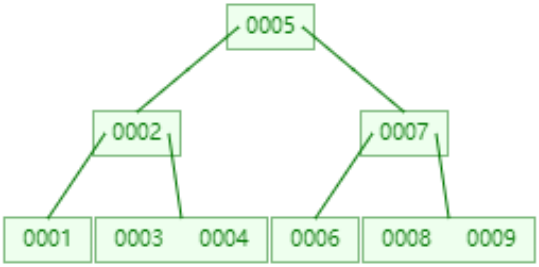
- B+树
  - 叶子节点存数据
- B树
  - 所有节点都存数据
- why choose B+
  - INSERT / DELETE
    - B树由于所有节点都保存所有数据，每当插入一条数据，哪怕是自增的，都可能造成整个树的自旋重构，当数据量很大的时候，这个时间成本和风险是巨大的
    - B+树使用叶子节点保存数据，插入一条数据只会在叶子节点上插入，一般不会影响树的结构
  - SELECT
    - B+树支持范围查找，同时查询更高效
    - why？因为叶子节点中，页于页之间是双向链表，而簇于簇之间有单向指针连接。

言的回答

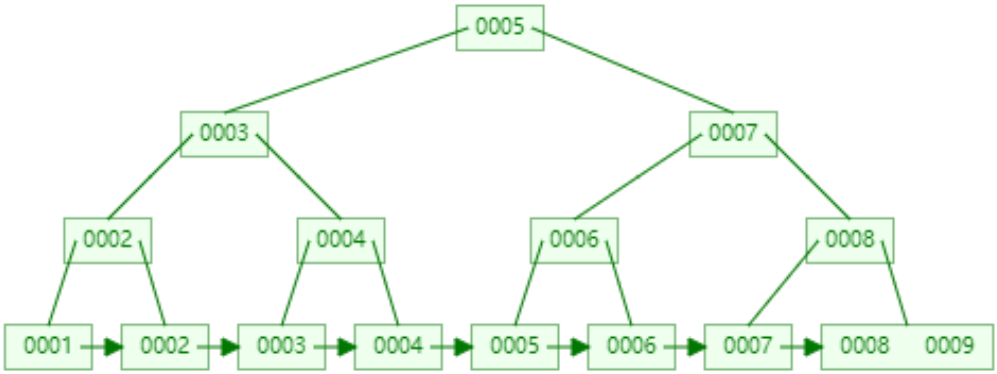
> http://www.rmboot.com/Algorithms.html 数据结构可视化，可以体验B树与B+树数据操作时，直观的样子。

使用B+树相比于B树的优点在于： **回旋查找** 的问题，也就是说当我们查询一段范围内的数据时，B+树是通过指针直接获取，而B树存在回旋查找的操作。下面我们可以通过图来展示区别：

B树插入9个数值



B+树插入9个数值



比如说我们现在要找 id >= 1 范围的数据

- 如果使用B树，找到1时，需要回旋查找2，再继续操作
- 如果使用B+树，找到1时，因为有指针，所以可以直接获取后面的数据

Yilin 的回答

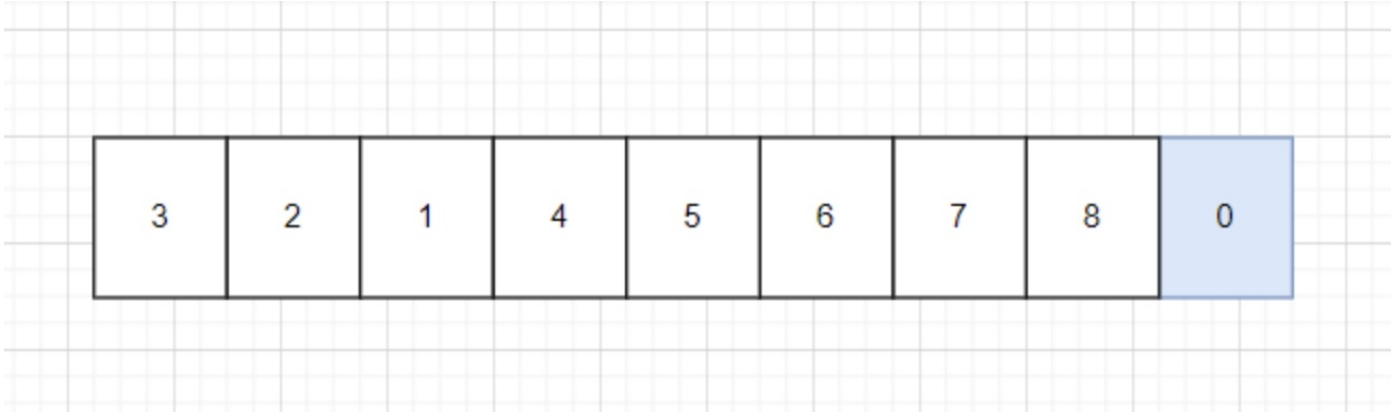
数据库的查询，如何设计存储结构使得数据访问更快（假设有1亿条数据）

查询磁盘中的数据起决定性因素的就是

1. 遍历数据的次数
2. 从磁盘加载到内存的IO时间，最重要的次数少了加载IO时间也少。

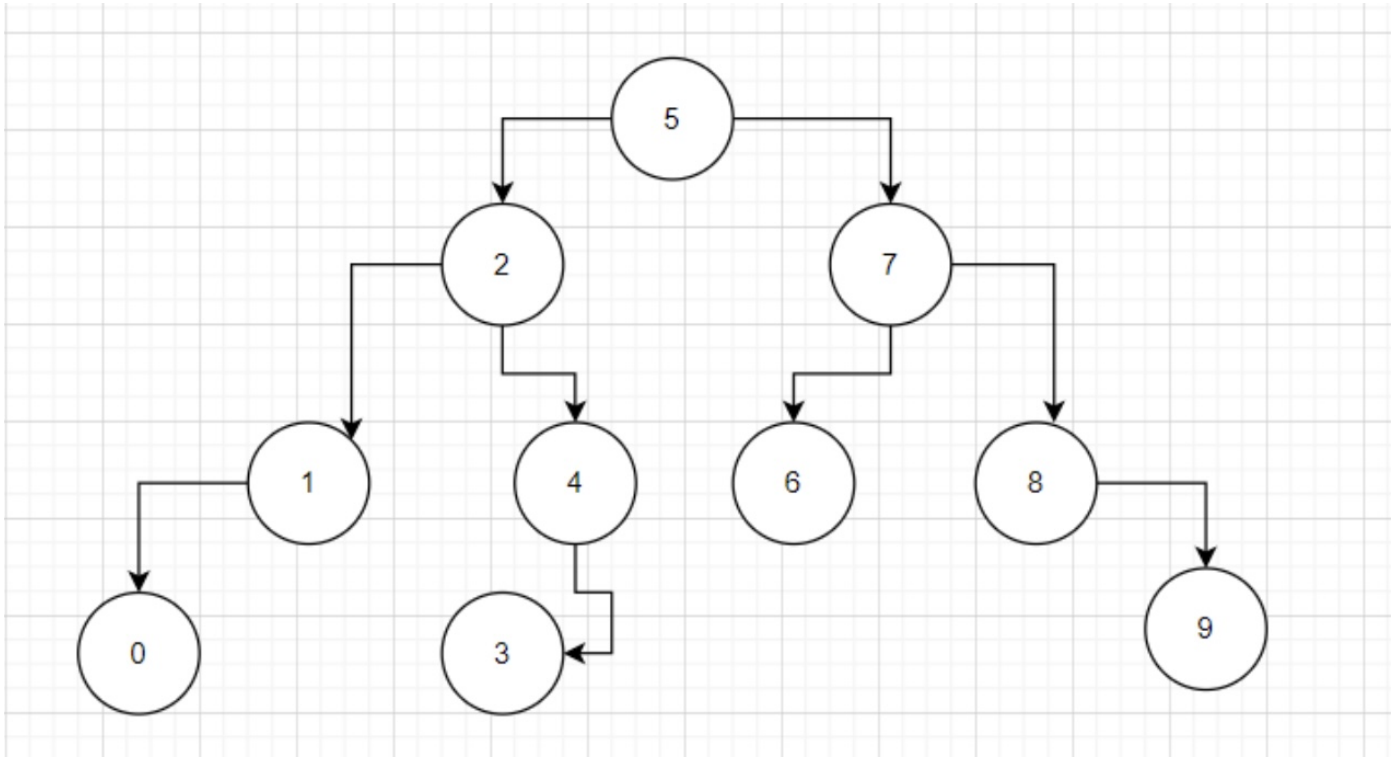
线性查找

线性存储数据显然是不行的，运气不好每次都要追溯到最后一个数据块才能查询到。比如查找数据0

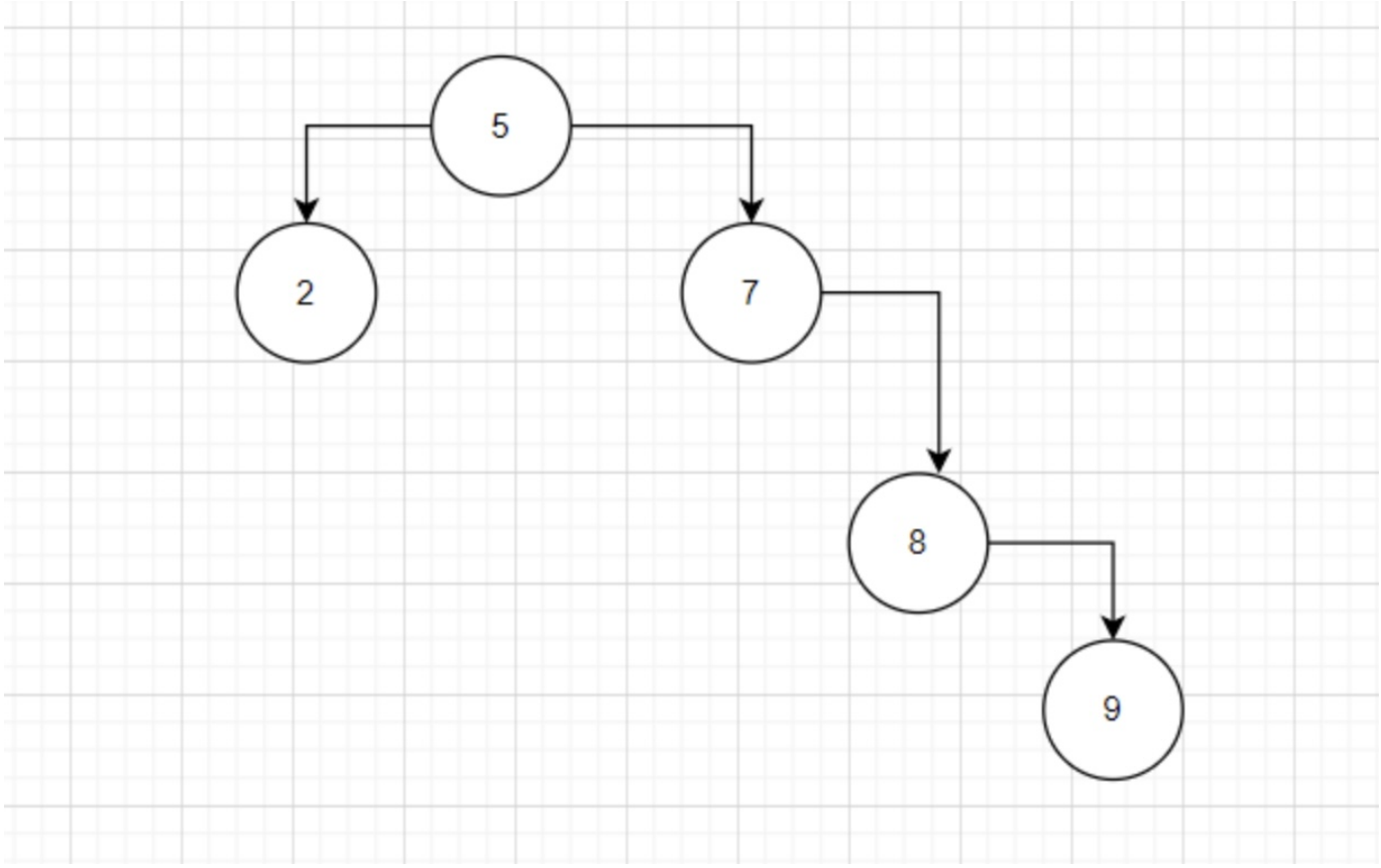


二叉树查找

我们都学过二分查找，如果我们一开始在插入数据就将数据按顺序排列，那查找效率就会大大提高。将数据按照顺序二叉树排列好每次查询就可以使用二分法即从根节点出发，查询效率就增加了。

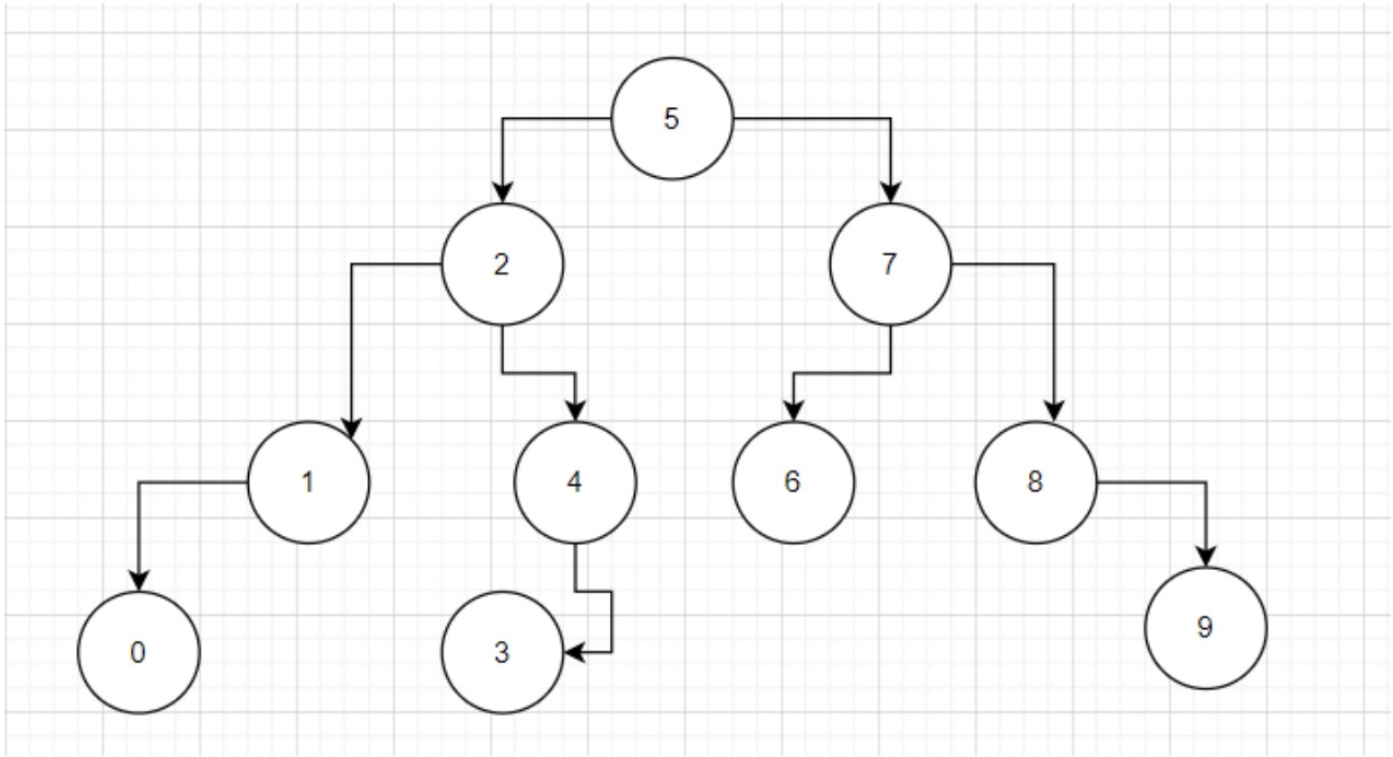


但是如果删除了一些数据比如0，6，1，3，4删除了那么右边的数据就有变成了线性的了，查询效率就会有所浪费，



平衡二叉树

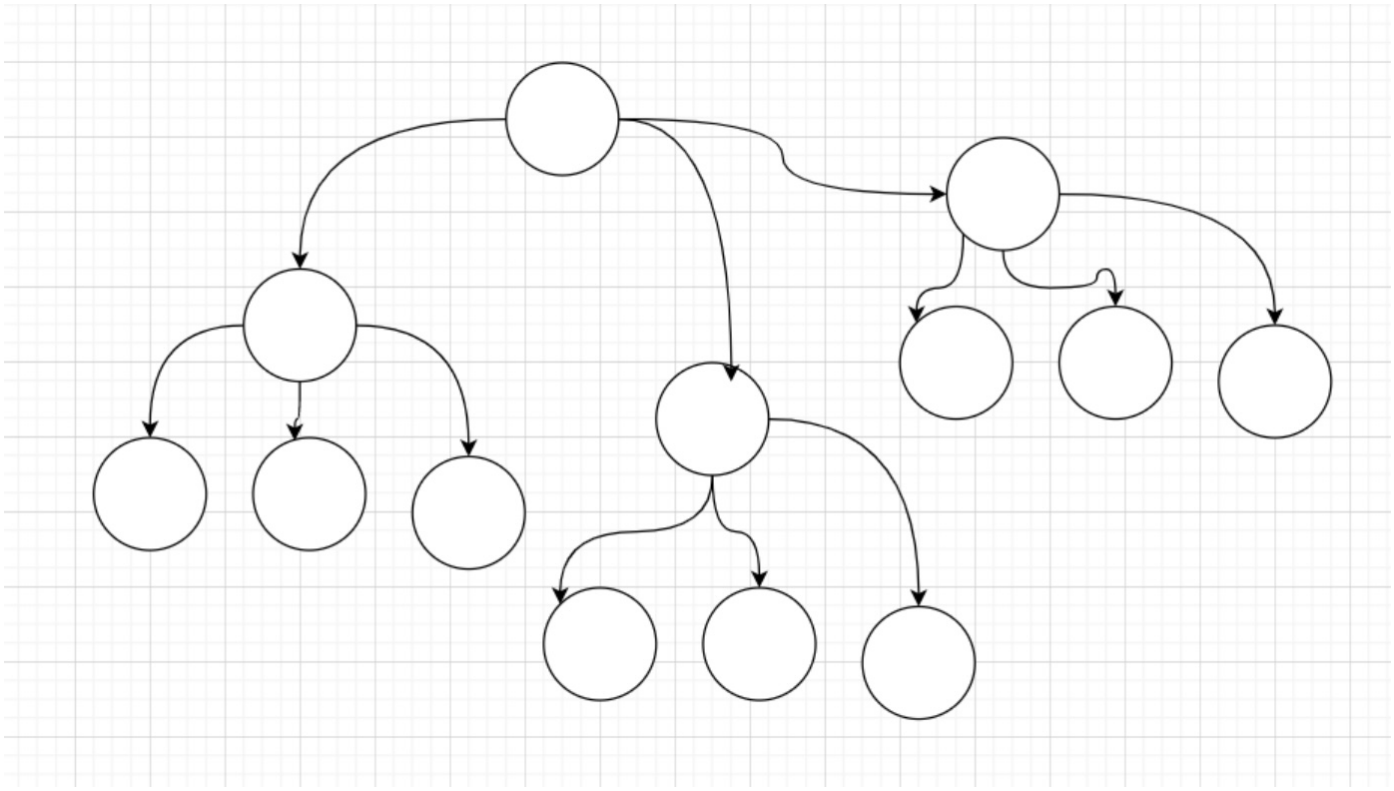
如果有这样一棵树左右子树高度差不超过 1，而且也满足顺序排列就可以继续高效下去，然后平衡二叉树就出现了。每当删除一个节点都应该发生相应的节点位置转换反转保证二叉树的平衡。



说了这么多我们无非就是想让查找效率降低，从线性的 O（n） 到 O (logn），好像还有疑问为什么不用更高效的 Hash 地址法来查找呢？这样可以降到 O（1），答案是在查询过程中我们不仅有等值查询还有范围查询 模糊查询，使用 Hash 存储其位置的不确定性，如果要查询 范围我们就要遍历全表。而二叉树只要遍历左右节点。

B树

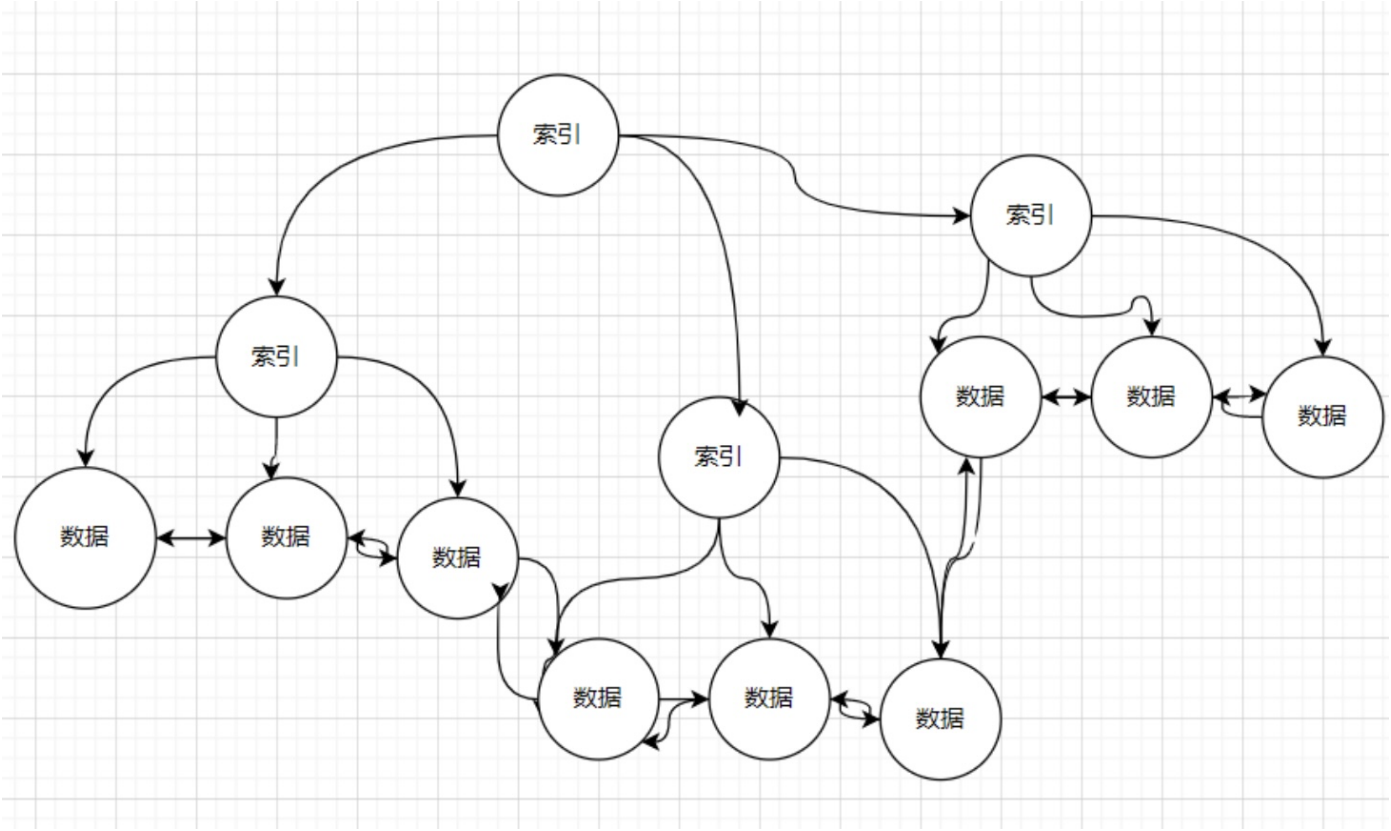
由于平衡二叉树的二叉特点，它每一个节点最多只有 2 个叉，假设有 100000 个数据，那么树的深度将会变得特别深，而每次比较就是拿比较的树和节点上的数在内存比较，所以每比较一次就是一次 IO 操作就下降一层，层数越多时间就越久。所以B树就来了，他是多叉平衡树，每个节点维护了多个比较范围（即子节点）



这样就降低了高度，每个圆圈可以理解为一页，16kb的数据. 所以他的每个节点都存储数据就会造成每个结点的分叉数减少，而且会造成先靠近根节点的先查到，靠近叶子结点的后查到。同样范围查找也会出现多次回退到父节点在到另一个兄弟节点的低效率问题。

B+树

我们改造一下 B树 为 B + 树 ，每个非叶子节点只存索引，**真实数据都存在叶子节点**，这样非叶子节点的空间 单个数据空间 减少 数量即分叉就可以增大。每次查询无论如何必须遍历到叶子节点才会结束，这样深度又减少了，同时我们把每个叶子结点用双向链表连接起来，范围查询就更快。



这种存储方式就是B+ 树实现的 聚簇索引 叶子节点索引即数据，数据即索引。

7、MySQL 事务有哪些隔离级别、分别有什么特点，以及 MySQL 的默认隔离级别是什么?

官方解析

MySQL 事务有四种隔离级别：

1. **读未提交**（Read Uncommitted）：事务可以**读取未提交的数据**，可能会读到**脏数据**，会导致**幻读、不可重复读、脏读**等问题；
2. **读已提交**（Read Committed）：**只能读取已经提交的数据**，可以**避免脏读**问题，但是可能会遇到**不可重复读、幻读**问题；
3. **可重复读**（Repeatable Read）：保证**同一个事务中多次读取同一数据的结果是一致的**，避免了**脏读和不可重复读**问题，但是可能会遇到**幻读**问题；
4. **序列化**（Serializable）：**最高的隔离级别**，可以避免所有并发问题，但是并发性能非常低，开销很大。

MySQL 的默认隔离级别是**可重复读**（Repeatable Read）。

其中，**脏读指一个事务读到了另一个事务未提交的数据，不可重复读指同一个事务多次读取同一数据得到不同结果，幻读指同一个事务前后读取的数据集合不一致。**

在实际使用中，应该根据具体情况选择合适的隔离级别，权衡数据的一致性和并发性能。

> 作者：编程导航知识星球 + 星球鱼友们

鱼友的精彩回答

苏打饼干的回答

隔离级别	特点	脏读	不可重复读	幻读	更新丢失
读未提交	允许读取未提交事务中更新的数据	√	√	√	×
读已提交	只能读取已提交的数据	×	√	√	×
可重复读	保证同一个事务中多次读取同一数据的结果是一致的	×	×	√	×
可串行读	完全串行化的执行，可能导致大量超时和锁竞争	×	×	×	×

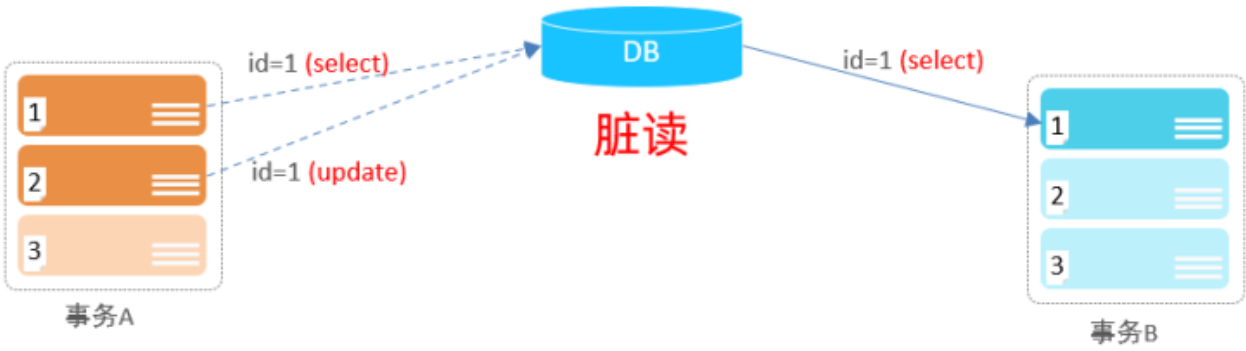
查看隔离级别

```
SELECT @@TRANSACTION_ISOLATION;
```

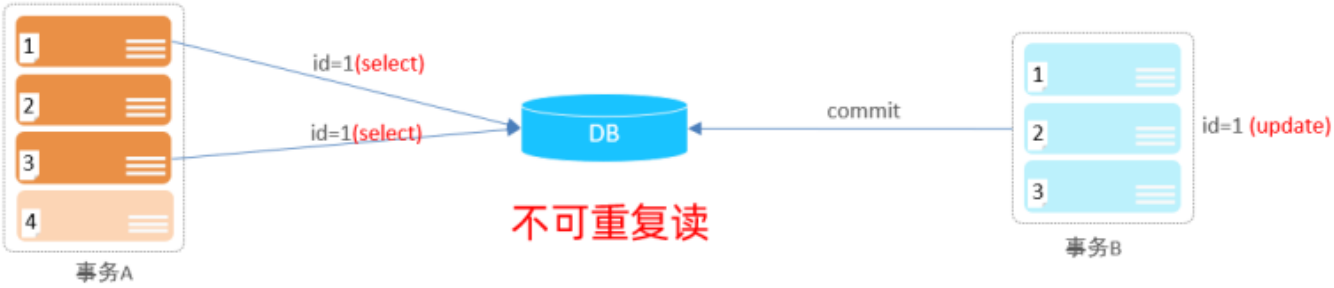
```
SET [SESSION|GLOBAL] TRANSACTION ISOLATION LEVEL [READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE];
```

注：mysql 中默认事物隔离级别是可重复读。

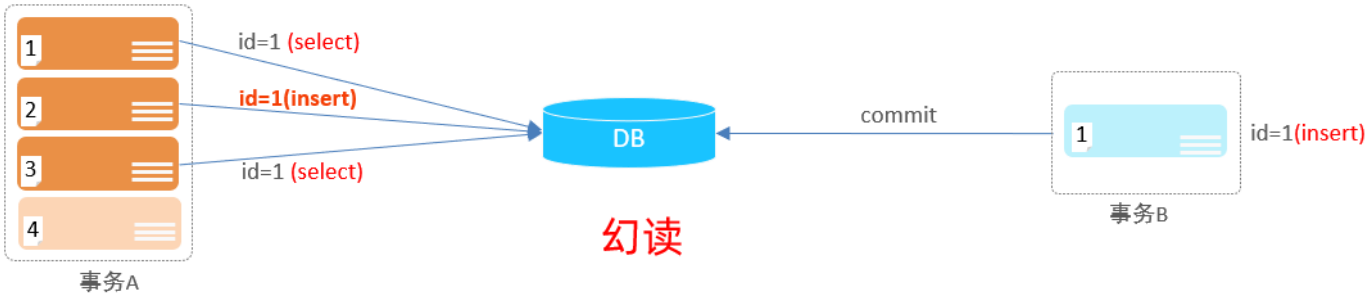
引发问题	描述
脏读	一个事务**读取到**另外一个事务**还没提交的数据**
不可重复读	一个事务先后**读取同一条记录**，但两次读取的**数据不同**
幻读	要给事务按照条件**查询数据时，没有对应的数据行**， 但是在**插入数据时，又发现这行数据已经存在**，如同"幻影"
丢失更新	当两个或多个事务选择同一行，最初的事务修改的值，会被**后提交**的事务修改的值**覆盖**



比如B读取到了A未提交的数据。



事务A两次读取同一条记录，但是读取到的数据却是不一样的。



爱吃鱼蛋的回答

在MySQL中事务的隔离级别是为了解决常见的并发问题，在保证数据库性能的同时保持事务的隔离性，常见的并发问题有：

脏读：如果一个事务读到了另一个未提交事务修改过的数据，那就意味着发生了脏读(Dirty Read)。

发生时间编号	Session A	Session B
①	BEGIN;	
②		BEGIN;
③		UPDATE hero SET name ='关羽' WHERE number = 1;
④	SELECT * FROM hero WHERE number = 1; (如果读到列name的值为'关羽'，则意味着发生了脏读)	
⑤	COMMIT;	
⑥		ROLLBACK;

不可重复读：如果一个事务只能读到另一个已经提交的事务修改过的数据，并且其他事务每对该数据进行一次修改并提交后，该事务都能查询得到最新值，那就意味着发生了不可重复读(Non-Repeatable Read);

发生时间编号	Session A	Session B
①	BEGIN;	
②	SELECT * FROM hero WHERE number = 1; (此时读到的列name的值为'刘备')	
③		UPDATE hero SET name ='关羽' WHERE number = 1;
④	SELECT * FROM hero WHERE number = 1; (如果读到列name的值为'关羽'，则意味着发生了不可重复读)	
⑤		UPDATE hero SET name ='张飞' WHERE number = 1;
⑥	SELECT * FROM hero WHERE number = 1; (如果读到列name的值为'张飞'，则意味着发生了不可重复读)	

幻读：如果一个事务先根据某些条件查询出一些记录，之后另一个事务又向表中插入了符合这些条件的记录，原先的事务再次按照该条件查询时，能把另一个事务插入的记录也读出来，那就意味着发生了幻读(Phantom reading)。

发生时间编号	Session A	Session B
①	BEGIN;	
②	SELECT * FROM hero WHERE number > 0; (此时读到的列name的值为'刘备')	
③		INSERT INTO hero VALUES(2, '曹操', '魏');
④	SELECT * FROM hero WHERE number > 0; (如果读到列name的值为'刘备'、'曹操'的记录，则意味着发生了幻读)	

三个并发问题的区别如下：

- 脏读的重点在于**未提交**。脏读应该是三个里面最好理解的，其定义很轻易便能理解，一个事务中读取了另外一个事务未提交的数据，是先修改再读；
- 不可重复读的重点在于对**单条数据读取了两遍**。T1先读取了一遍，而后T2修改该数据并提交，最后T1再次读取了该数据发现与之前的不同；
- 幻读的重点在于**针对一类条件对一系列数据读取了两遍**。比较特殊的点在于幻读是具备条件的查询，这种查询可能查出来的并不只有一条数据，而在两次查询过程中另外一个事务对查询的结果集中的某条数据进行了变动。

针对于上述的并发问题，在 SQL 标准中设立了以下4个隔离级别：

- READ UNCOMMITTED：未提交读。所有事务都可以看到其他未提交事务的执行结果；
- READ COMMITTED：已提交读。一个事务只能看见已经提交事务所做的改变；
- REPEATABLE READ：可重复读。确保了同一事务的多个实例在并发读取数据时，会看到同样的数据行；
- SERIALIZABLE：可串行化。强制事务串行，并发效率很低。

下面表格展示了在 SQL 标准中规定的并发事务执行过程中可能发生的现象，其中✔代表可能发生现象，❌代表不可能发生现象：

隔离级别	脏读	不可重复读	幻读	性能
READ UNCOMMITTED	✔	✔	✔	最好
READ COMMITTED	❌	✔	✔	较好
REPEATABLE READ	❌	❌	✔	一般
SERIALIZABLE	❌	❌	❌	最差

不同的数据库厂商对 SQL 标准中规定的 4 种隔离级别的支持是不一样的。其中 MySQL 的默认隔离级别为 REPEATABLE READ，即可重复读在该隔离级别下可以**很大程度**上禁止了幻读现象的发生。

林风的回答

SQL 标准提出了四种隔离级别来规避这些现象，隔离级别越高，性能效率就越低，这四个隔离级别如下：

- **读未提交**（read uncommitted），指一个事务还没提交时，它做的变更就能被其他事务看到；
- **读已提交**（read committed），指一个事务提交之后，它做的变更才能被其他事务看到；
- **可重复读**（repeatable read），指一个事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的，MySQL InnoDB 引擎的默认隔离级别；

- **串行化**（serializable ）；**会对记录加上读写锁**，在多个事务对这条记录进行读写操作时，如果发生了读写冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行；

拓展问题

- 不同的隔离级别会有不同的特性，那么为什么要设置这么多的隔离级别呢?
- 不同的隔离级别有哪些实际应用的场景，适合解决什么业务?
- 每个隔离级别是怎么实现的?

不同的隔离界别有哪些实际应用的场景，适合解决什么业务? 不同的隔离级别适用于不同的场景，下面列举一些实际应用：

- 读未提交（read uncommitted）：适用于数据**实时性要求较高**的场景，比如一些报表的生成，当数据量大、计算量大时，如果使用较高的隔离级别，可能会出现锁等待导致查询时间过长的情况，而读未提交隔离级别可以避免这种情况的发生。
- **读已提交**（read committed）：适用于对**数据准确性要求较高**的场景，比如金融交易等，由于该隔离级别会在读取时加共享锁，所以可以避免脏读的问题。
- **可重复读**（repeatable read）：适用于业务场景较为复杂的情况，比如**订单下单和支付**，如果使用较低的隔离级别可能会导致支付时查询出的订单状态不正确，从而造成数据不一致的问题，而可重复读隔离级别可以避免这种情况的发生。
- **串行化**（serializable）：适用于对**数据完整性要求非常高**的场景，比如一些高风险的金融交易，由于该隔离级别会对每个事务加排它锁，所以可以避免幻读和不可重复读的问题。不过需要注意的是，该隔离级别的并发性非常低，会影响系统的性能，一般只在必要的情况下才会使用。

8、意向锁是什么？有什么作用？它是表级锁还是行级锁?

官方解析

意向锁是一种 MySQL 数据库中的锁，用于表级锁协调多个行级锁的使用。在表级锁定一个表之前，MySQL 需要先获得一个意向锁，以表明要获取的锁类型（读锁或写锁），避免其他事务锁定整个表或锁定一部分表时引发死锁。

意向锁是一种轻量级锁，它不会影响其他事务的读操作，只有在某个事务要对表进行写操作时才会加上意向锁，而其他事务在读取表时只需要获取读锁，不需要等待意向锁的释放。

意向锁可以提高数据库并发性能，防止死锁的发生。它是**表级锁**，而不是行级锁。

鱼友的精彩回答

Ming 的回答

意向锁是什么？有什么作用？它是表级锁还是行级锁?

意向锁是什么

在使用 InnoDB 引擎的表里时对某些记录加上「**共享锁**」之前，需要先在表级别加上一个「**意向共享锁**」

在使用 InnoDB 引擎的表里时对某些记录加上「**独占锁**」之前，需要先在表级别加上一个「**意向独占锁**」

也就是，当执行插入、更新、删除操作，需要先对表加上「意向独占锁」，然后对该记录加独占锁。

意向锁的作用

意向共享锁和意向独占锁是**表级锁**，不会和行级的共享锁和独占锁发生冲突，而且意向锁之间也不会发生冲突，只会和共享表锁（lock tables ... read）和独占表锁（lock tables ... write）发生冲突。

表锁和行锁是满足**读读共享、读写互斥、写写互斥的**。

作用：**为了快速判断表里是否有记录被加锁**

如果没有「意向锁」，那么加「独占表锁」时，就需要**遍历表里所有记录**，查看是否有记录存在独占锁，这样效率会很慢。

那么有了「意向锁」，由于在对记录加独占锁前，先会加上表级别的意向独占锁，那么在加「独占表锁」时，直接查该表是否有意向独占锁，如果有就意味着表里已经有记录被加了独占锁，这样就不用去遍历表里的记录。

意向锁是表级锁还是行级锁?

表级别锁有以下几种

- 表锁
- 元数据锁（MDL）
- 意向锁
- AUTO-INC锁

所以意向锁是表级别锁

其他

普通的 select 是不会加行级锁的，普通的 select 语句是利用 MVCC 实现一致性读，是无锁的。

不过，select 也是可以对记录加共享锁和独占锁的，具体方式如下：

```
//先在表上加上意向共享锁，然后对读取的记录加共享锁
select ... lock in share mode;

//先表上加上意向独占锁，然后对读取的记录加独占锁
select ... for update;
```

爱吃鱼蛋的回答

MySQL的意向锁(Intention Lock)是一种表级锁，属于辅助锁的一种，主要用于帮助事务在请求表级锁之前了解其他事务对同一个表的锁定情况，以便更好的协调锁定请求。

MySQL 的意向锁分为两种类型：意向共享锁（IS）和意向排他锁（IX）。当一个事务要请求某个表或某些行的锁时，它需要先获取适当类型的意向锁。如果表已经被其他事务锁定，则事务需要等待该锁释放才能获取意向锁，从而避免了死锁的发生。

意向锁的作用主要是协调事务对表的锁定，它本身并不会对表产生实际的锁定。意向共享锁表示事务将要对表或行进行读取操作，意向排他锁表示事务将要对表或行进行修改操作。在实际的锁定操作中，事务可以请求适当类型的行级锁，也可以请求表级锁。如果事务请求的是行级锁，则不需要先获取意向锁。

在实际开发中，假设有两个事务 T1 和 T2，它们同时要访问同一个表 T 中的某些行。如果 T1 先请求了表 T 的意向共享锁（IS），表示 T1 希望对 T 进行读操作，T2 再请求表 T 的意向锁时就会发现已经被锁定了，从而在请求锁的过程中进行等待，避免了 T1 和 T2 在请求锁的过程中产生死锁的情况。同样地，如果 T1 先请求了表 T 的意向排他锁（IX），表示 T1 希望对 T 进行写操作，那么 T2 在请求锁时也会被阻塞，直到 T1 的锁被释放。

需要注意的是，虽然意向锁是表级锁，但它并不会对表进行实际的锁定，而是用于协调事务对表的锁定。在实际的锁定操作中，事务可以请求适当类型的行级锁，也可以请求表级锁。因此，在开发中，我们需要根据具体情况选择适当的锁定方式，避免在并发访问时产生冲突和死锁。

9、MVCC 是什么？InnoDB 是如何实现 MVCC 机制的?

官方解析

MVCC 是指**多版本并发控制**（Multiversion Concurrency Control），是一种**并发控制机制**，常用于数据库系统中，用于实现**事务的并发控制**。它允许在**同一时间多个事务对同一个数据集合进行读取操作**，同时防止数据不一致和其他并发问题。

InnoDB 是 MySQL 中最常用的存储引擎之一，它的 MVCC 实现是**通过在每行记录中添加两个隐藏的列，分别记录行的创建时间和过期时间，以此来判断事务对该行记录的可见性**。当一个事务需要读取一行记录时，InnoDB 首先读取这行记录的创建时间和过期时间，并根据这些信息判断该行记录是否可见。如果创建时间早于当前事务的开始时间，且过期时间晚于当前事务的开始时间，那么该行记录对当前事务可见。

在 InnoDB 中，MVCC 主要是通过实现以下几个机制来实现的：

1. **事务版本号**：每个事务都有一个唯一的版本号，用来标识该事务的创建时间。
2. **读取视图**：每个事务在开始时都会创建一个读取视图，记录该事务开始时间和其他信息。在事务执行期间，所有读取操作都要检查该视图，以确定读取哪些版本的数据。
3. undo 日志：在事务执行期间，如果对数据进行修改，那么会先将原始数据复制一份到 undo 日志中。这样，在回滚操作时就可以使用 undo 日志中的数据来还原原始数据。
4. 快照读取：在某些情况下，事务需要读取一个数据的历史版本，而不是当前版本。这时可以使用快照读取来实现，即在读取时根据事务开始时间和 undo 日志来读取历史版本的数据。

鱼友的精彩回答

Gundam 的回答

MVCC（Multi-Version Concurrency Control）是一种**并发控制机制**，它**通过为每个读操作创建一个视图来实现读写分离，保证了多个事务同时读写同一个数据时的一致性和并发性**。

InnoDB 是 MySQL 数据库中的一种**存储引擎**，它通过使用 MVCC 实现了**高度并发的事务处理**。

在 InnoDB 中，每个事务读取的数据都会生成一个**当前版本号**，并将这个版本号与事务 ID 一起存储在数据行中。

在事务执行期间，如果某个事务想要修改一行数据，InnoDB 会首先为这个事务生成一个新的版本号，并将新的版本号和事务 ID 存储在新版本的数据行中。由于每个事务都有自己的版本号，因此不会出现读写冲突的情况，多个事务可以同时读取同一行数据，并发执行而不会相互影响。

为了保证 MVCC 机制的正确性，InnoDB 需要使用**多版本链表**（Multi-Version List）来存储数据。**每个数据行都会有一个多版本链表，其中存储着该行数据的所有历史版本**。当一个事务执行查询操作时，InnoDB 会根据该事务的 ID 和时间戳来确定该事务能够读取到的版本号，然后将这个版本号对应的数据返回给该事务。当一个事务执行插入、删除或更新操作时，InnoDB 会为这个操作生成一个新的版本，并将新版本的数据插入到多版本链表中。

在 **InnoDB** 中，为了实现 MVCC，还需要使用 **undo log 和 read view 来存储数据**。

undo log 记录了每个事务对数据行进行修改的情况，read view 记录了每个事务开始时的数据版本号和当前系统时间，用于判断事务是否可以读取某个数据行的版本。当一个事务执行查询操作时，InnoDB 会根据 read view 来确定该事务能够读取到的数据版本号，并使用 undo log 来恢复该数据行的历史版本。

haha 的回答

MVCC

MVCC 指的是**多版本并发控制**，是**指维护一条记录的多个版本，使得读写操作没有冲突**。

MVCC的实现

InnoDB 对于 MVCC 的实现，我主要从下面几个点来讲：

1. **当前读**和**快照读**的概念
2. MySQL 数据的**隐藏字段**
3. **undo 日志中的版本链**
4. **readView（读视图）**
5. MVCC实现流程

当前读与快照读

当前读：就是读取的记录总是最新的，实现的原理就是对正在读的记录加锁，使得读写互斥，这样保证每次读取的都是数据库中最新的记录

快照读：每次读取的时候不一定是最新的数据，而是这条记录的**快照版本**，这样可以**保证读写不互斥，能够并发执行**

MySQL **的隐藏字段**

MVCC 的实现主要依赖于：MySQL 的隐藏字段、undo log 中的版本链、readView

MySQL 中每条记录是有两个隐藏的字段，分别是：

- **最近修改事务 ID**：记录着上一次修改该条记录的日志 id
- **回滚指针**：指向上一个版本的记录的地址

另外，如果表中没有指定主键，那么还有有一个隐藏的主键

undo log 版本链

版本链顾名思义就是记录的版本的链表。

当事务并发执行修改某条记录的时候，不同的事物对这条数据的修改产生多个版本，每次修改之前都会记录下这条记录之前的数据，在隐藏字段中设置上本次操作事务的ID，并让回滚指针指向上一个版本，这样就会形成一条链表，就是所谓的版本链。

readView 读视图

readView 读视图记录并维护了当前系统活跃的事务 ID,为快照读时 MVCC 提供数据的依据。其主要有以下几个属性：

1. 记录当前活跃事务的 ID 集合
2. 最小活跃事务 ID
3. 预分配事务 ID，就是最大活跃事务 ID + 1（因为事务 ID 是自增的）

- readView 创建者的事务 ID

MVCC执行的流程

当并发事务执行的时候，执行查询操作的时候，会根据这个事务的 ID 和 readView 中的事务 ID 进行一些比较来确定读取哪个版本的快照记录，具体的规则为：

- 记录中的最近修改事务的 ID 小于读视图中的最小事务 ID，说明记录的修改已经提交了，可以访问
- 事务 ID 等于读视图中创建者 ID，说明这条记录就是当前事务修改的，可以访问
- 事务 ID 大于等于读视图中预分配事务 ID，说明这条记录是在读视图创建之后修改的，不可访问当前版本，去访问下一个版本吧
- 如果事务 ID 在最小事务 ID 和最大 ID 之间且不在 ID 集合中，说明当前版本的修改事务已经提交，可以访问

RC 和 RR 的区别

RC 级别时，在一个事务中，每执行一次查询都会生成一次读视图。

RR 级别时，在一个事务中，只有第一次查询会生成一个读视图，后面的查询都是复用这个读视图，保证了可重复读

回家养猪的回答

什么是 MVCC

MVCC 全称 Multi-Version Concurrency Control, **多版本并发控制**。指**维护一个数据的多个版本，使得读写操作没有冲突**, 具体实现就是**快照读**, 快照读为 MySQL 实现 MVCC 提供了一个非阻塞读功能。MVCC 的具体实现，还需要依赖于数据库记录中的**隐式字段、undo log 日志、readView**。

MVCC 的实现原理

MVCC 的具体实现，依赖于数据库记录中的隐式字段、undo log 日志、readView。

在内部实现中，InnoDB 通过数据行的 DB_TRX_ID(**最近更新的事务id**) 和 **Read View** 来判断数据的可见性，如不可见，则通过数据行的 DB_ROLL_PTR(**回滚指针**) 找到 **undo log 版本链中的历史版本**。这就是**快照读**

每个事务读到的数据版本可能是不一样的，**在同一个事务中，用户只能看到该事务创建 Read View 之前已经提交的修改和该事务本身做的修改**

MVCC 是怎么解决不可重复读的

在 RC 读已提交下, 在事务中每一次执行快照读时生成 ReadView, **这也就造成了每次读取就有不同 ReadView**, 那么就会**读到已提交的事务修改的内容**, 造成**不可重复读**的问题.

解决 RR 不可重复读主要靠 readview, 在隔离级别为可重复读时, 仅在事务中第一次执行快照读时生成 ReadView, 后续复用该 ReadView.

由于后续**复用了 ReadView**, **所以数据对当前事务的可见性和第一次是一样的**, 所以从 undolog 中读到的数据快照和第一次是一样的, 即便过程中有其他事务修改也读不到.

MVCC是怎么防止幻读的

InnoDB 存储引擎在 **RR** 级别下通过 **MVCC** 和 **Next-key Lock(临键锁)** 来解决**幻读**问题

1、执行**普通 select**，此时会以 **MVCC 快照读**的方式读取数据

> 快照读: 避免加锁通过 MVCC 来进行控制, 使其他事务所做的更新对当前事务不可见, 从而防止幻读.

在快照读的情况下，RR 隔离级别只会在事务开启后的第一次查询生成 Read View ，并使用至事务提交。所以在生成 Read View 之后其它事务所做的更新、插入记录版本对当前事务并不可见，实现了可重复读和防止快照读下的 “幻读”

2、执行 select...for update/lock in share mode、insert、update、delete 等当前读

> 当前读: 通过临键锁 next-key-lock 锁住空隙, 防止其他事务在查询的范围内插入数据, 从而防止幻读.

在当前读下，读取的都是最新的数据，如果其它事务有插入新的记录，并且刚好在当前事务查询范围内，就会产生幻读！InnoDB 使用 **Next-key Lock 临键锁**来防止这种情况。当执行当前读时，会锁定读取到的记录的同时，锁定它们的间隙，防止其它事务在查询范围内插入数据。**只要我不让你插入，就不会发生幻读**

10、覆盖索引和联合索引是什么？讲一下索引的最左前缀匹配原则。

官方解析

覆盖索引和**联合索引**是数据库中常见的两种索引类型。

覆盖索引是指一个**包含了所有查询需要的列的索引**，查询时可以直接从索引中取到需要的数据，而**不需要再回到表中查找**，从而可以提高查询效率。

联合索引是指**使用多个列组合起来作为一个索引**，可以**同时查询多个列**，以提高查询效率。联合索引可以包含多个列，但是查询时只能使用**前缀列**进行查询，即只有在查询中使用了联合索引的前几个列，才能利用联合索引进行查询。如果查询中没有使用**前缀列**，那么**联合索引就不能发挥作用**，需要使用单独的索引或全表扫描。

最左前缀匹配原则是指**如果一个联合索引包含了多个列，那么在查询时只能使用前面的列进行匹配**。

例如，一个联合索引包含了 **A**、**B**、**C** 三列，那么查询时只能使用 **A**、**AB 或 ABC** 进行匹配，而不能只使用 B 或 C 进行匹配。这是因为如果查询时使用的列不是**最左前缀列**，那么 MySQL 就无法使用索引进行查询，会导致全表扫描，从而降低查询效率。

在实际的应用中，**覆盖索引**和**联合索引**可以结合使用，以提高查询效率。同时，使用**最左前缀匹配原则**可以让我们更加合理地设计索引，从而提高查询性能。

鱼友的精彩回答

维萨斯的回答

两个sql回答本题

> 联合索引 = 聚合索引 = 组合索引

```
CREATE TABLE test
(
    id         INT,
    age        INT,
    ... (以下省略敲多字段)
    INDEX id_age_index (id, age) // 这个就是联合索引
);
```

覆盖索引 (索引覆盖结果)
注意是覆盖!!! index如果是 id_age_index 也是覆盖

```
SELECT
id ⬅-------------
FROM test        | 前提: 给id建了索引!!!
WHERE            | 前提: 给id建了索引!!!
id > 1; ⬅--------
```

最左前缀匹配

以最左边的为起点任何连续的索引都能匹配上。同时遇到范围查询(>、<、between、like)就会停止匹配。

```
上面的test表
有个(id, age) 这个索引
在查询优化器中,索引要先走 id,再走 age,优先匹配左,并且必须匹配上,才会继续匹配 age
```

一句话: 脑中模拟B+树 答得HR直发怵

关于索引实现可以看一下往期面试题挑战文章

Day10 MySQL 中的索引是怎么实现的？B+ 树是什么

Ming 的回答

覆盖索引和联合索引是什么？讲一下索引的最左前缀匹配原则

覆盖索引

是指 SQL 中 query 的所有字段，在索引 B+ Tree 的叶子节点上都能找得到的那些索引，从二级索引中查询得到记录，而不需要通过**聚簇索引**查询获得，可以**避免回表**的操作

联合索引

通过将**多个字段组合成一个索引**，该索引就被称为联合索引。

最左前缀匹配原则

使用**联合索引**时，存在**最左匹配原则**，也就是按照**最左优先**的方式进行索引的匹配。在使用联合索引进行查询的时候，如果不遵循「**最左匹配原则**」，**联合索引会失效**，这样就无法利用到索引快速查询的特性了。

有效的例子

比如，如果创建了一个 (a, b, c) 联合索引，如果查询条件是以下这几种，就可以匹配上联合索引：

```
where a=1； where a=1 and b=2 and c=3； where a=1 and b=2；
```

注意，因为有**查询优化器**，所以 a 字段在 where 子句的顺序并不重要。即 where b=2 and a=1； 也是符合最左前缀的规则

失效的例子

```
where b=2； where c=3； where b=2 and c=3；
```

因为(a, b, c) 联合索引，是**先按 a 排序，在 a 相同的情况再按 b 排序，在 b 相同的情况再按 c 排序**。所以，b 和 c 是全局无序，局部相对有序的，这样在没有遵循**最左匹配原则**的情况下，是无法利用到索引的。

范围查询例子

Q1

```
select * from t_table where a > 1 and b = 2
```

只有 a 字段用到了联合索引进行索引查询，而 b 字段并没有使用到联合索引

Q2

```
select * from t_table where a >= 1 and b = 2
```

a 和 b 字段都用到了联合索引进行索引查询。

Q3

```
SELECT * FROM t_table WHERE a BETWEEN 2 AND 8 AND b = 2
```

a 和 b 字段都用到了联合索引进行索引查询

Q4

```
SELECT * FROM t_user WHERE name like 'j%' and age = 22
```

a 和 b 字段都用到了联合索引进行索引查询

> 注意：结合 explain 分析

总结

综上所示，**联合索引的最左匹配原则**，在遇到**范围查询**（如 **>**、**<**）的时候，就会停止匹配，也就是范围查询的字段可以用到联合索引，但是在范围查询字段的后面的字段无法用到联合索引。注意，对于 **>=**、**<=**、**BETWEEN**、**like** 前缀匹配的范围查询，并不会停止匹配，前面我也用了四个例子说明了。

11、什么是 MySQL 执行计划？如何获取执行计划并对其进行分析？

官方解析

MySQL 执行计划是指 **MySQL 查询优化器**生成的一份详细的**查询执行计划**，它展示了 MySQL 在执行查询时所采取的具体执行计划，包括**表的访问顺序**、**数据读取方式**、**使用的索引**、**使用的排序方式**等等。通过分析执行计划，可以帮助我们找出查询性能瓶颈所在，进而进行优化，提高查询效率。

要获取执行计划，可以在执行 SQL 语句时在前面添加 explain 关键字，例如：

```
explain select * from table where id = 1;
```

这样，MySQL 会输出该查询语句的执行计划。 执行计划中的各个字段含义如下：

- id：每个 Select 子句或者是一个操作符或者是一个查询语句。
- select_type：查询类型，表示查询的类型（简单查询、联合查询、子查询等等）。
- table：查询涉及的表。
- partitions：匹配的分区。
- type：访问类型，表示 MySQL 在表中找到所需行的方式。
- possible_keys：表示查询可能使用到的索引。
- key：实际使用到的索引。
- key_len：使用的索引长度。
- ref：列与索引的比较。
- rows：根据表统计信息及索引选用情况，大致估算出找到所需的记录所需要读取的行数。
- filtered：返回结果的行数占总行数的比例。
- Extra：包含 MySQL 解决查询的详细信息。

分析执行计划时，需要注意以下几个方面：

- **扫描行数**：rows 字段，表示查询所需扫描的行数，如果该值过大，说明查询效率不高，需要优化。
- **使用索引**：key 字段，表示查询所使用的索引，如果没有使用索引或者使用的不是最优索引，需要考虑优化。
- **排序**：Extra 字段，如果查询需要使用 filesort 排序，说明查询效率不高，需要优化。
- **嵌套循环**：如果查询类型是 nested loop，说明查询中包含嵌套循环，也需要考虑优化。

通过分析执行计划，可以确定查询优化的方向和方法，提高查询效率。

> 鱼皮补充：这里大家不用去记忆执行计划中各字段的含义，你可以给面试官举个例子，你是怎么使用查询计划来分析定位一次慢查询的。

鱼友的精彩回答

Gianhing 的回答

MySQL 执行计划是 MySQL 查询优化器分析 SQL 查询时生成的一份详细计划，包括**表如何连接、是否走索引、表扫描行数等**。通过这份执行计划，我们可以**分析这条 SQL 查询中存在的问题**（如是否出现全表扫描），从而进行针对优化。

MySQL 中可以在 SQL 语句的前面加上 **explain** 命令来获取执行计划，例如：

```
mysql> explain select * from t_user where id = 1;
+----+-------------+--------+------------+-------+---------------+---------+---------+-
------+------+----------+-------+
| id | select_type | table  | partitions | type  | possible_keys | key     | key_len |
ref   | rows | filtered | Extra |
+----+-------------+--------+------------+-------+---------------+---------+---------+-
------+------+----------+-------+
|  1 | SIMPLE      | t_user | NULL       | const | PRIMARY       | PRIMARY | 4       |
const |    1 |   100.00 | NULL  |
+----+-------------+--------+------------+-------+---------------+---------+---------+-
------+------+----------+-------+
```

其中，各字段的含义如下：

- id：SELECT 查询的序列号，表示执行 select 子句的顺序（id 相同，执行顺序从上到下；id 不同，值越大越先执行；如果是并集结果，则 id 值为 NULL）
- select_type：查询类型，来区分简单查询、联合查询、子查询等。常见的类型有：
  - SIMPLE：简单查询，不包含表连接或子查询
  - PRIMARY：主查询，外层的查询
  - SUBQUERY：子查询中第一个 SELECT
  - UNION：UNION 后面的 SELECT 查询语句
  - UNION RESULT：UNION 合并的结果
- table：查询用到的表名
- partitions：匹配的分区，没有分区的表为 NULL
- type ★：连接的类型，常见的类型有（性能从好到差）：
  - system：存储引擎能够直接知道表的行数（如 MyISAM）并且只有一行数据
  - const：通过索引一次找到，通常在用主键或唯一索引时出现
  - eq_ref：用主键或唯一索引字段作为连接表条件
  - ref：用普通索引的字段作为连接表条件
  - range：对索引列进行范围查询
  - index：利用索引扫描全表
  - all：全表扫描
- possible_keys：可能用到的索引

- key ★：实际使用的索引，没有使用为 NULL
- key_len：索引字段的最大长度，在满足需求下越短越好
- ref：当使用索引等值查询时，与索引作比较的列或常量
- rows ★：预计扫描的行数，值越大，查询性能越差
- filtered：表示返回结果的行数占需读取行数的百分比
- Extra ★：有关查询执行的其他信息
    - using index：使用覆盖索引，不用回表查询
    - using where：使用 where 子句来过滤结果集
    - using temporary：使用到临时表来存储中间结果，可能会导致性能问题
    - using filesort：查询需要进行文件排序操作，可能会导致性能问题
    - using index condition：先根据能用索引的条件获取符合条件的数据行，然后在根据其他条件去过滤数据

> 带 ★ 是需要重点关注的字段

12、MySQL 支持哪些存储引擎？默认使用哪个？MyISAM 和 InnoDB 引擎有什么区别，如何选择?

官方解析

MySQL 支持多种存储引擎，包括 **InnoDB**、**MyISAM**、**MEMORY**、**CSV** 等。默认情况下，MySQL 使用的存储引擎是 **InnoDB**。

MyISAM 和 InnoDB 是 MySQL 中 最常用 的两种存储引擎，它们有以下区别：

1. **锁定方式不同**：MyISAM 使用**表级**锁定，而 InnoDB 使用**行级**锁定。在并发访问时，InnoDB 的锁定方式更加精细，可以避免锁定整个表，提高了并发性能。
2. **数据完整性不同**：MyISAM 不支持事务和外键约束，而 **InnoDB 支持事务和外键约束**，可以保证数据的完整性和一致性。
3. **读写性能不同**：**MyISAM 的读写性能相对较高**，适合于**读密集**型应用；而 **InnoDB 的写性能相对较高**，适合于**写密集型**应用。
4. **空间利用率不同**：**MyISAM 不支持行级别的存储**，存储空间利用率较低；而 **InnoDB 支持行级别的存储**，存储空间利用率更高。

在选择 MyISAM 和 InnoDB 引擎时，需要考虑应用场景和需求：

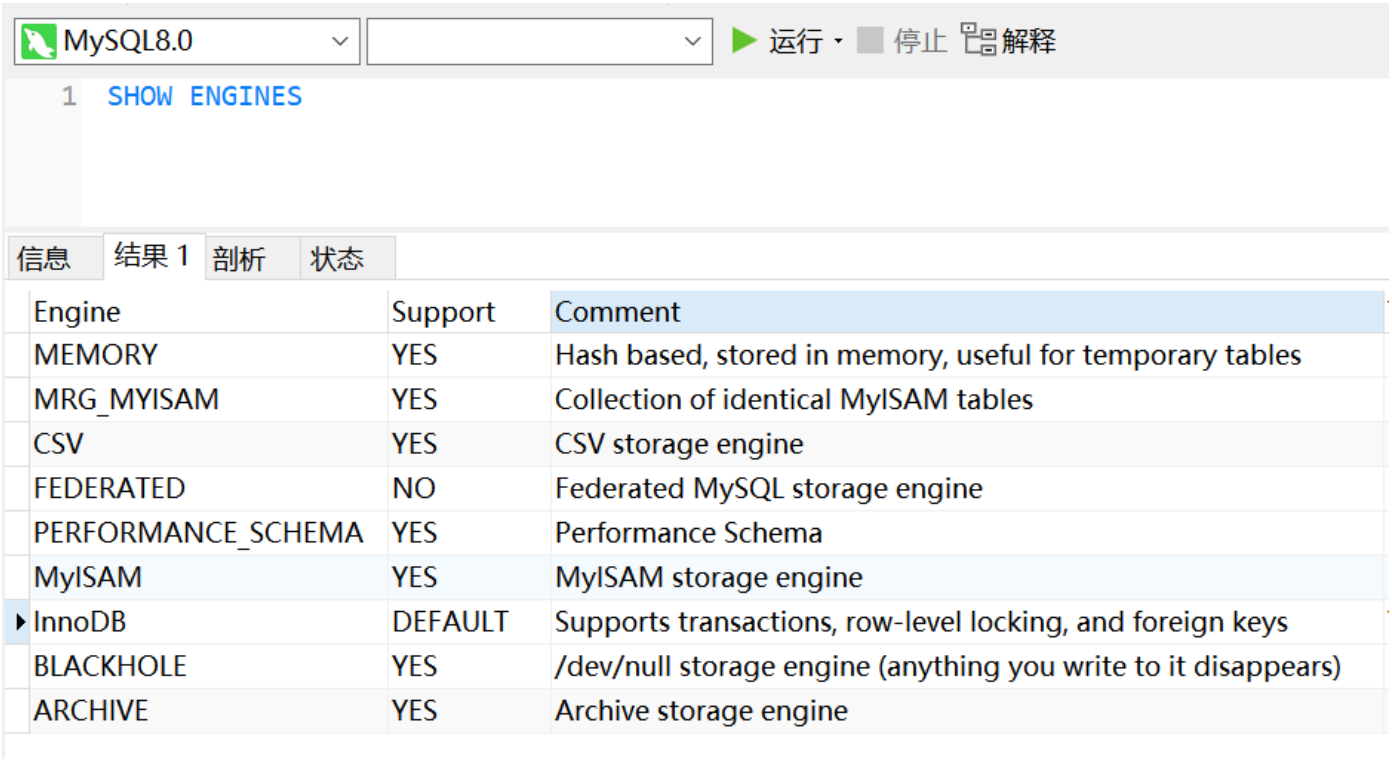
1. 如果应用主要是**读操作**，可以考虑选择 **MyISAM** 引擎，以**提高读取性能**。
2. 如果应用主要是**写操作**或**需要支持事务和外键约束**，可以考虑选择 **InnoDB** 引擎，以保证数据的**完整性**和**一致性**。
3. 如果需要**高性能**和**高可用性**，可以考虑选择使用 **MySQL 集群**或使用**多个副本实例**，并将数据分布在不同的节点上。

总结：MySQL 支持多种存储引擎，**MyISAM 和 InnoDB** 是其中最常用的两种，它们有不同的特点和优缺点，在选择时需要根据应用场景和需求进行考虑和权衡。

鱼友的精彩回答

yes.的回答

看到这个问题，我反手就是去数据库查了一下。



Engine	Support	Comment
MEMORY	YES	Hash based, stored in memory, useful for temporary tables
MRG_MYISAM	YES	Collection of identical MyISAM tables
CSV	YES	CSV storage engine
FEDERATED	NO	Federated MySQL storage engine
PERFORMANCE_SCHEMA	YES	Performance Schema
MyISAM	YES	MyISAM storage engine
InnoDB	DEFAULT	Supports transactions, row-level locking, and foreign keys
BLACKHOLE	YES	/dev/null storage engine (anything you write to it disappears)
ARCHIVE	YES	Archive storage engine

可以看到 Mysql 支持多种搜索引擎，默认为 InnoDB。

其中最常用的是 InnoDB 和 MyISAM，他们有以下区别：

1. InnoDB 支持**事务**操作，而 MyISAM 不支持事务
2. InnoDB 支持**外键**，而 MyISAM 不支持外键
3. InnoDB 支持**行级锁，表级锁**，而 MyISAM 只支持表级锁
4. InnoDB 支持数据库异常崩溃后的安全恢复（**redo log**），而 MyISAM 不支持
5. InnoDB 性能比 MyISAM 更强，不管是在读写混合模式下还是只读模式下，随着 CPU 核数的增加，InnoDB 的读写能力呈线性增长。MyISAM 因为**读写不能并发**，它的处理能力跟核数没关系。

如果你对 B+ 树有较深的自信你还可以答

6. InnoDB 和 MyISAM **实现索引都是使用 B+ 树**，但实现方式不同

如何选择?

- 在**读密集**的情况下，如果你**不需要事务**，也**不需要保证数据库的崩溃回复**，可以选择 MyISAM
- 其他时候大可放心使用 InnoDB

《**MySQL 高性能**》上面有一句话这样写到:

> 不要轻易相信“MyISAM 比 InnoDB 快”之类的经验之谈，这个结论往往不是绝对的。在很多我们已知场景中，**InnoDB 的速度都可以让 MyISAM 望尘莫及**，尤其是用到了**聚簇索引**，或者需要访问的数据都可以放入内存的应用。

航仔丶的回答

MySQL 支持多种存储引擎，每种存储引擎都有各自的优缺点。常用的存储引擎有 **InnoDB**、**MyISAM**、**Memory**、**Merge**、**Archive**、**CSV**、**BLACKHOLE**。其中，**InnoDB 是 MySQL 默认的存储引擎**。可以使用命令 SHOW ENGINES;查看当前数据库支持的存储引擎。

```
SHOW ENGINES
```

Engine	Support	Comment	Transactions	XA	Savepoints
MEMORY	YES	Hash based, stored i	NO	NO	NO
MRG_MYISAM	YES	Collection of identic	NO	NO	NO
CSV	YES	CSV storage engine	NO	NO	NO
FEDERATED	NO	Federated MySQL st	(Null)	(Null)	(Null)
PERFORMANCE_SCH	YES	Performance Schem	NO	NO	NO
MyISAM	DEFAULT	MyISAM storage en	NO	NO	NO
InnoDB	YES	Supports transactior	YES	YES	YES
BLACKHOLE	YES	/dev/null storage en	NO	NO	NO
ARCHIVE	YES	Archive storage eng	NO	NO	NO

以下是 MySQL 支持的存储引擎的详细介绍：

- **InnoDB**：MySQL **默认的存储引擎**，支持**事务、行级锁机制和外键约束**，适合**处理大量数据和高并发**的应用场景，但对于**频繁的全表扫描和大量的写操作**，性能可能不如 MyISAM。
- **MyISAM**：**MySQL 最早提供的存储引擎**，**不支持事务、行级锁机制和外键约束**，但对于**只读**或者**大量的查询操作**，性能比 InnoDB 更好。
- **Memory**：这种类型的数据表**只存在于内存中**，使用**散列索引**，数据的存取速度非常快。因为是存在于内存中，所以这种类型常应用于**临时表**中。
- **Merge**：将**多个相同的 MyISAM 表合并为一个虚表**，常应用于**日志**和**数据仓库**。
- **Archive**：适用于对于**只偶尔需要查询**的历史数据进行存储，将数据进行**压缩存储**，**占用空间小**，但**不支持索引和更新操作**。
- **CSV**：将数据**以CSV**格式存储，适合用于**导入和导出**数据。
- **BLACKHOLE**：这种存储引擎**不实际存储数据**，**所有写入的数据都会被丢弃**，但可以**记录数据的写入日志**。

可以通过 CREATE TABLE 语句创建表时指定使用的存储引擎，例如：

```
CREATE TABLE mytable (id INT, name VARCHAR(20)) ENGINE=InnoDB;
```

使用 SHOW CREATE TABLE 语句可以查看表的创建语句，其中包含了使用的存储引擎。例如：

```
SHOW CREATE TABLE mytable;
```

输出结果中包含了 ENGINE=InnoDB。

- 这两种存储引擎之间有以下区别，如何选择存储引擎取决于应用程序的需求和特点。

1. **事务支持**：InnoDB 支持事务处理，可以使用 **ACID**（原子性、一致性、隔离性、持久性）来保证数据的完整性和一致性。而 MyISAM 不支持事务处理，不能保证数据的一致性。如果需要使用事务，应该选择 InnoDB。
2. **锁机制**：InnoDB 采用**行级**锁定，只锁定需要修改的行，**提高并发性能**。而MyISAM采用**表级**锁定，会锁定整个表，如果多个用户同时访问一个表，就会出现互相等待的情况，降低并发性能。如果需要高并发性能，应该选择 InnoDB。
3. **外键约束**：InnoDB 支持外键约束，可以通过外键约束实现关联查询和级联删除等功能。而MyISAM不支持外键约束。如果需要使用外键约束，应该选 InnoDB。
4. **性能**：MyISAM在读取数据方面的性能表现较好，在大量读取的情况下效率更高。而 InnoDB 在处理事务和大量并发查询的情况下性能更好。选择存储引擎的时候需要根据应用程序的读写比例和并发性能的需求来选择。
5. 其他：MyISAM 支持**全文搜索索引**，而 InnoDB 不支持。MyISAM 的表可以被压缩，而 InnoDB 的表不支持压缩。

因此，在选择存储引擎时，需要根据应用程序的需求和特点进行选择。如果**需要支持事务和外键约束，以及并发性能比较重要**，应该选择 InnoDB。如果**需要读取数据的性能**比较重要，可以选择 MyISAM。如果需要全文搜索索引或表压缩，应该选择 MyISAM。

> 作者：编程导航知识星球 + 星球鱼友们

Spring 框架

1、Spring 框架是什么？使用 Spring 框架有哪些好处?

官方解析

Spring 框架是一个开源的 Java 企业应用程序框架，它通过依赖注入(Dependency Injection)和面向切面编程(Aspect Oriented Programming)等技术为开发者提供了一个全面的编程和配置模型。它可以降低 Java 开发的复杂度，提高代码的可维护性和可测试性，使得开发者能够更专注于业务逻辑的实现。

使用Spring框架有以下好处：

- 依赖注入(Dependency Injection)：通过 Spring 框架的依赖注入功能，开发者可以将应用程序中的不同组件之间的依赖关系交给 Spring 来管理，从而降低组件之间的耦合度，并方便后续的组件替换和维护。
- 面向切面编程(Aspect Oriented Programming)：Spring 框架提供了面向切面编程的支持，可以将应用程序的不同功能抽象成切面，并将这些切面与应用程序中的不同组件关联起来，从而降低了应用程序中的重复代码量，并提高了代码的可重用性和可维护性。
- 提供了多种技术整合方案：Spring 框架可以与其他的 Java 企业应用程序框架和技术进行整合，如 Hibernate、MyBatis、Struts、JSF 等，从而降低了技术整合的复杂度。

- 支持声明式事务管理：Spring 框架提供了声明式事务管理的支持，开发者可以通过配置来管理应用程序中的事务，从而简化了事务管理的过程。
- 提供了 IoC 容器：Spring 框架提供了一个 IoC 容器，可以对应用程序中的不同组件进行管理，并支持对组件进行 AOP 增强，从而实现了应用程序中的组件解耦和高度可配置性。
- 便于测试：Spring 框架可以方便地进行单元测试和集成测试，提高了代码的可测试性和可靠性。

鱼友的精彩回答

一切总会归于平淡的回答

介绍

Spring 框架是一个开源的 Java 应用程序框架，用于构建企业级应用程序。它提供了全面的基础设施支持，包括控制反转（IoC）、依赖注入（DI）、AOP（面向切面编程）等功能，使开发者可以更加专注于业务逻辑的实现，而不必处理底层的框架代码。

- Spring 框架由多个模块组成，其中核心模块是 Spring Framework，它提供了许多常用的工具和服务，例如 Spring MVC、Spring ORM、Spring JDBC、Spring Security等。另外，Spring 还有其他的项目和模块，例如 Spring Boot、Spring Cloud 等，可以进一步扩展和增强框架的功能。

开发场景：

- Web 应用程序开发：Spring 框架是最常用的 Java Web 应用程序框架之一，因此在 Web 开发中广泛使用。使用 Spring 可以轻松地处理 Web 请求和响应，而且还提供了许多其他有用的功能，如事务管理、安全性、数据访问和集成等。
- RESTful API 开发：Spring 提供了一种简单的方法来创建 RESTful API，并允许使用各种 HTTP 方法处理请求。它还提供了许多工具来解析 JSON 数据、处理异常、执行身份验证等等。
- 企业级应用程序开发：Spring 的核心目标是使企业级应用程序开发更加容易和高效。它提供了一整套工具和框架，可以大大减少企业级应用程序的开发和维护成本。Spring 框架中包含了许多常用的设计模式和最佳实践，如依赖注入、面向切面编程、模板模式、单元测试等等。
- 批处理应用程序开发：Spring 提供了许多用于批处理应用程序开发的工具和框架，如 Spring Batch、Spring Integration 和 Spring XD。Spring Batch 是一个强大的批处理框架，可用于处理大量数据，并提供了一整套工具来处理事务、处理异常、执行监控等等。Spring Integration 是一个企业集成框架，可用于将不同的系统和组件集成到一个统一的体系结构中。Spring XD 是一个分布式数据处理框架，可用于处理大数据流。
- 移动应用程序开发：Spring 提供了一些用于移动应用程序开发的工具和框架，如 Spring Mobile 和 Spring for Android。Spring Mobile 可用于开发适用于移动设备的 Web 应用程序，并根据设备的特定属性提供不同的视图和功能。Spring for Android 则可用于开发 Android 应用程序，并提供了一些有用的功能，如异步任务处理、远程调用和 Web 服务集成等等。

如果面试官问到你关于 Spring 框架的问题，可能想了解你对 Spring 的了解程度，以及你是否具备使用 Spring 进行开发的能力。此外，可能也想了解你对 Spring 相关技术的掌握程度，如 Spring Boot、Spring Cloud 等。

having的补充

Spring 框架的主要优点具体如下：
1）方便解耦，简化开发。
Spring 就是一个大工厂，可以将所有对象的创建和依赖关系的维护交给 Spring 管理。

2）方便集成各种优秀框架。
Spring 不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如 Struts2、Hibernate、MyBatis 等）的直接支持。

3）降低 Java EE API 的使用难度。
Spring 对 Java EE 开发中非常难用的一些 API（JDBC、JavaMail、远程调用等）都提供了封装，使这些 API 应用的难度大大降低。

4）方便程序的测试。
Spring 支持 JUnit，可以通过注解方便地测试 Spring 程序。

5）AOP 编程的支持。
Spring 提供面向切面编程，可以方便地实现对程序进行权限拦截和运行监控等功能。

6）声明式事务的支持。
只需要通过配置就可以完成对事务的管理，而无须手动编程。

2、Spring 的两大核心概念是什么？简单讲一下你对它们的理解

官方解析

Spring 框架的两大核心概念是控制反转（Inversion of Control，IoC）和面向切面编程（Aspect Oriented Programming，AOP）。

控制反转指的是将对象的创建和依赖注入由应用代码转移到了 Spring 容器中进行，即由 Spring 容器负责创建对象和管理它们之间的依赖关系。这样，应用代码只需要关注业务逻辑的实现，而不需要关注对象的创建和管理，降低了应用代码的复杂度，提高了代码的可重用性和可维护性。

面向切面编程是指将与业务逻辑无关的代码（如日志、安全、事务等）从业务逻辑中剥离出来，以便于统一管理和维护。通过 AOP，我们可以将这些与业务逻辑无关的横切关注点（Cross-cutting Concerns）定义为切面（Aspect），并将它们织入到业务逻辑中，从而实现了业务逻辑与横切关注点的解耦。

这两个概念是 Spring 框架的核心，它们使得 Spring 框架具有了高度的可扩展性、灵活性和模块化，极大地提高了应用程序的开发效率和代码的可维护性。

鱼皮补充：这题在回答的时候可以通过举自己做项目的例子，来表达使用这两个核心概念的好处，比如用 AOP 来实现全局统一登录校验，就不用在每个方法里单独校验了。

鱼友的精彩回答

大橘已定的回答

IOC

- IOC(Inverse Of Controll,控制反转)：就是原来代码里面需要自己手动创建的对象，依赖，反转给Spring来帮忙实现。我们需要创建一个容器，同时需要一种描述来让容器知道要创建的对象与对象之间的关系。
- 在Spring中BeanFactory就是IOC容器，在Spring初始化的时候，创建容器，并将需要创建对象和对象的关系（xml，注解）通过BeanDefinitionReader加载到BeanDefinition中并保存在BeanDefinitionMap中，然后再由IOC容器创建bean对象.

两种bean的注册方式

- 方法1：通过@Bean+@Configuration的方式直接定义要创建的对象与对象的关系
- 方式2：通过@Component定义类，这种方式必须使用@ComponetScan定位Bean扫描路径

AOP

- AOP 在面向对象编程（oop）思想中，我们将事物纵向抽成一个个的对象。而在面向切面编程中，我们将一个个的对象某些类似的方面横向抽成一个切面，对这个切面进行一些如权限控制、事物管理，记录日志等公用操作处理的过程就是面向切面编程的思想。AOP 底层是动态代理，如果是接口采用 JDK 动态代理，如果是类采用CGLIB 方式实现动态代理。

having 的回答

一、IOC（Inverse of Control）：控制反转，也可以称为依赖倒置
所谓依赖，从程序的角度看，就是比如A要调用B的方法，那么A就依赖于B，因为A要用到B，所以A就必须依赖于B的方法。

所谓倒置，你必须理解如果不倒置，会怎么着，因为A必须要有B，才可以调用B，如果不倒置，意思就是A主动获取B的实例：B b = new B()，这就是最简单的获取B实例的方法（当然还有各种设计模式可以帮助你去获得B的实例，比如工厂、Locator等等），然后你就可以调用b对象了。

所以，不倒置，意味着A要主动获取B，才能使用B；到了这里，就应该明白了倒置的意思了。倒置就是A要调用B的话，A并不需要主动获取B，而是由其它人自动将B送上门来。

二、AOP：面向切面编程
面向切面编程的目的就是分离关注点。

什么是关注点呢？就是你要做的事，就是关注点。

假如你是个公子哥，没啥人生目标，天天就是衣来伸手，饭来张口，整天只知道玩一件事！那么，每天你一睁眼，就光想着吃完饭就去玩（你必须要做的事），但是在玩之前，你还需要穿衣服、穿鞋子、叠好被子、做饭等等等等事情，这些事情就是你的关注点，但是你只想吃饭然后玩，那么怎么办呢？这些事情通通交给别人去干。在你走到饭桌之前，有一个专门的仆人A帮你穿衣服，仆人B帮你穿鞋子，仆人C帮你叠好被子，仆人C帮你做饭，然后你就开始吃饭、去玩（这就是你一天的正事），你干完你的正事之后，回来，然后一系列仆人又开始帮你干这个干那个，然后一天就结束了！

3、什么是 IOC，简单讲一下 Spring IOC 的实现机制?

官方解析

IOC (Inversion of Control)，中文翻译为**控制反转**，是一种**编程思想**，它将程序中**对象的创建、组装、管理**等控制权**从代码中转移到框架中**，实现了**松耦合**和**可重用性**的设计。

Spring IOC 是 Spring 框架的一个核心特性，它的实现机制主要包括以下几个步骤：

- **定义 Bean**：在 Spring IOC 中，所有的对象都被看作是 Bean，需要在配置文件或者使用注解的方式中进行定义和配置。
- **创建 Bean 工厂**：在 Spring 中，Bean 工厂负责管理 Bean 的创建、组装和销毁等任务。Spring IOC 容器就是 Bean 工厂的一种实现。
- **读取配置文件**：Spring IOC 容器会读取配置文件或者使用注解的方式来获取 Bean 的定义和配置信息。
- **创建 Bean 实例**：Spring IOC 容器根据配置文件中的信息，使用**反射**技术来创建 Bean 实例，并将其保存在容器中。
- **组装 Bean**：Spring IOC 容器根据配置文件中的信息，将不同的 Bean 实例组装起来，形成一个完整的应用程序。
- **注入依赖**：Spring IOC 容器根据配置文件中的信息，自动为 Bean 注入依赖的对象或者值。
- **提供 Bean 实例**：应用程序通过 Spring IOC 容器获取需要的 Bean 实例，从而使用其中的方法和属性等。

需要注意的是，Spring IOC 还提供了多种作用域，例如**单例、原型、会话、请求**等作用域，可以根据具体的需求来选择。

同时，Spring IOC 容器也支持 AOP、事务管理等功能，可以为应用程序提供更完整的服务。

鱼友的精彩回答

大橘已定的回答

什么是IOC容器，以及IOC的创建过程

基本概念

- IOC(Inverse Of Controll,控制反转)：就是**原来代码里面需要自己手动创建的对象，依赖，反转给 Spring 来帮忙实现**。我们需要创建一个容器，同时需要一种描述来让容器知道要创建的对象与对象之间的关系。
- 在 Spring 中 BeanFactory 就是 IOC 容器，在 Spring 初始化的时候，创建容器，并将需要创建对象和对象的关系（xml，注解）通过 BeanDefinitionReader 加载到 BeanDefinition 中并保存在 BeanDefinitionMap 中，然后再由 IOC 容器创建 bean 对象.

两种 bean 的注册方式

- 方法1：通过 @Bean + @Configuration 的方式直接定义要创建的对象与对象的关系
- 方式2：通过 @Component 定义类，这种方式必须使用 @ComponetScan 定位 Bean 扫描路径

IOC的创建

- 在 Spring 中 BeanFactory 就是 IOC 容器，在 Spring 初始化的时候，创建容器，并将需要创建对象和对象的关系（xml，注解）通过 BeanDefinitionReader 加载到 BeanDefinition 中并保存在 BeanDefinitionMap 中，在这个过程中会让 BeanDefinitionProcesser(Bean 的定义信息的后置处理器)进行增强，然后再由 IOC 容器创建 bean 对象.

Bean的生命周期(面试官顺着问题往下问的拓展)

- bean 的实例化：spring 启动后,会查找和加载需要被 spring 管理的 Bean,并且实例化
- bean 的属性注入(bean 的初始化)：bean 被实例化后将 Bean 的引用和值注入到 bean 的属性中
    - 查看是否调用一些 aware 接口,比如 BeanFactoryAware,BeanFactoryAware,ApplicationContextAware 接口,分别会将 Bean 的名字,BeanFactory 容器实例,以及 Bean 所在的上下文引用传入给 Bean
    - 在初始化之前,会查看是否调用了 BeanPostProcessor 的预初始化方法,可以对 bean 进行扩展
    - 调用 InitializingBean 的 afterPropertiesSet()方法：如果 Bean 实现了 InitializingBean 接口，spring 将调用他们的afterPropertiesSet()方法，类似的，如果 Bean 使用 init-method 生命了初始化方法的话，这个方法也会被调用。
    - 初始化成功之后,会查看是否调用 BeanPostProcessor 的初始化后的方法：如果 Bean 实现了 BeanPostProcessor 接口，spring 就将调用他们的 postprocessAfterInitialization()方法。可以对 bean 进行扩展
- bean的正常使用：可以被应用程序正常使用了,他们将驻留在上下文中,直到应用的上下文被销毁
- bean的销毁：调用 DisposableBean 的destory()方法：如果 Bean 实现 DisposableBean 接口，spring 将用他的 destory()方法，相同的，如果 Bean 使用了 destory-method 生命销毁方法，该方法也会被调用。(但由于 bean 也分为单例和多例,单例 bean 会随着 IOC 容器的销毁而销毁,多例的 bean 不会随着 IOC 容器的销毁而销毁,他是通过 JVM 里面的垃圾回收器负责回收)

4、Spring 框架中都用到了哪些设计模式?

官方解析

Spring 框架中使用了许多设计模式，以下列举一些比较重要的：

- **单例模式**：Spring 的 Bean 默认是单例模式，通过 Spring 容器管理 Bean 的生命周期，保证每个 Bean 只被创建一次，并在整个应用程序中重用。
- **工厂模式**：Spring 使用工厂模式通过 BeanFactory 和 ApplicationContext 创建并管理 Bean 对象。
- **代理模式**：Spring AOP 基于动态代理技术，使用代理模式实现切面编程，提供了对 AOP 编程的支持。
- **观察者模式**：Spring 中的事件机制基于观察者模式，通过 ApplicationEventPublisher 发布事件，由 ApplicationListener 监听事件，实现了对象间的松耦合。
- **模板方法模式**：Spring 中的 JdbcTemplate 使用了模板方法模式，将一些固定的流程封装在父类中，子类只需实现一些抽象方法即可。
- **策略模式**：Spring 中的 HandlerInterceptor 和 HandlerExecutionChain 使用了策略模式，允许开发者自定义处理器拦截器，按照一定顺序执行。
- **责任链模式**：Spring 中的过滤器和拦截器使用了责任链模式，多个过滤器和拦截器按照一定顺序执行，每个过滤器和拦截器可以拦截请求或者响应并做出相应的处理。

总之，Spring 框架中充分利用了许多设计模式，提供了良好的扩展性和灵活性，降低了代码的耦合度，提高了代码的可维护性。

> 鱼友补充：这题感兴趣的同学可以去看下部分源码，不用记所有的设计模式，重点把单例、代理、工厂、责任链理解了即可

鱼友的精彩回答

Gundam 的回答

依赖注入/控制反转（DI/IOC）：这是 Spring 框架最为核心的设计模式，通过依赖注入实现对象的解耦和松耦合，使得对象之间的关系更加灵活。

模板方法模式：Spring框架中的JdbcTemplate和HibernateTemplate等模板类都是采用模板方法模式，定义了一些抽象方法，由具体的子类实现。

工厂模式：Spring框架中的BeanFactory和ApplicationContext等容器类都是采用工厂模式，通过工厂方法创建对象，从而实现对象的解耦。

单例模式：Spring框架中的Bean默认是单例模式，即在容器中只创建一次，并且在整个应用中都可以共享使用。

观察者模式：Spring框架中的事件机制就是采用观察者模式，通过注册监听器和发布事件的方式实现对象之间的解耦。

装饰器模式：Spring框架中的AOP就是采用装饰器模式，通过动态代理和切面编程实现对方法的增强和横切关注点的分离。

适配器（Adapter）模式：Spring MVC 框架中的 HandlerAdapter 就是一个适配器模式的实现，它将不同类型的处理器适配成 Spring MVC 可以处理的处理器。

注册模式：Spring 的 BeanFactory 和 ApplicationContext 就是基于注册模式实现的，容器会将所有的 Bean 实例都注册到容器中，然后通过 Bean 名称或类型来获取实例。

代理（Proxy）模式：Spring AOP 框架就是基于动态代理实现的，通过代理可以对目标对象进行增强，比如添加事务管理等功能。

责任链模式：Spring 中的过滤器和拦截器使用了责任链模式，多个过滤器和拦截器按照一定顺序执行，每个过滤器和拦截器可以拦截请求或者响应并做出相应的处理。

5、Spring、SpringMVC、SpringBoot 三者之间是什么关系?

官方解析

Spring、SpringMVC、SpringBoot 是三个独立的框架，它们之间的关系是：

1. Spring 是一个 Java 的**轻量级应用框架**，提供了基于 **IoC** 和 **AOP** 的支持，用于构建企业级应用。Spring 有多个模块，包括 **Spring Core**、**Spring Context**、**Spring JDBC**、**Spring Web** 等，每个模块提供了不同的功能。

2. SpringMVC 是 Spring 框架的一部分，是**基于 MVC 设计模式的 Web 框架**，用于构建 Web 应用程序。它提供了**控制器、视图解析器、数据绑定、异常处理**等功能，使得开发 Web 应用变得更加简单。SpringMVC 还支持 RESTful 架构。
3. SpringBoot 是**基于 Spring 框架的一个开发框架**，用于快速构建独立的、生产级别的 Spring 应用程序。它通过**自动配置**和**约定优于配置**的方式，简化了 Spring 应用程序的配置和开发过程。SpringBoot 集成了很多常用的第三方库和工具，例如 Spring Data、Spring Security、Thymeleaf、Logback 等，可以极大地提高开发效率。

因此，SpringBoot 可以看作是**在 Spring 的基础上，通过自动配置和约定优于配置的方式**，提供了更加简单、快速的开发体验。而 **SpringMVC 则是 Spring 框架中用于构建 Web 应用程序的模块**。

> 鱼皮补充：这题可以提一下 Spring Boot 内置了 Tomcat、Undertow 等服务器，不用像传统 SSM 一样自己去搭 Tomcat 等环境了，简化了开发

6、有哪些注解可以注入 Bean？@Autowired 和 @Resource 的区别?

官方解析

在 Spring 框架中，常用的注入 Bean 的注解包括：

- @Autowired：自动注入，按照类型自动装配，如果有多个同类型的 Bean，则需要通过 @Qualifier 指定具体的 Bean。
- @Resource：Java 自带的注入方式，按照名称自动装配，默认是按照属性名称进行匹配，如果需要按照 Bean 的名称进行匹配，可以使用 @Resource(name="beanName")。
- @Inject：和 @Autowired 类似，也是按照类型进行自动装配，但是 @Inject 注解是 JSR-330 提供的，而 @Autowired 注解是 Spring 框架提供的。
- @Value：用于注入配置文件中的属性值，可以指定默认值。
- @Component：用于声明一个 Bean，作用类似于 XML 中的 <bean> 标签。

以上注解都可以用于注入 Bean，不同的注解之间的区别主要在于注入方式和实现方式的不同。@Autowired 和 @Resource 最常用，其中 @Autowired 按照类型自动装配更为常用，而 @Resource 按照名称自动装配则比较适合需要明确指定 Bean 名称的情况。

需要注意的是，注入 Bean 的时候最好使用接口类型作为注入对象，这样可以避免因为具体实现类变更导致注入失败的问题。

鱼友的精彩回答

Starry 的回答

Spring 内置的 @Autowired 以及 JDK 内置的 @Resource 和 @Inject 都可以用于注入 Bean。

一般在工作中，@Autowired和 @Resource 使用的比较多一些。

- @Autowired 是 Spring 提供的注解， @Resource 是 JDK 提供的注解。
- Autowired 默认的注入方式为 byType （根据类型进行匹配）， @Resource 默认注入方式为byName （根据名称进行匹配）。

- 当一个接口存在多个实现类的情况下， @Autowired 和 @Resource 都需要通过名称才能正确匹配到对应的 Bean。 Autowired 可以通过 @Qualifier 注解来显示指定名称， @Resource 可以通过 name 属性来显示指定名称。

7、Spring 如何处理线程并发问题，ThreadLocal 你了解过吗?

官方解析

Spring 框架中处理线程并发问题的方式包括以下几种：

- **同步关键字 synchronized**：使用 synchronized 关键字可以**对共享资源进行加锁**，从而**保证多线程访问时的同步性**。但是，synchronized 对性能会产生一定的影响，并且容易导致**死锁**等问题。
- **Lock 接口**：Lock 接口提供了比 synchronized 更加**灵活**的加锁方式，并且可以**防止死锁**问题的发生。但是，Lock 接口的**使用相对较复杂**，需要**手动进行加锁和解锁**操作。
- **ThreadLocal 类**：ThreadLocal 类提供了**线程本地变**量的功能，可以让每个线程拥有自己的**变量副本**，从而避免了多个线程之间的共享问题。但是，ThreadLocal 类的使用需要注意**内存泄漏**问题。

关于 ThreadLocal，它是 Java 中一个非常重要的类，用于实现线**程本地存储**，即让每个线程都拥有自己的变量副本，从而避免多个线程之间的共享问题。

在 **Spring** 框架中，ThreadLocal 类经常用于存储一些**与当前线程相关的数据**，例如**请求上下文、用户信息**等。通过将这些数据存储到 ThreadLocal 对象中，可以方便地在整个应用程序中进行访问和传递，同时**避免了多个线程之间的共享问题**。

ThreadLocal 类的使用需要注意**内存泄漏**问题，因为**线程本地变量只有在对应的线程被回收时才会被回收**，如果没有及时清理，就可能导致内存泄漏问题。因此，在使用 ThreadLocal 类时，需要注意在不需要存储数据时及时调用 **remove()** 方法清理 ThreadLocal 对象中的数据。

> 鱼皮的补充：这题大家可以结合自己的项目去举例并发编程和 ThreadLocal 的应用

鱼友的精彩回答

Starry 的回答

Spring 如何处理线程并发问题?

Spring 中处理线程并发问题的主要方式是**使用线程安全的对象**和**并发包中提供的类**来避免线程安全问题。

例如，Spring 中的**单例 Bean** 是线程安全的，因为 Spring 容器在创建单例 Bean 时会确保只有一个实例存在。Spring 还提供了对多线程的支持，例如使用 **@Async** 注解实现异步方法调用等。

在 Spring 中处理线程并发问题，常用的方法有以下几种：

- **Synchronized 关键字**：使用 synchronized 关键字可以**锁定某个对象或类**，使得多个线程无法同时进入该代码块，从而保证数据的安全性。
- **ReentrantLock**：与 synchronized 相比，ReentrantLock 提供了更加**灵活的加锁方式**，可以**控制锁的获取和释放的时机**，提供了更多的扩展功能。

- **Atomic 包**：Java 的 **Atomic** 包提供了一些**原子性操作**，例如 AtomicLong、AtomicInteger 等，可以保证某个操作的原子性。
- **ThreadLocal**：ThreadLocal 可以使得**每个线程拥有自己的变量副本**，避免了多个线程之间共享变量所带来的线程安全问题。

ThreadLocal 你了解过吗?

ThreadLocal 是 Java 中的一个**线程局部变量**，它可以**为每个线程提供一个独立的变量副本**，避免了多线程之间的**数据共享**和**数据竞争问题**。

ThreadLocal 可以在每个线程中存储一个对象，并且每个线程只能访问自己的对象，而不会影响其他线程中的对象。

ThreadLocal 主要用于解决**线程安全**问题，例如在 Web 应用中，可以**使用 ThreadLocal 存储用户的会话信息**，这样每个线程就可以**独立地访问自己的会话信息**，避免了多个线程之间的**数据共享**和**数据竞争**问题。

在 Spring 中，ThreadLocal 通常用于**存储和传递一些与线程相关的上下文信息**，例如**当前登录用户的信息、请求的 IP 地址等**。可以将 ThreadLocal 对象定义为一个 Bean，然后在需要使用时注入到其他 Bean 中使用。

Spring 还提供了一些与 ThreadLocal 相关的类和工具，例如 **SimpleThreadScope**，用于实现线程范围内的 Bean，以及 **TaskExecutor**，用于在多线程环境中执行任务。

8、Spring 中的 BeanFactory 和 ApplicationContext 有什么区别和联系?

官方解析

在 Spring 中，BeanFactory 和 ApplicationContext 都是用于**管理 Spring Bean 的容器**，它们的区别和联系如下。

区别：

1. BeanFactory 是 Spring 框架的**基础设施**，**用于管理 Bean 的生命周期和依赖关系**，提供了 **IoC** 和 **DI** 功能。**ApplicationContext 是 BeanFactory 的扩展**，提供了更多的功能，例如**国际化支持**、**AOP 支持**等。
2. BeanFactory 是**延迟加载**的，即只有在**获取 Bean 时才会进行实例化**，可以减少系统资源的占用。而 ApplicationContext 在**启动时会立即加载所有的 Bean**，导致启动时间较长。
3. BeanFactory 是**单例模式**，即在整个应用中只有一个 BeanFactory 实例。而 ApplicationContext 可以有**多个实例**，并且可以通过**父子容器**的方式组织起来，方便模块化开发。

联系：

1. BeanFactory 和 ApplicationContext 都是用于**管理 Spring Bean 的容器**，可以**管理 Bean 的生命周期和依赖关系**，提供了 **IoC** 和 **DI** 功能。
2. ApplicationContext 是 BeanFactory 的**扩展**，提供了更多的功能，例如**国际化支持**、**AOP 支持**等，同时也**支持 BeanFactory 的所有功能**。
3. BeanFactory 和 ApplicationContext 都可以**管理单例 Bean 和原型 Bean**，可以**控制 Bean 的作用域和生命周期**。

总结：BeanFactory 是 Spring 框架的基础设施，提供了 **IoC 和 DI** 功能，而 ApplicationContext 是 BeanFactory 的**扩展**，提供了更多的功能和扩展性，可以通过多种方式进行配置和管理 Spring Bean。

> 作者：编程导航知识星球 + 星球鱼友们

鱼友的精彩回答

CodeJuzi的回答

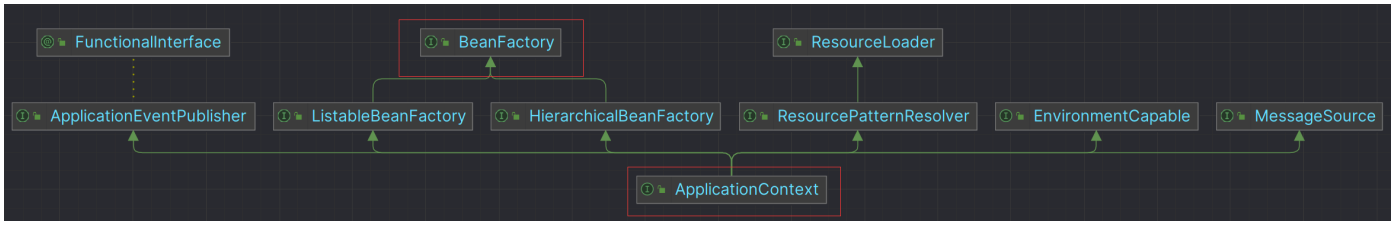
BeanFactory 和 ApplicationContext 是 Spring 框架中的两个核心接口，它们的主要区别如下：

- **容器加载方式不同**：BeanFactory 是 Spring 框架**最基础**的容器，提供了最基本的依赖注入和对象生命周期管理等功能。ApplicationContext 是 **BeanFactory 的子接口**，它在 BeanFactory 的基础上**增加了很多特性**，比如**国际化、事件机制**等，**并且在容器启动时就预先加载了所有的 bean**。
- **容器管理范围不同**：BeanFactory 是 Spring 框架的最基础容器，只提供了基本的 Bean 管理功能，通常在小型应用中使用。ApplicationContext 是更高级的容器，能够在**启动时预先加载所有的 Bean**，并提供了更多的应用功能，通常在中大型应用中使用。
- **容器启动速度不同**：**BeanFactory 启动较快**：BeanFactory 只在**实际获取 Bean 时才会去加载 Bean**，因此启动速度较快。而 **ApplicationContext 在启动时会预先加载所有的 Bean**，因此**启动速度较慢**，但是**获取 Bean 的速度较快**。
- **自动装配方式不同**：BeanFactory 提供的自动装配功能比较有限，只能通过配置文件进行装配。ApplicationContext 则提供了更强大的**自动装配**功能，可以通过注解、Java 配置等方式进行装配。
- **容器销毁时的处理不同**：BeanFactory 容器中的 Bean 是延迟加载的，容器销毁时不会自动销毁 Bean，需要手动进行销毁。ApplicationContext 容器中的 Bean 在容器销毁时会自动进行销毁。

=> 综上考虑，在使用 Spring 框架时，通常会优先选择使用 ApplicationContext，因为它提供了更多的功能，并且性能相对也更好。在特殊场景下，比如应用程序比较小，并且对性能要求较高，那么可以考虑使用 BeanFactory。

HeiHei 的回答

BeanFactory 是 Spring 容器的超级接口。**ApplicationContext 是 BeanFactory 的子接口**



区别：

- **BeanFactory**：Spring IoC 容器的顶级对象，BeanFactory被翻译为“Bean工厂”，在Spring的IoC容器中，“Bean工厂”负责创建Bean对象。BeanFactory是工厂。提供了基本的依赖注入和控制反转功能，它可以加载配置文件中定义的bean，并管理它们的生命周期。

- **ApplicationContext** 是 BeanFactory 的一个子接口，它在 **BeanFactory** 的基础上增加了更多的企业级功能，例如**事件发布**、**资源处理**、**AOP**等。ApplicationContext 还可以自动识别 Spring 框架内置的 bean，并对它们进行预处理。

9、讲一讲 Spring 框架中 Bean 的生命周期?

官方解析

在 Spring 框架中，Bean 的生命周期包括以下几个阶段：

1. 实例化（Instantiation）：在这个阶段，Spring 将根据配置文件或注解等方式创建 Bean 实例，并将其存储在容器中。
2. 属性赋值（Populate Properties）：在这个阶段，Spring 将会自动将 Bean 的属性值从配置文件或注解等方式中注入到 Bean 实例中。
3. 初始化（Initialization）：在这个阶段，Spring 会调用 Bean 实例的 init-method 方法，完成一些初始化的操作，例如建立数据库连接等。
4. 使用（In Use）：在这个阶段，Bean 实例已经可以正常使用，供应用程序调用。
5. 销毁（Destruction）：在这个阶段，Spring 会调用 Bean 实例的 destroy-method 方法，完成一些资源的释放和清理操作，例如关闭数据库连接等。

具体的实现方式可以通过实现 BeanPostProcessor 和 BeanFactoryPostProcessor 接口来进行扩展。其中，BeanPostProcessor 接口定义了两个方法 postProcessBeforeInitialization 和 postProcessAfterInitialization，分别在 Bean 的初始化前后被调用，用于扩展 Bean 初始化的过程；BeanFactoryPostProcessor 接口则定义了一个方法 postProcessBeanFactory，用于在 Bean 工厂实例化 Bean 定义后对其进行修改。

总之，Spring 的 Bean 的生命周期通过上述阶段进行管理，开发者可以通过实现相关接口和方法来扩展和定制 Bean 的创建和销毁过程，以满足各种业务需求。

鱼友的精彩回答

猿二哈的回答

bean 的生命周期

1. bean 的生命周期有五个阶段：
2. 创建前准备：主要是扫描 spring 中的上下文和配置（解析和查找）bean 的一些扩展配置
3. 创建实例：在这个阶段调用 bean 的构造器进行实例的创建。这是通过反射进行实例的创建的，并且会扫描和解析 bean 的一些属性
4. 依赖注入：实例化的 bean 可能依赖其他的 bean，这个阶段就是注入实例的 bean 依赖的 bean，如用注解 @Autowired 进行依赖的注入
5. 容器缓存：这个阶段中，将实例化的 bean 放入容器和 spring 的缓存中，此时开发者可以使用实例化的 bean
6. 销毁实例：当 spring 的上下文关闭时，这些上下文的 bean 也会被销毁

关键词：创建前准备，创建实例，依赖注入，容器缓存，销毁实例，反射，扩展配置，上下文

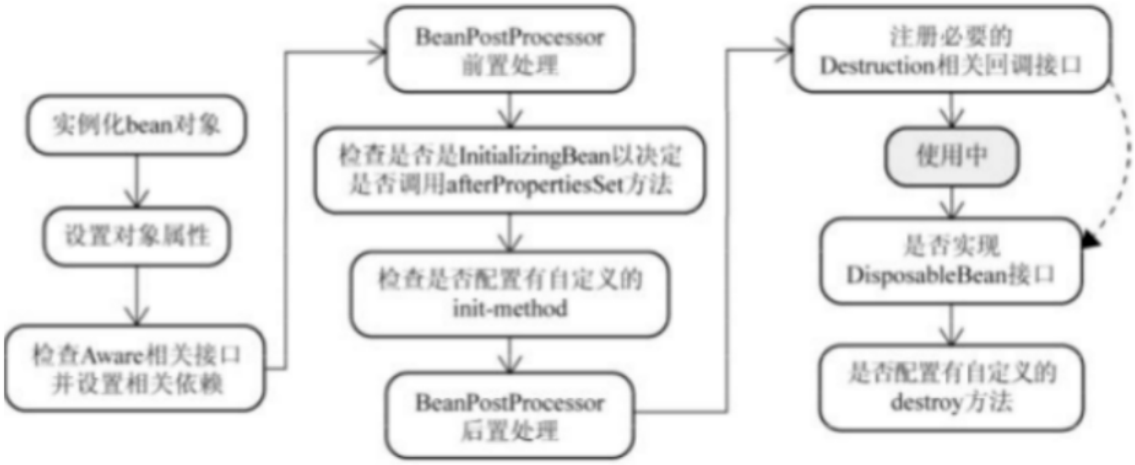
Starry 的回答

（1）默认情况下，IOC容器中 Bean 的生命周期分为五个阶段：

- 实例化 Bean：调用构造器或者是通过工厂的方式创建 Bean 对象。
- 属性注入：当 Bean 实例化后，Spring会给 Bean 对象的属性注入值。
- 初始化：调用初始化方法，进行初始化，初始化方法是通过 init-method 来指定的。
- 使用 Bean
- 销毁：IOC 容器关闭时，调用 Bean 的 destroy 方法，销毁 Bean 对象。

（2）当加入了 Bean 的后置处理器后，IOC 容器中 bean 的生命周期分为七个阶段：

- 实例化 Bean：当 Spring 容器启动时，根据 Bean 的定义（配置文件或注解）创建 Bean 的实例。在这个阶段，Spring 会使用 Java 反射或 CGLIB等技术创建 Bean 的实例。
- 属性注入：当 Bean 实例化后，Spring 会注入 Bean 的属性，包括基本类型、引用类型、集合类型等。属性注入可以通过构造函数注入、Setter 方法注入、注解注入等方式实现。
- BeanPostProcessor 的前置处理：当属性注入完成后，Spring 会调用所有实现了 BeanPostProcessor 接口的类的postProcessBeforeInitialization 方法，可以在这个方法中对 Bean 进行一些自定义的初始化操作。
- 初始化：在 BeanPostProcessor 的前置处理后，Spring 会调用 Bean 的 init 方法（可以是自定义的 init 方法或实现了 InitializingBean 接口的 afterPropertiesSet 方法），进行 Bean 的初始化工作。
- BeanPostProcessor 的后置处理：当 Bean 的初始化完成后，Spring 会调用所有实现了 BeanPostProcessor 接口的类的postProcessAfterInitialization 方法，可以在这个方法中对 Bean 进行一些自定义的后续处理。
- 使用 Bean
- 销毁：当 Spring 容器关闭时，Spring 会调用 Bean 的 destroy 方法（可以是自定义的 destroy 方法或实现了 DisposableBean 接口的 destroy 方法），进行 Bean 的销毁工作。



Algorithm 的回答

一、容器启动阶段

1、BeanDefinitionReader 读取 Bean 的配置信息（如XML等），将读取到每个 Bean 的配置信息使用 BeanDefinition 表示，同时注册到相应的 BeanDefinitionRegistry 中

2、之后会有 BeanFactory 的后置处理器，即 实现 BeanFactoryPostProcessor 的类，自定义修改 BeanDefinition 中的信息（此时 Bean 已经注册好了，现在做的只是修改已经注册好的 Bean 的信息）

二、Bean 的实例化阶段

1、**Bean 开始实例化**， 通过反射或 CGLIB（没听过）完成

2、**Bean 实例化完成后**，触发回调检测该 Bean 实现了哪个 Aware 接口，如实现了 BeanNameAware 子接口的可以获取到 Bean 的名称并 setBeanName。即根据 BeanDefinition 中的信息，使用 Aware 接口的实现类获取并 set，然后填充 Bean 的属性和依赖

3、**Bean 依赖注入完毕后**，执行 BeanPostProcessor 后置处理器，该接口有两个方法 postProcessBeforeInitialization 和 postProcessAfterInitialization，分别是在 Bean 初始化前和初始化后执行。

4、**Bean 初始化前**，执行 Bean 后置处理器的 postProcessBeforeInitialization 用来完成指定 Bean 的定制初始化任务

5、**Bean 开始初始化**，postProcessBeforeInitialization 执行完毕后，开始执行 Bean 的初始化方法，此时如果实现了 InitializingBean 接口和 自定义了 init-method 方法，则在 Bean 初始化期间执行其方法

6、**Bean 初始化结束**，执行 Bean 后置处理器的 postProcessAfterInitialization，执行初始化之后的任务

7、**Bean 开始使用**

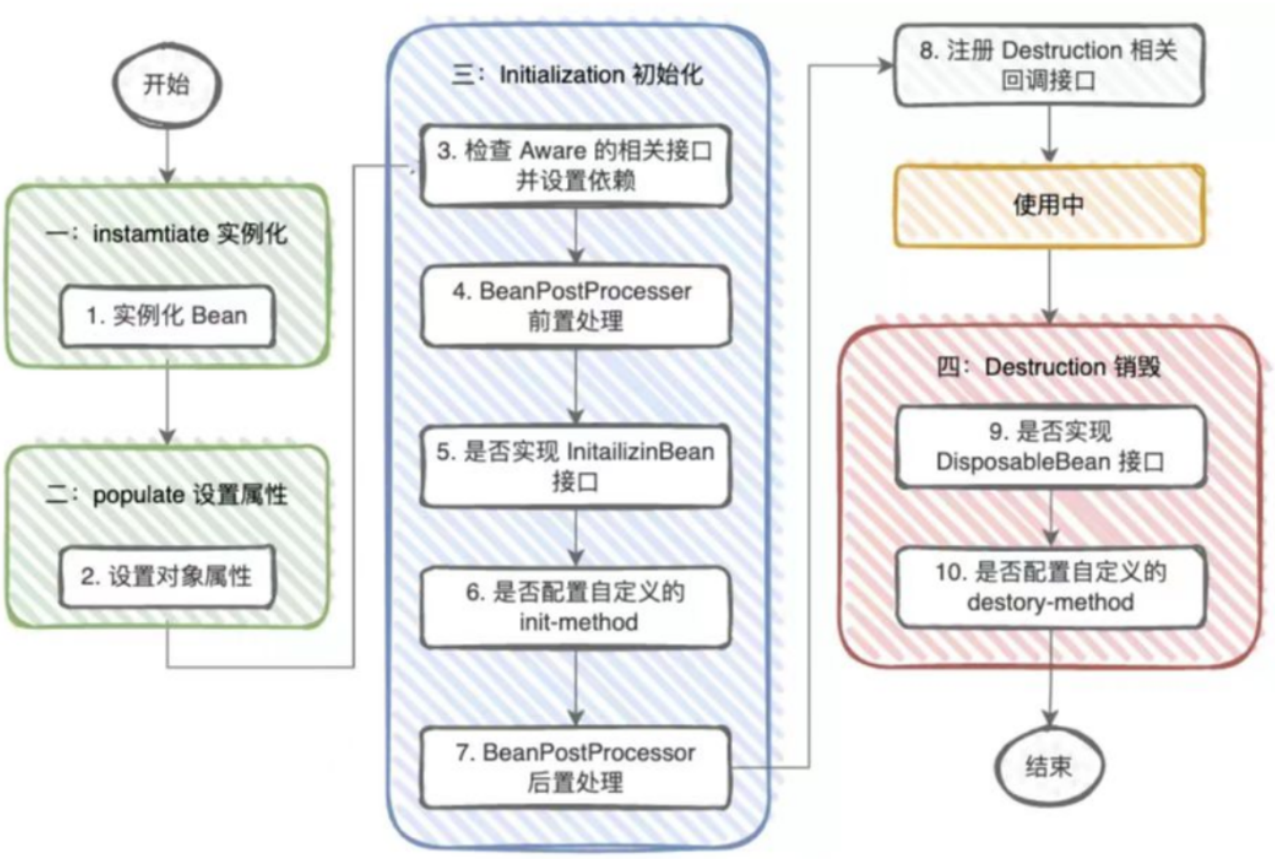
8、**Bean 销毁**，容器关闭时，上下文销毁，如果实现了 DisposableBean 接口，则执行对应的 destroy 方法，如果自定义了 destory-method，则执行对应的自定义销毁方法

Wbbp 的回答

Spring 单例 Bean 的生命周期

1. 实例化（Instantiation）： Spring 会根据配置信息或注解创建 Bean，通常调用构造函数（还有工厂方法）实现，实例化完成后将 Bean 放入工厂中进行管理
2. 属性赋值（Population）： Spring 为Bean 的属性进行赋值，包括依赖注入（Dependency Injection）、属性注入（Property Injection）、方法注入（Method Injection）等
3. 初始化（Initialization）： Spring 调用 Bean 的初始化方法进行初始化操作
4. 使用（Using）： Bean 正常使用

5. 销毁（Destruction）：Spring 调用 Bean 的销毁方法完成一些资源的释放和清理操作



生命周期中的扩展方法：

- Awar e接口：让 Bean 获取容器的资源，例如 BeanNameAware 的 setBeanName()，BeanFactoryAware()的setBeanFactory()
- BeanPostProcessor 处理器：进行初始化前置处理和后置的处理，对应 postProcessBeforeInitialization()和postProcessAfterInitialization()
- 生命周期接口：定义初始化方法和销毁方法的，例如 InitializingBean 的 afterPropertiesSet()，以及 DisposableBean 的 destory()
- 配置生命周期方法：通过配置文件自定义初始化和销毁方法，例如配置文件配置的 init()和 destoryMethod()

HeiHei 的回答

Bean 的生命周期之5步

- 实例化 Bean（调用无参数构造方法）
- Bean 属性赋值（调用 set 方法）
- 初始化 Bean（调用 Bean 的 init 方法）
- 使用 Bean
- 销毁 Bean（调用 Bean 的 destroy 方法）

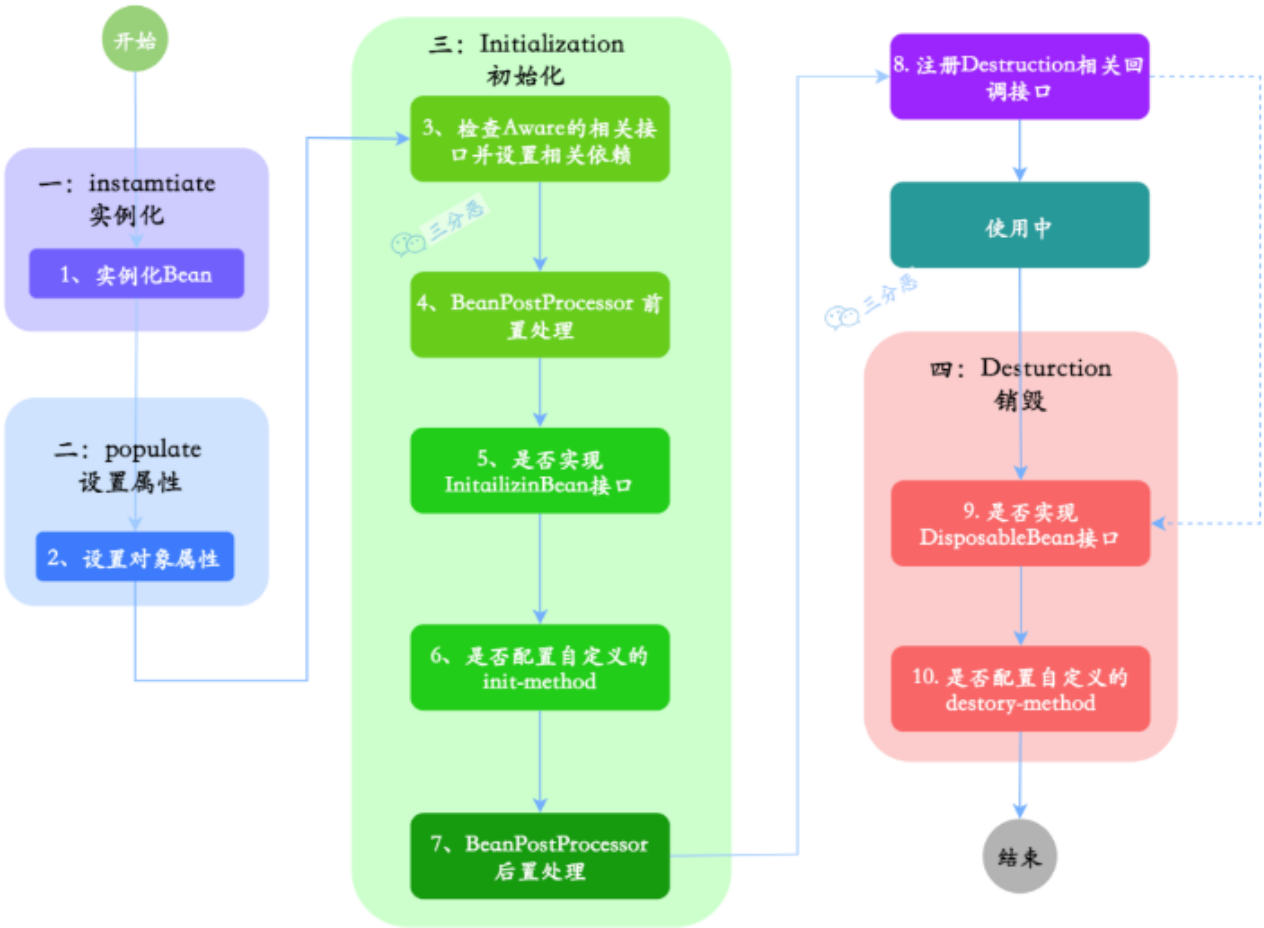
Bean 的生命周期之7步

在以上的 5 步中，第 3 步是初始化 Bean，如果想在初始化前和初始化后添加代码，可以加入“Bean 后处理器”。

Bean 的生命周期之10步

增加步骤：

1. 在 Bean 后处理器 before 之前
2. 在 Bean 后处理器 before 之后
3. 在使用 Bean 之后，或者销毁之前



Aware相 关的接口包括：BeanNameAware、BeanClassLoaderAware、BeanFactoryAware

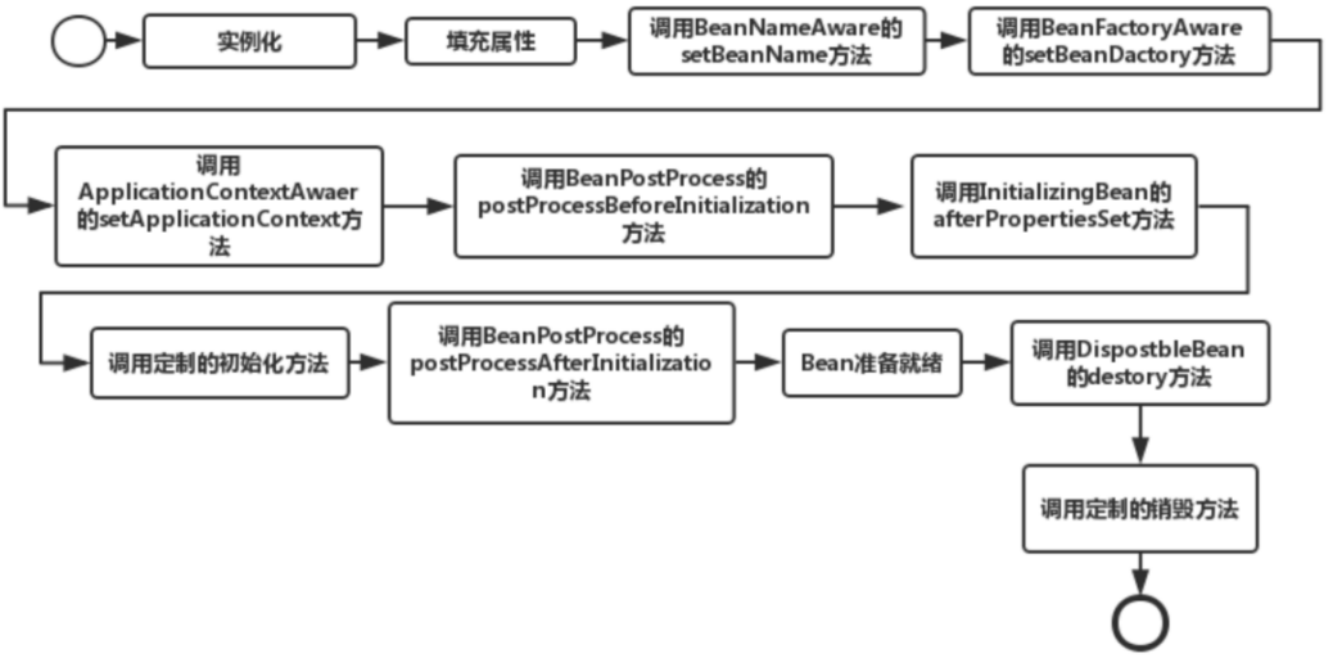
- 当 Bean 实现了 BeanNameAware，Spring 会将 Bean 的名字传递给 Bean。
- 当 Bean 实现了 BeanClassLoaderAware，Spring 会将加载该Bean的类加载器传递给 Bean。
- 当 Bean 实现了 BeanFactoryAware，Spring 会将 Bean工厂对象传递 给Bean。

会冒泡的可乐的回答

在 Spring 框架中，Bean 的生命周期包括以下几个阶段：

1. Spring 对 Bean 进行实例化
2. Spring 将值和 Bean 的引用注入进 Bean 对应的属性中
3. 如果 Bean 实现了 BeanNameAware 接口，Spring 将 Bean 的 ID 传递给 setBeanName()方法
4. 如果 Bean 实现了 BeanFactoryAware 接口，Spring 将调用 setBeanDactory(BeanFactory bf)方法并把 BeanFactory 容器实例作为参数传入。
5. 如果 Bean 实现了 ApplicationContextAwaer 接口，Spring 容器将调用 setApplicationContext(ApplicationContext ctx)方法，把 y 应用上下文作为参数传入.
6. 如果 Bean 实现了 BeanPostProcess 接口，Spring 将调用它们的 postProcessBeforeInitialization（预初始化）方法,作用是在 Bean 实例创建成功后对进行增强处理，如对 Bean 进行修改，增加某个功能
7. 如果 Bean 实现了 InitializingBean 接口，Spring 将调用它们的 afterPropertiesSet 方法.
8. 如果 Bean 实现了 BeanPostProcess 接口，Spring 将调用它们的 postProcessAfterInitialization（后初始化）方法
9. 经过以上的工作后，Bean 将一直驻留在应用上下文中给应用使用，直到应用上下文被销毁
10. 如果 Bean 实现了 DispostbleBean 接口，Spring 将调用它的 destory 方法，作用与在配置文件中对 Bean 使用 destory-method 属性的作用一样，都是在 Bean 实例销毁前执行的方法。

Spring 中 Bean 生命周期过程图如下：



10、Spring 支持哪几种事务管理类型，Spring 的事务实现方式和实现原理是?

官方解析

Spring 支持的事务管理类型包括：

1. **编程式事务管理**：在代码中显式地编写事务管理相关代码，如开启事务、提交事务、回滚事务等。
2. **声明式事务管理**：使用 AOP 技术，在代码中通过配置进行声明，从而实现对事务管理的控制。
3. **注解式事务管理**：基于**声明式事务管理**，使用注解的方式进行事务的管理。

Spring 的事务实现方式采用了模板模式，通过模板模式实现了事务的封装。Spring 事务管理实现原理主要是通过 AOP 和代理模式来实现的。在使用**声明式事务管理**时，Spring 通过拦截器和代理对象来实现对方法调用的拦截和控制。当方法被调用时，Spring 会在方法调用前开启一个新的事务，如果方法执行成功，Spring 会提交事务，否则回滚事务。在使用注解式事务管理时，Spring 会扫描带有事务注解的方法，并在运行时使用动态代理为这些方法生成代理对象，在代理对象的方法调用前后进行事务管理的操作。

鱼友的精彩回答

猫十二懿的回答

Spring 支持的事务管理有

1. 编程式事务管理：使用编程的方式来管理事务，包括事务的开始、提交或回滚等操作，可以精细地控制事务的边界。
2. 声明式事务管理：使用注解或 XML 配置来声明事务的属性，例如事务的隔离级别、传播行为、超时时间等，Spring 会自动为这些方法添加事务管理的支持，无需手动编写事务管理代码。
3. 注解式事务管理：通过在方法上添加 @Transactional 注解，开发人员可以非常方便地声明事务的属性，例如隔离级别、传播行为、超时时间等

Spring 的事务实现方式和实现原理是

Spring 事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring 是无法提供事务功能的。真正的数据库层的事务提交和回滚是通过 binlog 或者 redo log 实现的。

1. **事务管理器（Transaction Manager）**：Spring 事务管理器是事务的核心组件，它负责管理事务的生命周期和事务的状态。Spring 事务管理器提供了多种实现，包括 JDBC、Hibernate、JPA、JTA 等，开发人员可以根据自己的需求来选择适合的事务管理器。
2. **事务代理（Transaction Proxy）**：Spring 使用代理模式来实现事务管理，即通过代理对象来拦截被代理对象的方法调用，以实现事务的开启、提交或回滚等操作。Spring 事务代理有两种实现方式：基于 AOP 的代理和基于动态代理的代理。
3. **事务切面（Transaction Aspect）**：Spring 事务切面是一组通知（Advice）和切点（Pointcut）的组合，用于定义事务的行为，例如事务的开启、提交或回滚等操作。Spring 事务切面可以基于注解或 XML 配置来实现，开发人员可以根据自己的需求来选择适合的方式。
4. **事务同步器（Transaction Synchronization）**：Spring 事务同步器用于管理事务的提交或回滚时需要执行的一些操作，例如清理缓存、释放资源等。Spring 事务同步器通过 Synchronization 接口和 TransactionSynchronizationManager 类来实现，可以实现对事务的后置处理。

通过以上四种实现方式提高了事务管理的效率和灵活性。

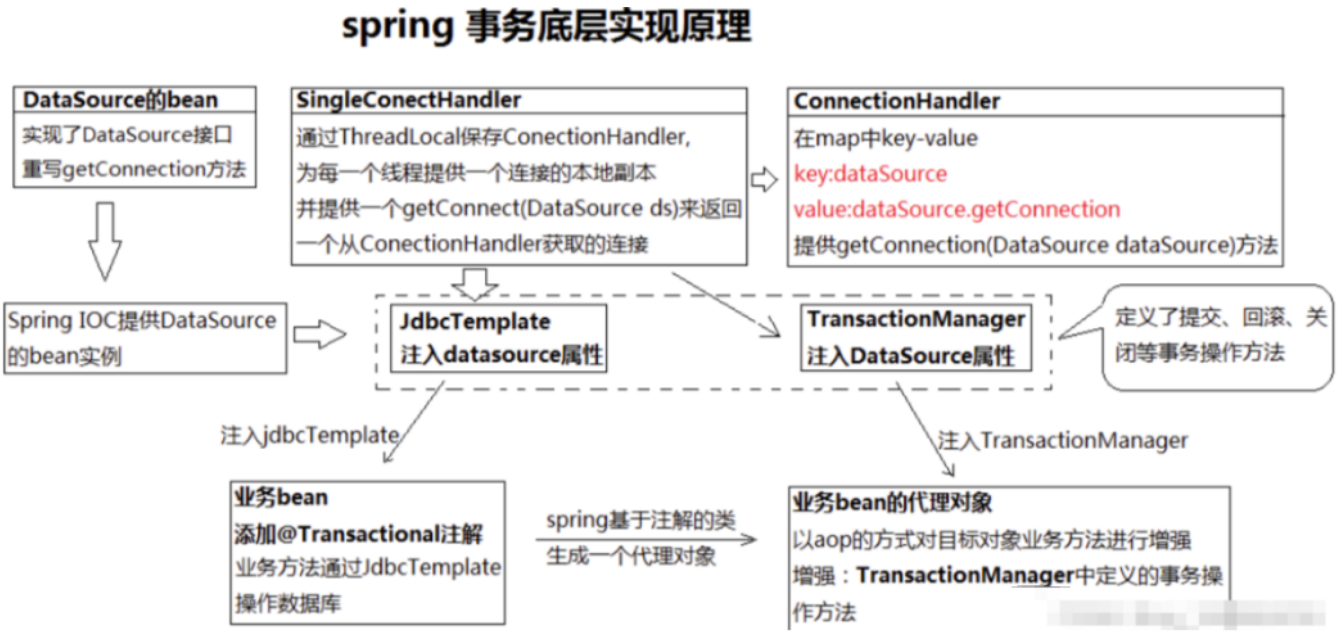
HeiHei 的回答

Spring 支持的事务管理类型：

- 编程式事务：
  - transactionTemplate：此种方式是自动的事务管理，无需手动开启、提交 、回滚
  - PlatformTransactionTemplate：此方式，可手动开启、提交、回滚事务。
- 声明式事务：
  - 基于Aspectj AOP开启事务
- 注解式事务管理：
  - @Transactional 的声明式事务管理

Spring 的事务实现原理：

- Spring 对事务的管理底层实现方式是**基于 AOP 实现的，采用 AOP 的方式进行了封装**，核心接口是 PlatformTransactionManager：
- 当在某个类或者方法上使用 @Transactional 注解后，**spring 会基于该类生成一个代理对象**，并将这个代理对象作为 bean。当调用这个代理对象的方法时，如果有事务处理，则会先关闭事务的自动功能，然后执行方法的具体业务逻辑，如果业务逻辑没有异常，那么代理逻辑就会直接提交，如果出现任何异常，那么直接进行回滚操作。当然我们也可以控制对哪些异常进行回滚操作。



11、什么是 Spring 的依赖注入，依赖注入的基本原则以及好处?

官方解析

依赖注入（Dependency Injection，简称 DI）是 Spring 框架中的一种设计模式，用于实现控制反转（Inversion of Control，简称 IoC）。它是一种将对象之间的依赖关系从硬编码中解耦的方法。通过依赖注入，Spring 框架可以在运行时自动为类的属性、构造函数或方法参数提供所需的依赖对象，从而实现对象之间的松耦合。

依赖注入的基本原则：

1. 高层模块不应该依赖低层模块。它们都应该依赖抽象。
2. 抽象不应该依赖具体实现。具体实现应该依赖抽象。

依赖注入的好处：

1. 解耦：依赖注入降低了组件之间的耦合度，使得组件可以独立地进行开发、测试和维护。通过将组件的创建和管理交给IoC容器，组件之间的依赖关系变得更加清晰，有助于提高代码的可维护性。
2. 提高代码的可测试性：依赖注入使得对组件进行单元测试变得更加容易。通过使用依赖注入，我们可以轻松地为组件提供模拟（Mock）的依赖对象，从而实现组件在隔离环境中的测试。
3. 更好的代码重用：由于组件之间的依赖关系变得更加清晰，这有助于提高代码的可重用性。组件可以在不同的上下文中重用，而无需对其进行修改。
4. 更简洁的代码：依赖注入使得代码更加简洁，因为组件不再需要直接处理依赖对象的创建和管理。这使得开发人员可以更加专注于组件的核心功能，从而提高开发效率。
5. 更容易进行配置管理：依赖注入允许我们将组件的配置与实现代码分离，从而使得配置管理变得更加容易。通过使用外部配置文件或注解，我们可以在不修改代码的情况下，调整组件之间的依赖关系。

鱼友的精彩回答

Gundam 的回答

Spring 的依赖注入（Dependency Injection，简称 DI）是指，在一个对象创建时，由 Spring 框架动态地将其依赖的其他对象注入到它当中，以解耦对象之间的关系，达到灵活、可维护、可测试的目的。

依赖注入的基本原则是控制反转（Inversion of Control，简称 IoC），即将对象之间的关系由调用者的控制转变为被调用者的控制。在传统的开发模式中，一个对象创建时通常需要主动创建它所依赖的其他对象，这些对象的创建、配置、连接等关系都由这个对象主动控制，这会导致对象之间高耦合、难以扩展、难以维护。而采用依赖注入的方式，则将这些依赖关系交给 Spring 框架来处理，大大降低了对象之间的耦合度，提高了代码的可读性、可维护性和可扩展性。

依赖注入的好处主要有以下几点：

1. 灵活性：通过依赖注入，对象之间的关系不再由程序员手动控制，而是由 Spring 容器动态地维护，可以灵活地组合和切换不同的对象实现。
2. 可测试性：依赖注入可以使各个对象之间的依赖关系更加松散，方便进行单元测试和集成测试。
3. 可扩展性：依赖注入将各个对象之间的依赖关系解耦，方便对系统进行扩展和升级。
4. 可维护性：依赖注入可以减少重复代码，降低了代码的复杂度，使代码更加易于维护和修改。

12、什么是 AOP？有哪些实现 AOP 的方式？Spring AOP 和 AspectJ AOP 有什么区别？

官方解析

AOP（Aspect-Oriented Programming，面向切面编程）是一种编程范式，它旨在解决软件开发中的横切关注点（cross-cutting concerns）问题。横切关注点是那些分布于多个模块或对象的功能，例如日志记录、安全检查、事务管理等。AOP 通过将横切关注点与业务逻辑分离，从而提高了代码的模块化程度，使得开发更加简洁、易于维护。

Spring AOP 和 AspectJ AOP 是两种不同的 AOP 实现。

1. Spring AOP：是 Spring 框架中的 AOP 实现，基于动态代理实现。Spring AOP 主要用于解决 Spring 容器中 Bean 的横切关注点问题。由于它使用了动态代理，所以只支持方法级别的切面（即横切关注点只能织入方法的执行）。Spring AOP 的性能略逊于 AspectJ，但对于大部分应用来说，性能影响不大。
2. AspectJ AOP：是一个独立的、功能更强大的 AOP 实现，不仅支持方法级别的切面，还支持字段、构造器等其他切面。AspectJ 可以通过编译时织入（编译时修改字节码）或加载时织入（在类加载时修改字节码）的方式实现 AOP。Spring 可以与 AspectJ 结合使用，以提供更强大的 AOP 功能。

实现 AOP 的方式主要有以下几种：

1. 动态代理：通过代理模式，为目标对象生成一个代理对象，然后在代理对象中实现横切关注点的织入。动态代理可以分为JDK动态代理（基于接口）和 CGLIB 动态代理（基于类）。
2. 编译时织入：在编译阶段，通过修改字节码实现 AOP。AspectJ 的编译时织入就是这种方式。
3. 类加载时织入：在类加载阶段，通过修改字节码实现 AOP。AspectJ 的加载时织入就是这种方式。

AOP的实现方式取决于具体需求和技术选型。对于 Spring 应用来说，通常可以使用 Spring AOP 满足大部分需求，如果需要更强大的 AOP 功能，可以考虑使用 AspectJ。

鱼友的精彩回答

Antony 的回答

AOP：Aspect Orlented Programming 面向切面编程；

AOP的作用：

下面两点是同一件事的两面，好比硬币的两面；

简化代码：把方法中固定位置的重复的代码抽取出来，让被抽取的方法更专注于自己的核心功能，提高了内聚性；
代码增强：把特定的功能封装到切面类中，看哪里有需要就往上进行套用，被套用了切面逻辑的方法就被切面给增强了；

实现方法如下：

动态代理（InvocationHandler）：JDK原生的实现方式，需要被代理的目标类必须实现接口。因为这个技术要求**代理对象和目标对象实现同样的接口**（兄弟两个拜把子模式）。
cglib：通过**继承被代理的目标类**（认干爹模式）实现代理，所以不需要目标类实现接口。
AspectJ：本质上是静态代理，**将代理逻辑“织入”被代理的目标类编译得到的字节码文件**，所以最终效果是动态的。weaver 就是织入器。Spring 只是借用了 AspectJ 中的注解。

Gianhing 的回答

AOP：面向切面编程，是一种编程思想，将一些与核心业务无关的公共逻辑（日志、事务等）抽离出来，在不改变原有业务逻辑上，以切面的方式对程序进行一个横向的扩展，提高代码的可重用性、可维护性和可扩展性。

AOP 的相关概念：

概念	含义
目标对象（Target Object）	被切面织入的对象
连接点（Join Point）	可以被切面插入的地方，如方法调用、异常抛出等，所有方法都可以是连接点
切入点（Pointcut）	被切面增强的连接点，即哪些连接点会触发切面的通知
通知（Advice）	切面在连接点处执行的代码，它包括了前置通知、后置通知、环绕通知、异常通知和最终通知
切面（Aspect）	横切关注点的模块化，即通知 + 切入点
织入（Weaving）	将切面应用到目标对象的过程，可以在编译期、类加载期和运行期进行
引入（Introduction）	为目标对象动态的添加属性和方法

Spring AOP 和 AspectJ AOP 是 AOP 的两种实现方式：

- Spring AOP 是基于动态代理实现的，通过 JDK 动态代理或 CGLIB 代理来生成代理对象，只能用于方法级别的拦截和处理，在运行时织入
- AspectJ AOP 是一个更加完整的 AOP 框架，基于字节码操作，支持更多的切面表达式和切面类型，可以在编译期、类加载期进行织入，性能更加高效

实现 AOP 的方式：

- 静态代理：手动编写代理类，实现增强逻辑。需要为每个目标类编写代理类，代码量大
- JDK 动态代理：利用反射机制生成代理类，只能用于实现了接口的类
- CGLIB 动态代理：利用字节码生成目标类的子类（即代理类），可以代理未实现接口的类
- AspectJ AOP：使用 AspectJ 在编译期或类加载期对目标类进行织入

HeiHei 的回答

AOP：

IoC 使软件组件解耦合。AOP 可以捕捉系统中经常使用的功能，把它转化成组件。将核心业务无关的代码独立的抽取出来，形成一个独立的组件，然后以**横向交叉**的方式应用到业务流程当中的过程被称为 AOP。**简单的说就是在不改变方法源代码的基础上对方法进行功能增强。**AOP 底层使用的就是**动态代理**来实现的

Spring AOP 和 AspectJ AOP 的区别：

Spring AOP 属于**运行时增强**，主要具有如下特点：

基于动态代理来实现，默认如果使用接口的，用 JDK 提供的动态代理实现，如果是方法则使用 CGLIB 实现
Spring AOP 需要依赖 IOC 容器来管理，并且只能**作用于 Spring 容器，使用纯 Java 代码实现**
在性能上，由于 Spring AOP 是基于**动态代理**来实现的，在容器启动时需要生成代理实例，在方法调用上也会增加栈的深度，使得 Spring AOP 的性能不如 AspectJ 的那么好。
Spring AOP 致力于解决企业级开发中最普遍的 AOP(方法织入)。

AspectJ 是一个易用的功能强大的 AOP 框架，**属于编译时增强**， 可以单独使用，也可以整合到其它框架中，是 AOP 编程的完全解决方案。AspectJ 需要用到单独的编译器 ajc。

AspectJ 属于**静态织入**，通过修改代码来实现，在实际运行之前就完成了织入，所以说它生成的类是没有额外运行时开销的，一般有如下几个织入的时机：

1. 编译期织入（Compile-time weaving）：如类 A 使用 AspectJ 添加了一个属性，类 B 引用了它，这个场景就需要编译期的时候就进行织入，否则没法编译类 B。
2. 编译后织入（Post-compile weaving）：也就是已经生成了 .class 文件，或已经打成 jar 包了，这种情况我们需要增强处理的话，就要用到编译后织入。
3. 类加载后织入（Load-time weaving）：指的是在加载类的时候进行织入，要实现这个时期的织入，有几种常见的方法

SpringAOP	AspectJ
在纯Java中实现	用Java编程语言扩展实现
编译器javac	一般需要ajc
只可运行时织入	支持编译时、编译后、加载时织入
仅支持方法级编织	可编织字段、方法、构造函数、静态初始值等
只可在Spring管理的Bean上实现	可在所有域对象实现
仅支持方法执行切入点	支持所有切入点
比AspectJ慢很多	速度比AOP快很多
创建目标对象的代理，切面在代理中执行	执行程序前，各方面直接织入代码中

实现AOP：

1. 静态代理：通过编写代理类，将目标对象和切面结合起来，从而实现AOP。这种方式需要手动编写代理类，而且对于多个目标对象需要编写多个代理类，因此不够灵活。
2. 动态代理：通过使用 JDK 提供的 Proxy 类或者第三方库（如 CGLIB）来创建代理对象，从而在运行时动态地将切面织入到目标对象中。这种方式比静态代理更灵活，但仍然存在一些限制，例如只能代理实现了接口的类。
3. 基于注解：通过在切面类上使用注解来标记切点和通知类型，然后使用 AOP 框架（如 Spring AOP）来自动创建代理对象并将切面织入到目标对象中。这种方式可以减少手动编写代理类的工作量，同时也比较灵活，支持多种切面类型。

4. 面向切面编程框架：通过使用专门的 AOP 框架（如 AspectJ）来实现 AOP。这种方式可以提供更为灵活、高效的 AOP 支持，并且支持更多的切面类型和功能，例如使用 AspectJ 语言编写切面、进行更细粒度的控制等。

Redis 缓存

1、什么是 Redis？Redis 有哪些特点？Redis 有哪些常见的应用场景?

官方解析

Redis（Remote Dictionary Server）是一个开源的高性能键值存储系统，也被称为数据结构服务器。它支持多种类型的数据结构，如字符串、哈希、列表、集合、有序集合等，并提供了丰富的操作这些数据结构的命令。

Redis的特点包括：

- **高性能**：Redis使用内存来存储数据，并且数据存储在单一的进程中，因此速度非常快。
- **多样的数据类型**：Redis 支持多种数据结构，包括字符串、哈希、列表、集合、有序集合等。
- **持久化**：Redis 支持多种持久化方式，包括 RDB 快照和 AOF 日志。
- **分布式**：Redis 支持分布式部署，可以将数据分布在多个节点上。
- 简单易用：Redis 提供了丰富的命令，使得操作数据非常方便。
  Redis 的常见应用场景包括：
- **缓存**：Redis 可以作为缓存使用，加速数据读取和响应速度。
- **消息队列**：Redis 提供了列表和发布/订阅功能，可以用来实现消息队列。
- **计数器**：Redis 的计数器功能非常高效，可以用来实现页面访问量、点击量等的计数。
  排行榜：Redis 的有序集合功能可以用来实现排行榜。
  分布式锁：Redis 可以用来实现分布式锁，保证多个进程之间的互斥访问。
  实时数据分析：Redis 可以作为实时数据分析的缓存层，加速数据分析速度。

总之，Redis 具有**高性能、多样的数据类型、分布式、简单易用**等特点，可以应用于各种场景，特别适合用来**解决读写频繁**的问题。

鱼友的精彩回答

Gianhing 的回答

Redis 是基于内存的键值型（key - value）的 NoSQL 数据库（非关系型数据库）。key 一般是 String 类型，而 value 支持丰富的数据类型，包括String、Hash、List、Set、SortedSet 这五种基本类型，此外还有 GEO、BitMap、HyperLogLog 等其他类型。

Redis 有哪些特点?

- 读写性能优异
  - 基于内存，内存的访问速度是比磁盘快很多的
  - 采用单线程模型，不存在多线程的上下文切换，不需要考虑锁的问题

    - 使用 IO 多路复用模型，让 Redis 不需要创建额外的线程来监听客户端的大量请求，减少性能的消耗
    - 内置了多种优化过的数据结构实现
- 所有操作命令都是原子性的
- 支持事务
    - 允许多个命令按顺序执行并不会被打断
    - 不支持回滚
- 支持数据持久化
    - RDB：通过创建快照来获得存储在内存里面的数据在某个时间点上的副本
    - AOF：执行完更改数据的命令后，会将该命令记录到日志中
- 支持分布式部署

Redis 有哪些常见的应用场景?

- 数据缓存
    - 将热点数据缓存到 Redis，提高数据读取和系统响应速度。
    - 将用户凭证（如 token）存入 Redis，实现单点登录。
- 分布式锁
    - 利用 Redis 的 setnx 命令实现互斥，对于 setnx 命令，如果 key 不存在则会设置 key 的值并返回 1，否则直接返回 0。
    - 因此，可以通过 setnx 命令去尝试获取锁，获取锁成功才能继续执行相应业务。
    - 此外，还得给锁设置一个过期时间，防止系统出现问题导致锁无法释放。

使用命令为：

```
SET key value EX expire NX # 使用单命令保证设置指定 key 的值和过期时间是一个原子操作
```

如果需要更复杂的需求，可以采用 Redisson，它里面提供了多种分布式锁的实现。

- 限时业务的运用
    - Redis 中可以使用 expire 命令设置一个键的生存时间，到时间后 Redis 会删除它。利用这一特性可以运用在限时的优惠活动信息、手机验证码等业务场景。
- 全局唯一 ID
    - Redis 由于 incrby 命令可以实现原子性的递增，所以可以用于唯一分布式序列号的生成。
- 消息队列
    - 基于 List 实现：利用 LPUSH 和 RPOP；
    - 基于 PUB/SUB 实现；
    - 基于 Stream 实现。
- 共同关注
    - 由于 Set 支持交集、并集、差集等功能，可以实现共同关注、共同爱好等功能。
- 排行榜

    - 由于 SortedSet 是支持排序的，可以用于各种排行榜的场景。
- 统计活跃用户
    - 利用 BitMap，以日期为 key，用户 ID 为 offset，当日活跃过就设置为 1。
- 页面统计 UV
    - 将访问指定页面的每个用户 ID 添加到 HyperLogLog 中，调用 PFCOUNT 获取。

yes.的补充

因为我的项目中也有使用到Redis，在这里大胆猜测一下面试官会问到： **在你的项目中是怎么使用Redis的?**

1. 在我的项目中使用 Redis 配合 Spring 的 redis-session，将用户的登录态存入 redis，实现单点登录。
2. 对于系统的首页，为了加快响应速度，设置定时任务将首页数据缓存到 Redis中，避免了首页加载过慢用户体验差的情况。
3. 对于用户加入队伍的限制，使用 Redisson 作分布式锁，防止了用户恶意请求加入超出限制数量的队伍。

2、讲一下 Redis 的单线程模型，IO 多路复用是什么?

官方解析

Redis 是一款基于内存的高性能键值存储系统，采用**单线程模型**的设计。在 Redis 中，所有客户端的请求都是由一个单线程来处理的，这个单线程不断地从客户端套接字中读取命令请求，并将命令请求放入一个请求队列中。接着，Redis 的事件处理器会按照一定的规则选择一个请求进行处理，处理完之后将响应结果返回给客户端。

单线程模型的优点是可以**避免多线程并发访问共享数据时的竞争和死锁问题**，简化了系统的设计和调试。此外，由于 Redis 的内存访问速度非常快，因此单线程处理请求也能够保证足够的性能。

IO 多路复用是指在一个线程中同时监听多个文件描述符，一旦某个文件描述符就绪，就立即处理对应的事件。在 Redis 中，采用的是基于 **epol**l 的 IO 多路复用技术，可以实现高效的事件监听和响应。

在 Redis 中，客户端的请求是由一个单线程来处理的，而 IO 操作却是通过 epoll 多路复用技术实现的。这种设计方式既能充分利用 CPU 的计算能力，又能够保证足够的 IO 处理能力，从而实现了高效的键值存储服务。

> 鱼皮补充：这题的另一个问法是 “Redis 为什么快？” 建议大家通过画图理解 IO 多路复用

鱼友的精彩回答

林风的回答

Redis实现的单线程并不是一个纯粹的单线程，所谓的单线程指的是**Redis在解析客户端发送的命令，处理这样的请求使用的是单线程**。

在Redis2.6版本会有后台线程来处理**文件关闭、AOF刷盘**

Redis 在 4.0 版本之后，新增了一个新的后台线程，用来异步释放 Redis 内存，也就是 lazyfree 线程。

那为什么 Redis 使用的是单线程但是性能还是这么高呢?

1. 避免了高并发竞争带来的消耗
2. 采用**多路复用**机制

IO 多路复用机制指的是一个进程可以处理多个不同任务。

无名的回答

在Redis 6.0以前，Redis的核心网络模型选择用单线程来实现。

对于一个 DB 来说，CPU 通常不会是瓶颈，因为大多数请求不会是 CPU 密集型的，而是 I/O 密集型。具体到Redis的话，如果不考虑 RDB/AOF 等持久化方案，Redis是完全的纯内存操作，执行速度是非常快的，因此这部分操作通常不会是性能瓶颈，Redis真正的性能瓶颈在于网络 I/O，也就是客户端和服务端之间的网络传输延迟，因此Redis选择了单线程的 I/O 多路复用来实现它的核心网络模型。

实际上更加具体的选择单线程的原因如下：

- **避免过多的上下文切换开销**：如果是单线程则可以规避进程内频繁的线程切换开销，因为程序始终运行在进程中单个线程内，没有多线程切换的场景。
- **避免同步机制的开销**：如果 Redis 选择多线程模型，又因为 Redis 是一个数据库，那么势必涉及到底层数据同步的问题，则必然会引入某些同步机制，比如锁，而我们知道 Redis不仅仅提供了简单的 key-value 数据结构，还有 list、set 和 hash 等等其他丰富的数据结构，而不同的数据结构对同步访问的加锁粒度又不尽相同，可能会导致在操作数据过程中带来很多加锁解锁的开销，增加程序复杂度的同时还会降低性能。
- **简单可维**护：如果 Redis使用多线程模式，那么所有的底层数据结构都必须实现成线程安全的，这无疑又使得Redis的实现变得更加复杂。

总而言之，Redis选择单线程可以说是多方博弈之后的一种权衡：在保证足够的性能表现之下，使用单线程保持代码的简单和可维护性。

IO多路复用

IO 多路复用是指内核一旦发现进程指定的一个或者多个 IO 条件准备读取，它就通知该进程。

IO多路复用适用如下场合：

- 当客户处理多个描述字时（一般是交互式输入和网络套接口），必须使用I/O复用。
- 当一个客户同时处理多个套接口时，而这种情况是可能的，但很少出现。
- 如果一个TCP服务器既要处理监听套接口，又要处理已连接套接口，一般也要用到I/O复用。
- 如果一个服务器即要处理TCP，又要处理UDP，一般要使用I/O复用。
- 如果一个服务器要处理多个服务或多个协议，一般要使用I/O复用。
- 与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。

3、Redis 基础类型中的 String 底层实现是什么?

官方解析

Redis 中的 String 类型底层实现是一个**简单动态字符串（SDS，Simple Dynamic String）**，也就是字符串动态增长实现的一种方式。

SDS 是 Redis 自己实现的字符串库，相对于 C 语言原生的字符串，SDS **在空间使用上更加灵活**，而且支持 **O(1) 复杂度的长度计算**，避免了 C 语言字符串计算长度时的 O(N) 时空复杂度问题。

SDS 是一种**动态字符串**，它有如下特点：

1. 首先 SDS 对内存的分配和释放进行了封装，使得**字符串的空间可以根据需要进行增长或缩减**，避免了 C 语言字符串需要手动分配空间的问题。
2. SDS 除了记录字符串本身的长度外，还记录了**分配给字符串的空间的长度**，可以方便地计算出字符串是否需要扩容。
3. SDS 使用了惰性空间释放，不会在空间缩减时立即释放空间，而是等到需要扩容时再重新分配内存。
4. SDS 提供了**字符串追加操作**，可以在 **O(1)** 的时间内完成追加操作。

在 Redis 中，String 类型并不仅仅只是字符串类型，它还支持一些其他的操作，如递增/递减操作、位运算等，这些操作都是基于 SDS 底层实现的。

> 作者：编程导航知识星球 + 星球鱼友们

鱼友的精彩回答

林风的回答

> 整理于《Redis设计与实现》

String 底层实现是 **SDS**，也就是**动态字符串**。

Redis当中的动态字符串主要是对C语言中的做了一个封装，使得SDS具有**动态扩容、O(1)复杂度的长度计算**的特点。

并且避免了C字符串缓冲区溢出（C字符串不记录自身长度）。

当 SDS API 需要对 SDS 进行修改时，API 会先检查 SDS 的空间是否满足修改所需的要求，如果不满足的话，API 会自动将 SDS 的空间扩展至执行修改所需的大小

SDS主要由三个部分组成：

- len 实际使用长度
- free buf数组中未使用字节的数量
- char buf[] 用于保存字符串

```
struct sdshdr {
    //记录buf数组中已使用字节的数量
    //等于SDS所保存字符串的长度
    int len;
    //记录buf数组中未使用字节的数量
    int free;
    //字节数组，用于保存字符串
    char buf[];
};
```

C 字符串	SDS
获取字符串长度的复杂度为 $O(N)$	获取字符串长度的复杂度为 $O(1)$
API 是不安全的，可能会造成缓冲区溢出	API 是安全的，不会造成缓冲区溢出
修改字符串长度 N 次必然需要执行 N 次内存重分配	修改字符串长度 N 次最多需要执行 N 次内存重分配
只能保存文本数据	可以保存文本或者二进制数据
可以使用所有 <string.h> 库中的函数	可以使用一部分 <string.h> 库中的函数

SDS的主要操作API

函　数	作　用	时间复杂度
sdsnew	创建一个包含给定 C 字符串的 SDS	$O(N)$，N 为给定 C 字符串的长度
sdsempty	创建一个不包含任何内容的空 SDS	$O(1)$
sdsfree	释放给定的 SDS	$O(N)$，N 为被释放 SDS 的长度
sdslen	返回 SDS 的已使用空间字节数	这个值可以通过读取 SDS 的 len 属性来直接获得，复杂度为 $O(1)$
sdsavail	返回 SDS 的未使用空间字节数	这个值可以通过读取 SDS 的 free 属性来直接获得，复杂度为 $O(1)$
sdsdup	创建一个给定 SDS 的副本（copy）	$O(N)$，N 为给定 SDS 的长度
sdsclear	清空 SDS 保存的字符串内容	因为惰性空间释放策略，复杂度为 $O(1)$
sdscat	将给定 C 字符串拼接到 SDS 字符串的末尾	$O(N)$，N 为被拼接 C 字符串的长度
sdscatsds	将给定 SDS 字符串拼接到另一个 SDS 字符串的末尾	$O(N)$，N 为被拼接 SDS 字符串的长度
sdscpy	将给定的 C 字符串复制到 SDS 里面，覆盖 SDS 原有的字符串	$O(N)$，N 为被复制 C 字符串的长度
sdsgrowzero	用空字符将 SDS 扩展至给定长度	$O(N)$，N 为扩展新增的字节数
sdsrange	保留 SDS 给定区间内的数据，不在区间内的数据会被覆盖或清除	$O(N)$，N 为被保留数据的字节数
sdstrim	接受一个 SDS 和一个 C 字符串作为参数，从 SDS 中移除所有在 C 字符串中出现过的字符	$O(N^2)$，N 为给定 C 字符串的长度
sdscmp	对比两个 SDS 字符串是否相同	$O(N)$，N 为两个 SDS 中较短的那个 SDS 的长度

haha 的回答

Redis 中的任意数据类型的键和值都会被封装成一个 RedisObject，即 Redis 对象。会在 Redis 对象中主要由三个字段：**当前数据结构的类型、编码方式和指向实际数据的指针**。

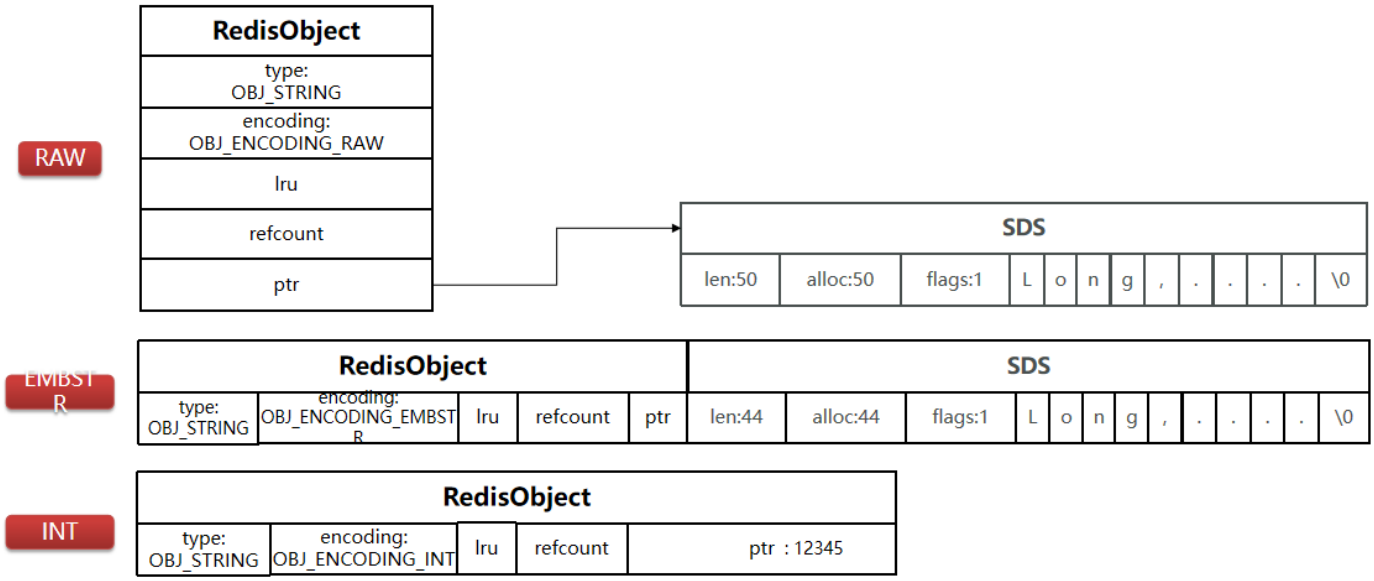
String类型，首先会在Redis对象中指明他的类型为**OBJ_STRING**。

String类型有三种编码方式：

1、其最基本的编码方式是 RAW 编码，基于简单动态字符串SDS来实现，存储上限为512mb。他就是在内存中申请一个 SDS，然后让 Redis 对象的指针指向 SDS 即可。

2、当要存储的 SDS 的长度小于 44 字节，就会使用 EMBSTR 编码，此时 Redis 对象的头部和 SDS 是在一个连续的内存空间中，申请内存的时候只需要一次分配，效率更高。为什么是 44 个字节呢？因为Redis对象数据结构中其头部有 16 个字节。然后SDS中头部 3 个字节，尾巴结束字符一个字节，加在一起共 20 个字节，20 个字节加上 44 个字节就是 64 个字节，因为 redis 内存分配会以 2 的 n 次方进行分配，64 个字节不会产生内存碎片。

3、如果要存储的字符串是一个整数，且大小不超过 8 位二进制能表示的范围。就采用 int 的编码方式，直接让 redis 对象的指针位设置为这个整数



> 鱼皮评论：这个回答非常好，官方给的题解只讲了 SDS，其实这个回答才是比较完整

4、如何使用 Redis 实现一个排行榜?

官方解析

使用 Redis 可以很方便地实现一个排行榜，以下是一种实现方法：

使用**有序集合（Sorted Set）**来存储排行榜数据，以用户得分作为分数(score)，用户 ID 作为成员(member)。

当用户得分改变时，使用 Redis 的 **ZADD** 命令将用户的分数更新到有序集合中。

获取排行榜数据时，使用 Redis 的 ZREVRANGE 命令按分数倒序获取有序集合中的成员。

可以使用 Redis 的 ZSCORE 命令获取某个用户的分数，或使用 ZREVRANK 命令获取某个用户的排名。

以下是一个简单的 Node.js 实现：

```
const redis = require('redis');
const client = redis.createClient();

// 将用户得分更新到排行榜中
function updateScore(userId, score) {
  client.zadd('leaderboard', score, userId, (err, res) => {
    if (err) {
      console.error('Error updating score:', err);
    }
  });
}

// 获取排行榜数据
function getLeaderboard() {
  return new Promise((resolve, reject) => {
    client.zrevrange('leaderboard', 0, -1, 'WITHSCORES', (err, res) => {
      if (err) {
        reject(err);
      } else {
        const leaderboard = [];
        for (let i = 0; i < res.length; i += 2) {
          leaderboard.push({
            userId: res[i],
            score: parseInt(res[i + 1])
          });
        }
        resolve(leaderboard);
      }
    });
  });
}

// 获取某个用户的分数
function getScore(userId) {
  return new Promise((resolve, reject) => {
    client.zscore('leaderboard', userId, (err, res) => {
      if (err) {
        reject(err);
      } else {
        resolve(parseInt(res));
      }
    });
  });
}

// 获取某个用户的排名
function getRank(userId) {
  return new Promise((resolve, reject) => {
    client.zrevrank('leaderboard', userId, (err, res) => {
```

```
      if (err) {
        reject(err);
      } else {
        resolve(res === null ? null : res + 1);
      }
    });
  });
}

// 示例代码
updateScore('user1', 100);
updateScore('user2', 200);
updateScore('user3', 300);

getLeaderboard().then((leaderboard) => {
  console.log('Leaderboard:', leaderboard);
});

getScore('user1').then((score) => {
  console.log('User1 score:', score);
});

getRank('user1').then((rank) => {
  console.log('User1 rank:', rank);
});
```

在实际应用中，还可以根据需要进行排行榜数据的缓存和更新策略等优化。

鱼友的精彩回答

爱吃鱼蛋的回答

Redis 实现排行榜是 Redis 中一个很常见的场景，主要使用的是 **ZSet** 进行实现，下面是为什么选用 ZSet：

- **有序性**：排行榜肯定需要实现一个排序的功能，在 Redis 中有序的数据结构有 List 和 ZSet；
- **支持分数操作**：ZSet 可以对集合中的元素进行增删改查操作，十分贴合排行榜中用户分数动态变化的场景，而 List 并不能针对分数进行操作，只有其中的 value 进行操作；
- **支持范围查询**：ZSet 可以按照分数进行范围查询，如排行榜中的 Top10 需求就可通过该特性进行实现；
- 支持去重：由于 ZSet 属于 Set 的特殊数据结构，因此同样拥有 Set 不可重复的特性，对于排行榜中不可出现重复项的需求也十分贴合，而 List 只能手动去重。

因此选择ZSet实现排行榜相对于List实现会更合适和高效。

以学生成绩排行为例，下面是使用Redis命令实现

```
# 添加示例数据
ZADD scores 90 "张三"
ZADD scores 85 "李四"
```

```
ZADD scores 95 "王五"
ZADD scores 92 "赵六"
# 查询排名前3的学生信息
ZRANGE scores 0 2 WITHSCORES
# 查询排名前3的打印
1) "王五"
2) "95"
3) "赵六"
4) "92"
5) "张三"
6) "90"
# 删除学生“李四”的成绩信息
ZREM scores "李四"
```

下面是SpringBoot整合Redis进行实现

```
// 添加学生成绩
public void addScore(String name, int score) {
    redisTemplate.opsForZSet().add("scores", name, score);
}

// 查询排名前N的学生成绩
public List<Map.Entry<String, Double>> getTopScores(int n) {
    return redisTemplate.opsForZSet().reverseRangeWithScores("scores", 0, n - 1)
            .stream()
            .map(tuple -> new AbstractMap.SimpleEntry<>(tuple.getValue(),
tuple.getScore()))
            .collect(Collectors.toList());
}

// 删除某个学生的成绩
public void removeScore(String name) {
    redisTemplate.opsForZSet().remove("scores", name);
}
```

5、Redis 的持久化机制有哪些？说说各自的优缺点和应用场景?

官方解析

Redis 的持久化机制主要分为 **RDB** 和 **AOF** 两种。

RDB 持久化机制

RDB 持久化机制是指将 Redis 在内存中的数据以**快照**的形式写入磁盘中，可以手动或自动执行快照操作，**将数据集的状态保存到一个 RDB 文件中**。

RDB 机制的优点在于：

- RDB 机制**适合在数据集比较大时进行备份操作**，因为它可以**生成一个非常紧凑、经过压缩的数据文件**，对于**备份、恢复、迁移数据**都很方便。

> RDB 机制在 Redis 重启时比 AOF 机制更快地将 Redis 恢复到内存中。

RDB 机制的缺点在于：

- RDB 机制可**能会出现数据丢失**，因为**数据是周期性地进行备份**，一旦 Redis 出现问题并且上一次备份之后还没有进行过数据变更，那么这部分数据将会丢失。
- RDB 机制会造成一定的 **IO 压力**，当数据集比较大时，进行备份操作可能会阻塞 Redis 服务器进程。

AOF 持久化机制

AOF 持久化机制是指**将 Redis 在内存中的操作命令以追加的形式写入到磁盘中的 AOF 文件**，AOF 文件记录了 Redis 在内存中的操作过程，只要在 Redis 重启后重新执行 AOF 文件中的操作命令即可将数据恢复到内存中。

AOF 机制的优点在于：

- AOF 机制比 RDB 机制更加**可靠**，因为 AOF 文件**记录了 Redis 执行的所有操作命令**，可以确保数据不丢失。
- AOF 机制在**恢复大数据集时更加稳健**，因为 AOF 文件记录了数据的操作过程，可以确保每一次操作都被正确地执行。

AOF 机制的缺点在于：

- AOF 机制生成的 **AOF 文件比 RDB 文件更大**，当数据集比较大时，AOF 文件会比 RDB 文件占用更多的磁盘空间。
- AOF 机制对于**数据恢复的时间比 RDB 机制更加耗时**，因为要重新执行 AOF 文件中的所有操作命令。

综上所述，RDB 适合用于**数据集较大、备份、恢复数据和迁移数据**等场景，AOF 适合用于**数据可靠性要求高、数据恢复稳健**等场景。

鱼友的精彩回答

Starry 的回答

Redis 的持久化机制主要有两种：RDB（Redis Database）和 AOF（Append Only File）。

1、RDB 持久化

RDB 持久化机制会在指定的时间间隔内对数据集做快照，将快照存储到磁盘中。

RDB 的优缺点：对于**数据恢复速度比较快**，同时**存储的文件也比 AOF 小**，缺点是可能会出现**数据丢失**的情况，因为**快照的时间间隔越长，数据丢失的可能性就越大**。

2、AOF 持久化

AOF 持久化机制则**会将每一次的写操作都追加到一个文件中**，这个文件就是 AOF 文件。

AOF 文件保存了 Redis 服务器接收到的所有写命令，当 Redis 重启时，就可以使用 AOF 文件重建数据集。

AOF 的优缺点：**数据丢失的可能性比 RDB 低**，缺点是**文件比 RDB 大，恢复速度比较慢**。

3、RDB + AOF 混合使用

除了 RDB 和 AOF 外，Redis 还支持**混合持久化**机制，即同时使用 RDB 和 AOF 两种持久化机制，以此来发挥它们的优点。

在混合持久化机制中，使用 **AOF 持久化机制保存每次写操作**，**使用 RDB 持久化机制保存指定时间点的数据快照**，以此来达到**数据恢复速度**和**数据完整性的平衡**。

应用场景：

- RDB 适合用于**数据集比较大**，**数据恢复速度比较快**的场景，例如**备份**、**灾难恢复**等；
- AOF 适合用于**数据集比较小**，**数据完整性比较重要**的场景，例如**金融**、**电商**等行业。

李狗蛋的回答

Redis的持久化机制

Reids的持久化机制主要有两种，一种是 RDB 快照，指使用**快照**的形式将**内存中的全部数据**写入到磁盘中，存的是非常紧凑经过压缩的数据文件。一种是 AOF 日志，指以**追加**的形式，**一条一条地将操作命令**写入到 AOF 日志中。

优缺点：

- 占用空间方面：
  - RDB存的是压缩过的数据集，占用空间更小
  - AOF存的是每一条操作命令，占用空间更大
- 速度方面：
  - RDB存储慢：周期性备份，每次都是内存中的全部数据，但恢复时更快
  - AOF存储快：追加的形式一条条地写入，每次写入一条指令，但恢复时更慢（需执行全部指令）
- 安全性方面：
- RDB 是周期性备份，可能会丢失没来得及备份的数据
- AOF 根据策略而定，可以记录全部数据的操作过程，更可靠
- 资源消耗方面：
  - RDB 会造成 IO 压力，数据集比较大时可能会阻塞正常的服务进程
  - AOF 每次记录一条，消耗资源更低

场景：

- 关闭持久化：数据不敏感，且可以从其他地方重新生成找回。
- RDB：数据集较大、备份、恢复数据和迁移数据等场景，可以承受数分钟级的数据丢失。
- AOF：适合用于数据可靠性要求高、数据恢复稳健等场景。

补充：

- AOF 的启动优先级比 RDB 高

4.0 后，多了一种混合持久化，前半段时 RDB 的全量数据，后半段是 AOF 的增量数据

优点是充分发挥了二者的优点，RDB 提供加载速度，AOF 提供数据可靠性保障

缺点是兼容性差，4.0 之前的版本不兼容，并且前半段的 RDB 可读性较差

追问：

生成 RDB 期间，Redis 可以同时处理写请求吗?

回答：

可以，Redis 使用操作系统的**多进程写时复制技术 COW（Copy On Write）**来实现快照持久化，保证数据一致性。

在持久化时，**会调用 glibc 的函数 fork 产生一个子进程，快照持久化完全交给子进程处理，父进程继续处理客户端请求**。当主进程执行写指令修改数据的时候，这个数据就会复制一份副本，子进程读取这个副本数据写到 RDB 文件。既保证了快照的完整性，也允许主进程同时对数据进行修改，避免了对正常业务的影响。

林寻的回答

Redis 支持 RDB 持久化、AOF 持久化、RDB-AOF 混合持久化这三种持久化方式。

RDB：

RDB(Redis Database)是 Redis 默认采用的持久化方式，它以**快照**的形式将进程数据持久化到硬盘中。

RDB会创建一个**经过压缩的二进制文件**，文件以“.rdb”结尾，内部存储了**各个数据库的键值对数据**等信息。

RDB持久化的触发方式有两种：

- **手动触发**：通过 SAVE 或 BGSAVE 命令触发 RDB 持久化操作，创建“.rdb”文件；
- 自动触发：通过配置选项，让服务器在满足指定条件时自动执行 BGSAVE 命令。

其中，SAVE 命令执行期间，Redis 服务器将阻塞，直到“.rdb”文件创建完毕为止。

而 BGSAVE 命令是异步版本的 SAVE 命令，它会使用 Redis 服务器进程的子进程，创建“.rdb”文件。BGSAVE 命令在创建子进程时会存在短暂的阻塞，之后服务器便可以继续处理其他客户端的请求。总之，BGSAVE 命令是针对 SAVE 阻塞问题做的优化，Redis 内部所有涉及 RDB 的操作都采用 BGSAVE 的方式，而 SAVE 命令已经废弃！

RDB 持久化的优缺点如下：

- 优点：RDB 生成紧凑压缩的二进制文件，体积小，使用该文件恢复数据的速度非常快；
- 缺点：BGSAVE 每次运行都要执行 fork 操作创建子进程，属于重量级操作，不宜频繁执行，所以 RDB 持久化没办法做到实时的持久化。

AOF：

AOF（Append Only File），解决了数据持久化的实时性，是目前 Redis 持久化的主流方式。AOF 以独立日志的方式，记录了每次写入命令，重启时再重新执行 AOF 文件中的命令来恢复数据。AOF 的工作流程包括：命令写入（append）、文件同步（sync）、文件重写（rewrite）、重启加载（load）

AOF持久化的优缺点如下：

- 优点：与 RDB 持久化可能丢失大量的数据相比，AOF 持久化的安全性要高很多。通过使用 everysec 选项，用户可以将数据丢失的时间窗口限制在1秒之内。
- 缺点：AOF 文件存储的是协议文本，它的体积要比二进制格式的”.rdb”文件大很多。AOF 需要通过执行 AOF 文件中的命令来恢复数据库，其恢复速度比 RDB 慢很多。AOF 在进行重写时也需要创建子进程，在数据库体积较大时将占用大量资源，会导致服务器的短暂阻塞。

RDB-AOF 混合持久化：

Redis 从 4.0 开始引入 RDB - AOF 混合持久化模式，这种模式是基于 AOF 持久化构建而来的。

用户可以通过配置文件中的“aof-use-rdb-preamble yes”配置项开启 AOF 混合持久化。Redis 服务器在执行 AOF 重写操作时，会按照如下原则处理数据：

- 像执行 BGSAVE 命令一样，根据数据库当前的状态生成相应的 RDB 数据，并将其写入 AOF 文件中；
- 对于重写之后执行的 Redis 命令，则以协议文本的方式追加到 AOF 文件的末尾，即 RDB 数据之后。

通过使用 RDB - AOF 混合持久化，用户可以同时获得 RDB 持久化和 AOF 持久化的优点，服务器既可以通过 AOF 文件包含的 RDB 数据来实现快速的数据恢复操作，又可以通过 AOF 文件包含的 AOF 数据来将丢失数据的时间窗口限制在 1s 之内。

6、如何用 Redis 实现分布式 Session?

官方解析

在分布式系统中，通常会将 **Session 存储在 Redis** 中来实现**分布式 Session**，这样就可以在多台服务器之间共享 Session 数据。

实现分布式 Session 的方式有多种，其中一种常用的方式是**使用 Redis 的数据结构 Hash**。具体实现步骤如下：

1. 在用户登录成功后，将 Session 数据存储在 Redis 中。
2. 将 Redis 中的 Session 数据的 Key 设置为一个全局唯一的 ID，一般使用类似于“session:token”这样的格式，其中 token 是一个随机生成的字符串，用来标识这个 Session 数据。
3. 在客户端返回响应的同时，将 Session ID（即 token）以 Cookie 的形式返回给客户端。客户端在后续的请求中都会携带这个 Cookie。
4. 在后续的请求中，服务器会从客户端传递过来的 Cookie 中获取 Session ID，然后根据这个 ID 从 Redis 中获取对应的 Session 数据。如果 Redis 中没有找到对应的 Session 数据，那么就表示这个请求无法通过认证。

- 在用户退出登录或 Session 失效时，需要将 Redis 中的对应 Session 数据删除。

可以使用 Redis 的 **EXPIRE** 命令来设置 Session 数据的过期时间，这样可以自动删除已经过期的 Session 数据。

同时，还需要注意保护 Redis 中的 Session 数据不被恶意攻击者窃取，一般可以通过设置 Session 数据的前缀和使用随机的 Session ID 等方式来提高安全性。

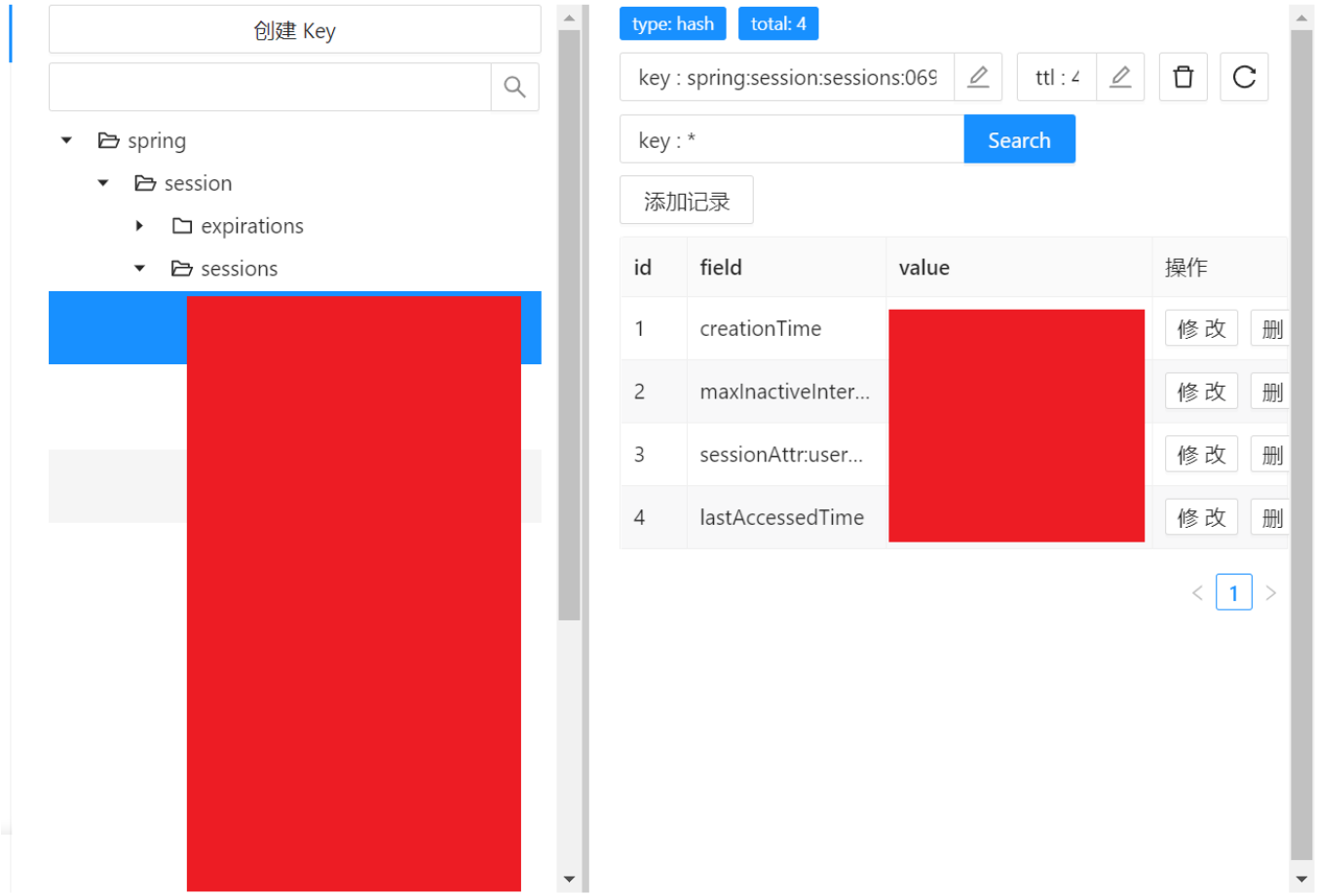
> 鱼皮的补充：这题可以结合 spring-session-data-redis 的实现去说

鱼友的精彩回答

yes.的回答

分布式 session 指在多个服务器间共享 session，我们可以使用 redis 来存储 session 来实现该功能。

在 redis 中我们通常使用 Hash 来存储 session。

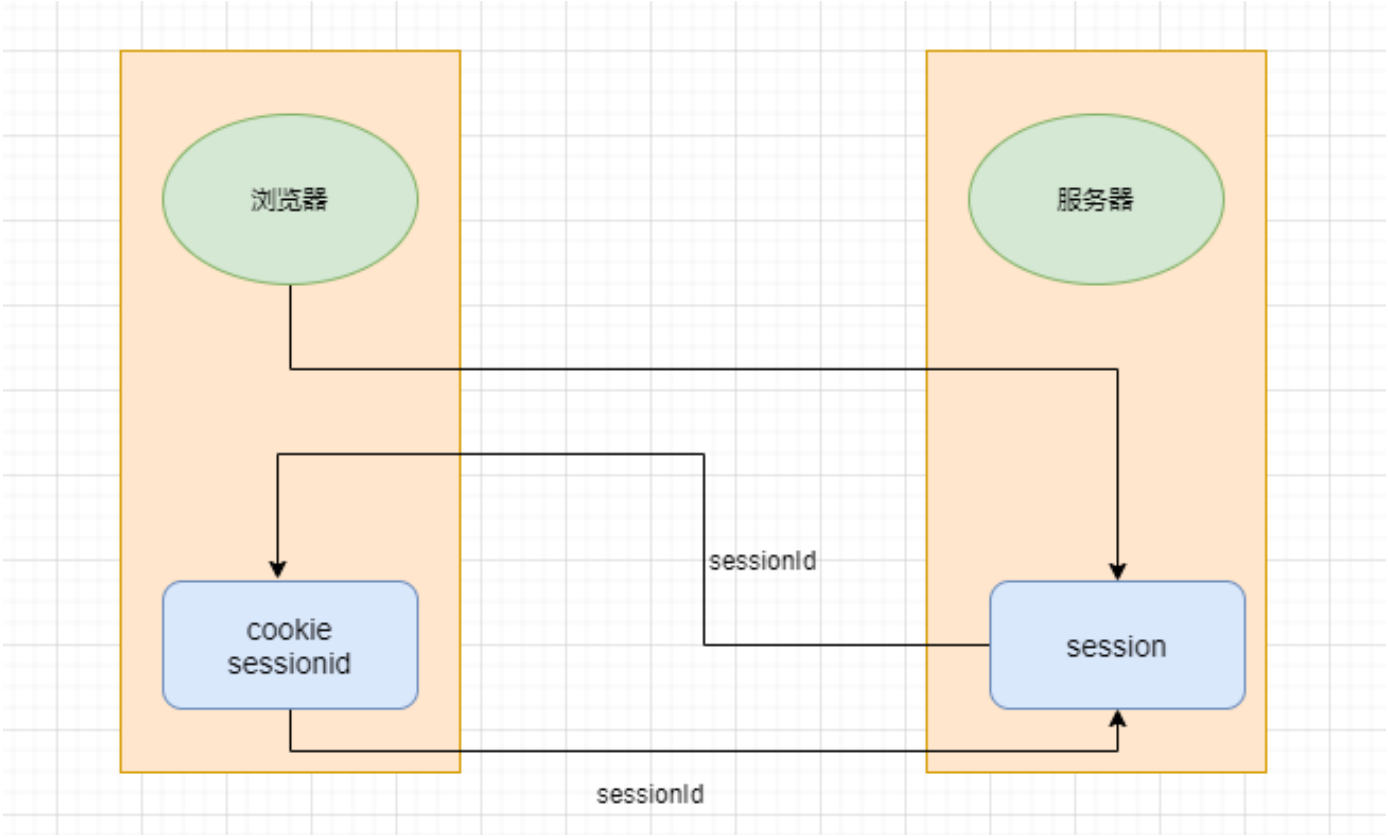


具体的步骤如下：

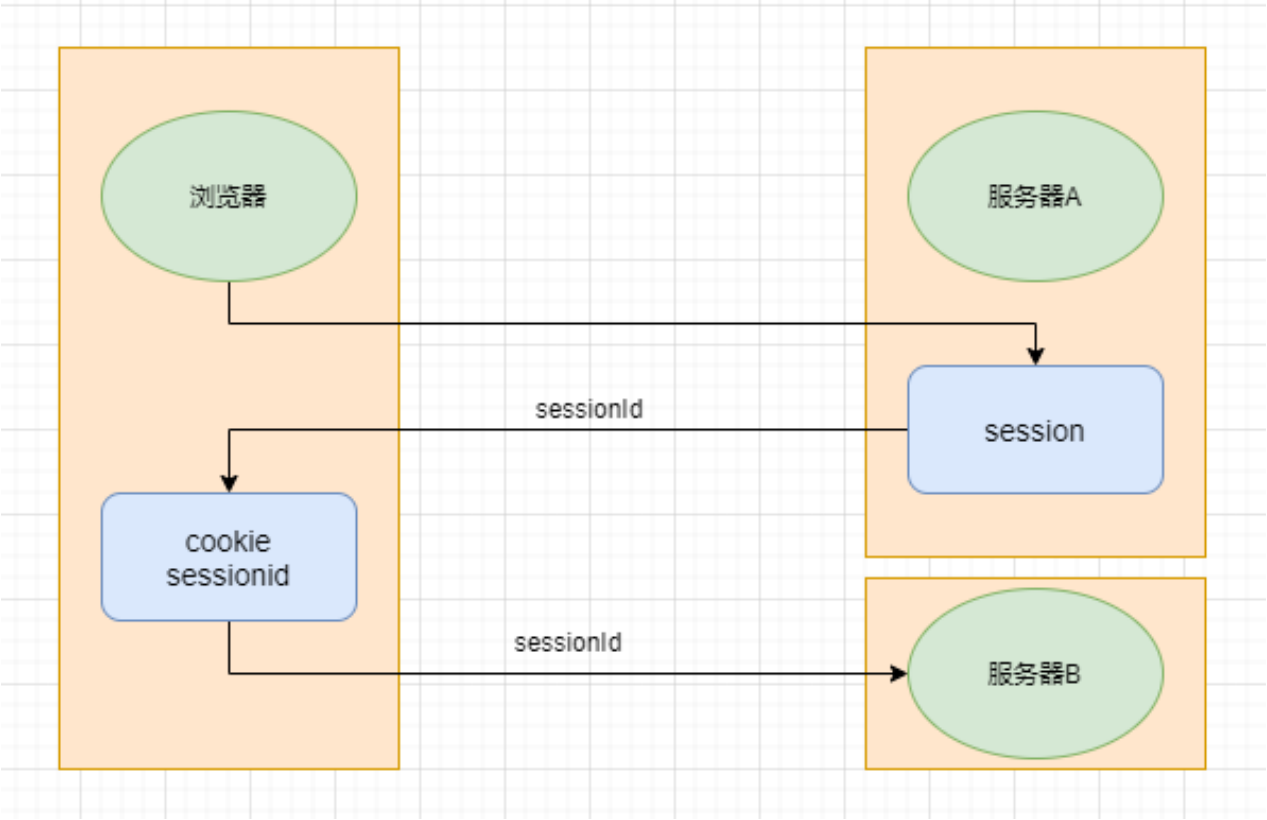
1. 用户登录成功后，将 Session 存到 redis 中
2. 将key设置为一个全局 id，格式可以采用“session:token”，其中 token 为 sessiond 的唯一标识。
3. 将 session 的唯一标识 token 以 cookie 的形式返回给客户端，客户端在后续请求中都会携带这个 cookie。
4. 后续请求中，服务器拿到客户端传来的 cookie，并根据它的值，也就是 token，去 redis 找对应的 session 数据。
5. 用户退出登录后，将 session 删除。

木木的回答

在 web 开发中，我们会把用户的登录信息存储在 session 中，而 session 是依赖于 cookie 的，即服务器创建 session 时，会自动分配一个唯一的 ID，并且的响应时创建一个 cookie 用于存储该 sessionId。当客户端收到这个 cookie 后，就会自动保存这个 sessionId，并且在下次访问时自动携带这个 sessionId。这时服务器就可以通过这个 sessionId 得到与之对应的 session，从而识别用户的身份。



目前的大部分应用，基本都是采用**分布式部署**的方式，即将应用程序部署在多台服务器上，并通过 Nginx 做统一的请求分发。但是服务器与服务器之间是相互隔离的，他们的 **session 是不共享的**，这就存在了 session 同步的问题了，如图：



如果客户端第一次访问服务器，请求被分发到了服务器 A，则服务器 A 就会为该客户端创建 session。如果客户端再次访问服务器，请求被分发到了服务器B，由于服务器B上没有这个 session，所以，用户的身份就无法得到验证，从而产生了不一致的问题。

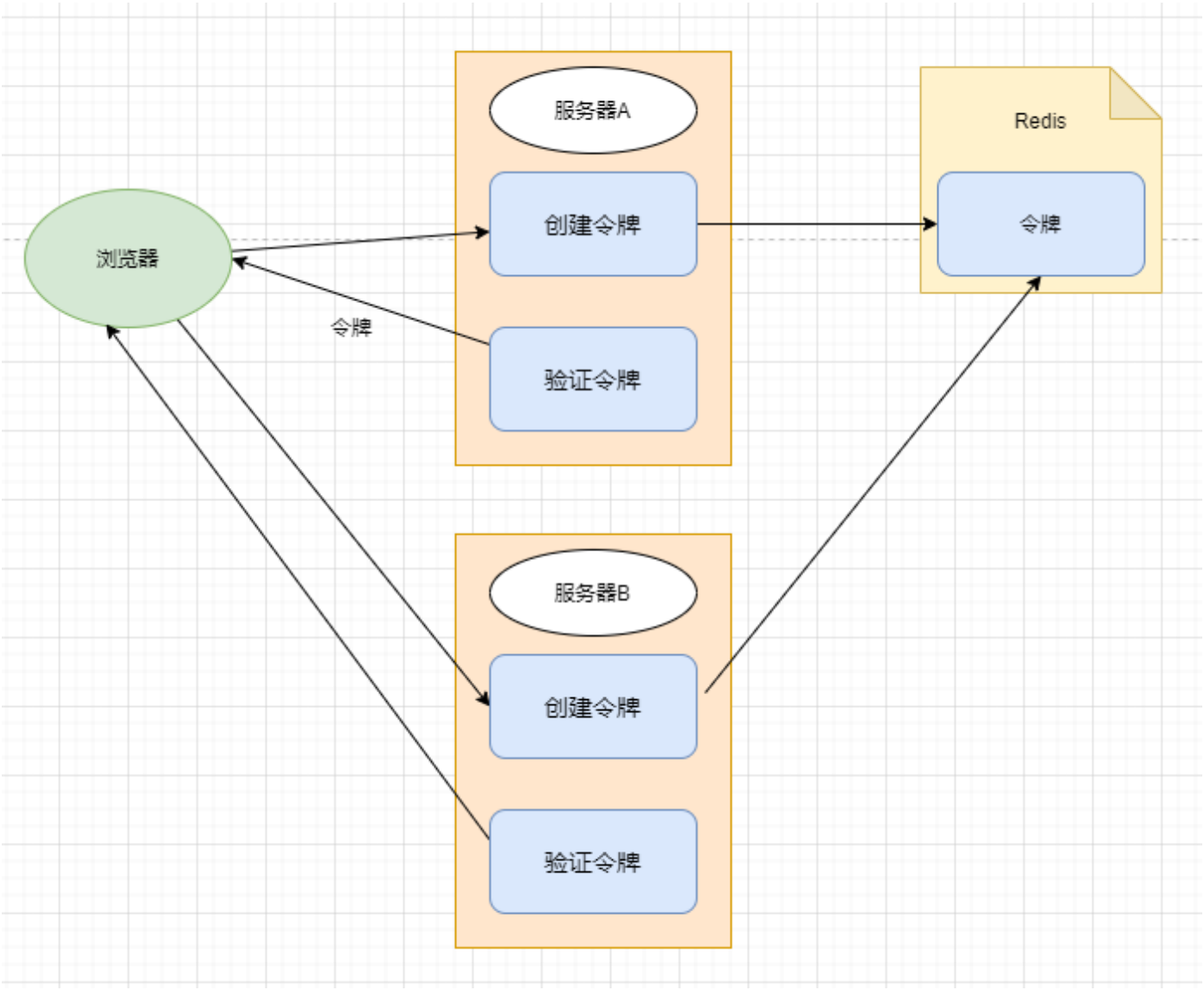
解决这个问题的办法很多，比如可以协调多个服务器，让他们的 session 保持同步，也可以在分发请求时做绑定处理，即将某一个 IP 固定分配给同一个服务器。但是这些方式都比较麻烦，而且在性能上也有一定的消耗。更加合理的方式就是使用 Redis 高性能缓存服务器，来实现分布式session

从上面的分析中，不难看出，其实要想做到多台服务器 session 共享，无非就是两件事：

- 保存用户信息
- 验证用户信息

具体实现如下：

1. **创建令牌**：用户初次访问服务器时，给他创建一个 唯一的身份标识，并且使用cookie封装这个标识后，再发送给客户端。当客户端再次访问服务器时，就回自动携带这个身份标识了，这个和单机版本的sessionId是一样的。在返回令牌之前，需要将他存储起来，以便于后续的验证。而这个令牌不能保存在服务器本地，因为多台服务器之间相互隔离。所以，这个令牌就可以使用Redis进行存储。
2. 验证令牌：用户再次访问服务器时，获取到之前的身份标识，通过Redis进行查询便知结果。



航仔丶的回答

> Java 实现

1.配置Maven依赖和Redis连接信息

添加Spring Boot和Spring Session依赖：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.session</groupId>
        <artifactId>spring-session-data-redis</artifactId>
    </dependency>
</dependencies>
```

配置Redis连接信息：

```
spring:
  redis:
    cluster:
      nodes:
        - 148.70.153.63:9426
        - 148.70.153.63:9427
        - 148.70.153.63:9428
        - 148.70.153.63:9429
        - 148.70.153.63:9430
        - 148.70.153.63:9431
    password: password
    timeout: 60000
```

2.配置 Spring Session 和 Redis

在Spring Boot应用程序的入口点上添加@EnableRedisHttpSession注释，并指定 session 的最大不活动时间：

```
@SpringBootApplication
@EnableRedisHttpSession(maxInactiveIntervalInSeconds = 3600)
public class RedisSessionApplication {
    public static void main(String[] args) {
        SpringApplication.run(RedisSessionApplication.class, args);
    }
}
```

配置Redis作为Spring Session的存储：

```
spring:
  session:
    store-type: redis
```

3.编写代码并测试

创建一个RESTful API，以获取当前Session的ID：

```
@RestController
public class SessionController {
    @RequestMapping("/getSessionId")
    public String getSessionId(HttpServletRequest request) {
        String sessionId = request.getSession().getId();
        return sessionId;
    }
}
```

在多个 Tomcat 实例中启动应用程序，并使用不同的浏览器访问 API。您应该会看到相同的 Session ID，这表明 Session 数据已在不同的Tomcat实例之间共享。

总结： 使用Redis实现分布式 Session 需要将 Session 数据存储在 Redis 中，并在多个 Tomcat 实例之间共享这些数据。可以使用 Spring Session 和 Spring Boot 自动配置来实现这一目标，并在代码中访问 Session 数据。

迷。的回答

讲一下整个 redis 实现共享 session 的业务流程：

- 在发送验证码的时候将手机号和对应验证码以 key value 形式存储到 redis 中
- 在对比验证码是否一致时，需要从 redis 里面取出手机号对应的 code
- 会使用 UUID 创建一个登录令牌 token
- 将 User 对象转为 HashMap,并与 token 令牌一起以 hash 键值对形式存储
- 设置一个 token 的有效期并返回给前端
- 设置一个新的拦截器，用于刷新 token，由于 LoginInterceptor 没有交给 Spring 进行管理，因此 StringRedisTemplate 不能通过@Resource自动注入。需要在配置文件中进行构造器注入。

```
public Result sendCode(String phone, HttpSession session) {
        //校验手机号
        //如果不符合就返回错误信息
        if (RegexUtils.isPhoneInvalid(phone)) {
            return Result.fail("手机号格式错误");
        }

        //符合就发送验证码
        //胡图工具类的使用
        String code = RandomUtil.randomNumbers(6);
        //保存验证码到redis  设置一个验证码有效时长   key-手机号 value-验证码
```

```
        stringRedisTemplate.opsForValue().set(LOGIN_CODE_KEY
+phone,code,LOGIN_CODE_TTL, TimeUnit.MINUTES);
        //发送验证码，返回ok（这里假装发一下验证码，实际上要不调用云平台的服务
        log.debug("短信验证码为：" + code);
        return Result.ok();
    }
```

```
 public Result login(LoginFormDTO loginForm, HttpSession session) {
        //校验手机号
        String phone = loginForm.getPhone();
        if (RegexUtils.isPhoneInvalid(phone)) {
            return Result.fail("手机号格式错误");
        }
        //从redis里获取并校验验证码

        String cashCode = stringRedisTemplate.opsForValue().get(LOGIN_CODE_KEY +
phone);//找到key对应的value
        String code = loginForm.getCode();
        if (cashCode == null || !cashCode.equals(code)) {
            return Result.fail("验证码错误");
        }
        User user = query().eq("phone", phone).one();//用mybatis-plus的框架查询
        if (user == null) {//用户不存在就创建出来
            user = createUserWithPhone(phone);
        }
        //保存在redis中
        //随机生成一个token作为登录的令牌
        String token= UUID.randomUUID().toString(true);
        //将User对象转为HashMap存储
        UserDTO userDTO =BeanUtil.copyProperties(user,UserDTO.class);//用户脱敏
        Map<String, Object> userMap = BeanUtil.beanToMap(userDTO, new HashMap<>(),
                CopyOptions.create()
                        .setIgnoreNullValue(true)
                        .setFieldValueEditor((fieldName, fieldValue) ->
fieldValue.toString()));
        String tokenKey=LOGIN_USER_KEY+token;
        //用哈希结构存储 多个字段有多个属性 可以存好几个键值对
        stringRedisTemplate.opsForHash().putAll(tokenKey,userMap);
        //设置token的有效期
        stringRedisTemplate.expire(tokenKey,LOGIN_USER_TTL,TimeUnit.MINUTES);//30分钟
        //返回token
        return Result.ok(token);
    }
```

```
public class RefreshTokenInterceptor implements HandlerInterceptor {

    private StringRedisTemplate stringRedisTemplate;
```

```
    public RefreshTokenInterceptor(StringRedisTemplate stringRedisTemplate) {
        this.stringRedisTemplate = stringRedisTemplate;
    }

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        // 1.获取请求头中的token
        String token = request.getHeader("authorization");
        if (StrUtil.isBlank(token)) {
            return true;
        }
        // 2.基于TOKEN获取redis中的用户
        String key = LOGIN_USER_KEY + token;
        Map<Object, Object> userMap = stringRedisTemplate.opsForHash().entries(key);
        // 3.判断用户是否存在
        if (userMap.isEmpty()) {
            return true;
        }
        // 5.将查询到的hash数据转为UserDTO
        UserDTO userDTO = BeanUtil.fillBeanWithMap(userMap, new UserDTO(), false);
        // 6.存在，保存用户信息到 ThreadLocal
        UserHolder.saveUser(userDTO);
        // 7.刷新token有效期
        stringRedisTemplate.expire(key, LOGIN_USER_TTL, TimeUnit.MINUTES);
        // 8.放行
        return true;
    }
```

7、讲一下 Redis 中的内存淘汰机制、有哪些内存淘汰策略?

官方解析

Redis 是一种基于内存的键值数据库，由于内存有限，当 Redis 占用的内存达到上限时，就需要进行**内存淘汰**，以腾出一些内存空间。

Redis 中的**内存淘汰机制**包括：

1. **定期删除**：Redis 可以设置一个定时器，定期扫描键空间中的键，并删除已经过期的键。
2. **惰性删除**：当一个键过期时，Redis 不会立即删除该键，而是等到该键被访问时再删除。
3. **内存淘汰策略**：当 Redis 内存占用达到上限时，会根据内存淘汰策略来选择一些键进行删除，以腾出更多的内存空间。

Redis 中的内存淘汰策略包括：

1. **noeviction**：禁止删除键，即不做任何操作。
2. **allkeys-lru**：从所有的键中选择最近最少使用的键进行删除。

3. **allkeys-random**：从所有的键中随机选择一些键进行删除。
4. **volatile-lru**：从已设置过期时间的键中选择最近最少使用的键进行删除。
5. **volatile-random**：从已设置过期时间的键中随机选择一些键进行删除。
6. **volatile-ttl**：从已设置过期时间的键中选择剩余时间最短的键进行删除。

其中，noeviction 策略是最简单的策略，但可能会导致 Redis 内存占满，并导致 Redis 无法正常工作。其他策略则会根据不同的算法进行键的选择和删除，以尽可能地保留重要的键。

总之，Redis 中的内存淘汰机制是保证 Redis 正常运行的重要机制之一，内存淘汰策略则根据不同的场景选择合适的策略来删除不必要的键，以腾出更多的内存空间。

鱼友的精彩回答

木木的回答

Redi 中的内存淘汰机制是指当Redis作为缓存使用时，他可以自动删除旧的数据以添加新的数据。这样可以避免内存溢出的情况。

Redis 有以下几种内存淘汰策略：

- noeviction：当内存打到限制时，不删除任何键，而是返回错误；
- allkeys-lru：当内存达到限制时，删除最近最少使用（LRU）的键；
- volatile-lru：当内存达到限制时，删除设置了过期时间并且最近最少使用的键；
- allkeys-random：当内存达到限制时，随机删除任意键；
- volatile-random：当内存达到限制时，随机删除设置了过期时间的键；
- volatile-ttl：当好内存达到限制时，删除设置了过期时间并且剩余生存时间最短的键。

如何设置 Redis 的内存淘汰策略

- Redis 的配置文件中，找到 maxmemory 和 maxmemory-policy 两个参数；
- 将 maxmemory 设置为你想要的内存限制，比如 100 Mb；
- 将 maxmemory-policy 设置为你想要的内存淘汰策略，例如 allkeys-lru
- 保存配置文件并重启 Redis 服务

也可以通过命令行动态修改这两个参数。

```
>redis-cli config set maxmemory 1--mb
>redis-cli config set maxmemory-policy allkeys-lru
```

8、Redis 6.0 之后为何引入了多线程？6.0 之前为什么不使用多线程?

官方解析

在 **Redis 6.0** 之前，Redis 是**单线程**的，这是因为 Redis 的主要瓶颈是**在 CPU 上**。但是随着硬件的发展，现代服务器的 CPU 核心数已经达到了几十个，这就导致 **Redis 单线程模型无法充分利用多核处理器的性能**。因此，Redis 6.0 引入了多线程，以提高 Redis 在多核处理器上的性能。

Redis 6.0 之前为什么不使用多线程，主要有以下几个原因：

1. Redis **单线程模型相对简单**，容易维护和调试，代码逻辑也比较清晰。
2. Redis 的**主要瓶颈在于 CPU，而不是 I/O**，因此**采用多线程模型并不能显著提高性能**。
3. Redis 是一个**内存型数据库**，它的性能主要受到 **CPU 和内存带宽**的限制。采用**多线程模型**会增加线程之间的竞争和锁等开销，反而可能降低 Redis 的性能。

但是随着硬件的发展，**多核处理器**已经成为了现代服务器的标配，因此 Redis 引入多线程的举措可以更好地发挥硬件的性能，提高 Redis 的吞吐量和响应速度。

鱼友的精彩回答

猿二哈的回答

redis 的多线程

1. redis 在 6.0 支持的多线程是指**网络 IO 的多线程**，而 **redis 的指令执行还是单线程的**
2. redis 的性能取决于**网络 IO，内存和 CPU**
3. 在 redis6.0 之前的**处理网络 IO 是单线程的**，服务端对客户端的请求，socket 的连接，指令数据的接收，解析都由**一个线程**来执行，效率较低
4. redis6.0 之后支持**网络 IO 的多线程**，通过**多线程并行**的执行来提高网络 IO 的处理效率
5. redis 的**服务端对用户传过来的指令的执行还是单线程的**
6. redis 的**网络 IO 多线程**默认是关闭的，需要在 **redis.conf** 中进行配置
7. redis 对指令的执行不改为多线程，是因为**指令的执行是在内存中，没有性能的瓶颈**，且 redis 对数据结构进行了很多的优化。若 redis 指令的执行支持多线程，则**需要对数据的操作加锁的同步**，这会导致性能的下降，**代价较大**。
8. redis6.0 之前不使用多线程的原因：
    - redis 的性能瓶颈为**内存和网络**，官方说明 redis 几乎不存在 CPU 的性能瓶颈问题
    - redis 使用**单线程**，其**可维护性提高**
    - redis 若**使用多线程**，会增加需要**对数据加锁**，这增加了**系统的复杂度**，且会**造成性能下降**

关键词：网络IO多线程，指令执行单线程，内存，网络，CPU，默认关闭，可维护性，加锁同步

Starry 的回答

Redis 6.0 之后，引入**多线程**主要是为了解决 Redis 在处理大规模数据时，**单线程的性能瓶颈**问题，以提高 Redis 的处理能力和效率。

在 **Redis 6.0** 之前，Redis 只采用**单线程**的方式进行处理，这是因为 Redis 的核心是**一个基于内存的键值对存储系统**，它的瓶颈**主要在于 CPU 的计算能力，而不是 I/O 操作的速度**。

在**单线程模式**下，Redis 可以使用简单的事件驱动模型，来实现高效的网络通信和事件处理，避免了线程切换和上下文切换带来的开销，同时也避免了**多线程**之间的锁竞争问题。因此，在 6.0 之前，Redis 一直采用**单线程模式**运行。

但是随着 Redis 的应用场景不断扩大和升级，Redis 也面临着越来越大规模、越来越高并发的挑战，**单线程模式**已经不能满足这些需求了。因此，在 Redis 6.0 中引入了**多线程**技术，以利用多核 CPU 的计算能力，提高 Redis 的性能和处理能力。

Redis 6.0 引入**多线程**后，采用了多种技术手段来实现**多线程**操作的安全和稳定性，如使用锁和原子操作来保证数据一致性和线程安全。

需要注意的是：

- Redis 6.0 中的**多线程**并不是完全替代了**单线程模型**，而是在其基础上引入了**多线程**支持，通过将一些负载耗时的操作（如I/O操作）交给后台线程处理，从而提高 Redis 的性能。同时，在**多线程**模式下，Redis 仍然保留了所有的**单线程模式**特性，如 **ACID 事务**等。
- Redis 6.0 中**多线程**的使用是可选的，并且可以通过配置文件进行启用或禁用，以便在不同的应用场景下选择最适合的运行模式。

航仔丶的回答

Redis 6. 0 引入多线程的原因有两个：

1. **充分利用 CPU 多核**，6.0 之前主线程只能利用一个核，多线程可以利用服务器 CPU 资源。
2. 多线程任务可以**分摊 Redis 同步 IO 读写负荷**。

Redis 6.0 默认情况下是关闭多线程的，需要**在 redis.conf 配置文件**中进行配置:

```
io-threads-do-reads yes
```

开启多线程后需要设置线程数，否则不会生效。**线程数应该小于机器核数**。官方建议 4 核的机器建议设置为 2 或 3 个线程，8 核的建议设置为 6 个线程 .

Redis 在**处理客户端的请求时是单线程的**，其中执行命令阶段，由于 Redis 是单线程来处理命令的，所有每一条到达服务端的命令不会立刻执行，所有的命令都会进入一个 Socket 队列中，当 socket 可读则交给单线程事件分发器逐个被执行. Redis6.0 引入多线程主要是**为了解决 Redis 在处理客户端请求时网络 I/O 消耗过大**的问题。**多线程主要用来处理网络数据的读写和协议解析，而执行命令仍然是单线程**.

Redis 的多线程网络模型并不是标准的 **Multi-Reactors/Master-Workers 模型**，I/O 线程任务仅仅是通过 socket 读取客户端请求命令并解析，但没有真正去执行命令. Redis 的多线程方案中，删除操作是可以被其他线程异步处理的，这些删除操作可以通过多线程的方式异步处理.

总的来说，Redis 引入多线程的目的是为了**提高 Redis 的性能，充分利用 CPU 多核，分摊 Redis 同步 IO 读写负荷，解决 Redis 在处理客户端请求时网络I/O消耗过大**的问题。

之前为什么不使用多线程的原因：

1. Redis 6.0 之前不使用多线程的原因是因为 Redis 通过 **AE 事件模型以及 IO 多路复用等技术**，处理性能非常高，因此没有必要使用多线程。单线程机制使得 Redis 内部实现的复杂度大大降低，Hash 的**惰性 Rehash**、**Lpush** 等等“线程不安全”的命令都可以无锁进行。
2. Redis 将所有数据放在内存中，内存的响应时长大约为 100 纳秒，对于小数据包，Redis 服务器可以处理 **80,000到 100,000 QPS**，这也是 Redis 处理的极限了，对于 80% 的公司来说，单线程的 Redis 已经足够使用了。因此，在处理性能方面，Redis 6.0 之前的单线程机制已经足够。
3. 但是随着互联网业务系统所要处理的线上流量越来越大，**Redis 的单线程模式会导致系统消耗很多 CPU 时间在网络 I/O 上从而降低吞吐量**。因此，**Redis 6.0 引入了多线程来提高性能**。
4. Redis 6.0 支持多线程有两个原因：**一是可以充分利用服务器 CPU 资源，目前主线程只能利用一个核；二是多线程任务可以分摊 Redis 同步 IO 读写负荷**。通过将主线程 IO 读写任务拆分出来给一组独立的线程处理，使得**多个 socket 读写可以并行化**，但是 **Redis 命令还是主线程串行执行**。
5. Redis 的多线程网络模型实际上并不是一个标准的 **Multi-Reactors/Master-Workers 模型**，**I/O 线程任务仅仅是通过 socket 读取客户端请求命令并解析**，却没有真正去执行命令。因此，Redis 6.0 的多线程模型相对于其他多线程模型更加轻量级。
6. **Redis 6.0 默认没有开启多线程，但可以在 conf 文件中进行配置**。关于线程数的设置，官方建议 4 核的机器设置为 2 或 3 个线程，8 核的机器建议设置为 6 个线程，线程数一定要小于机器核数。

9、Redis 有哪些数据类型？基础数据结构有几种？你还知道哪些 Redis 的高级数据结构?

官方解析

Redis 支持多种数据类型，不同的数据类型可以满足不同的需求。下面是 Redis 中常用的数据类型：

1. String（字符串）：Redis 中最基本的数据类型，可以存储任何形式的数据，例如整数、浮点数、二进制数据等。
2. Hash（哈希）：Redis 中的一种键值对类型，可以存储多个键值对，每个键值对又是一个键值对结构。
3. List（列表）：Redis 中的一个有序列表类型，可以存储多个元素，每个元素都有一个索引，支持多种列表操作，例如插入、删除、查找等。
4. Set（集合）：Redis 中的一种无序集合类型，可以存储多个元素，每个元素都是唯一的，支持多种集合操作，例如交集、并集、差集等。
5. Sorted Set（有序集合）：Redis 中的一种有序集合类型，可以存储多个元素，每个元素都有一个分值，支持根据分值进行排序和查询等操作。
   Redis 的基础数据结构有三种：字符串、列表和哈希，其他的数据类型都是基于这三种数据结构进行扩展和衍生的。例如，Redis 的 Set 数据类型就是基于字符串实现的。

除了基础数据结构之外，Redis 还提供了多种高级数据结构，例如：

1. HyperLogLog：一种基数估计算法，用于估计一个数据集合的基数。
2. GeoHash：一种地理位置编码算法，可以对地理位置信息进行编码和查询。

3. Pub/Sub：一种消息队列机制，可以实现消息的订阅和发布。
4. Bitmaps：一种位图数据结构，可以进行高效的位运算，用于统计用户在线时长、网站访问量等。
5. Lua 脚本：Redis 中可以使用 Lua 脚本进行扩展和定制，可以实现一些复杂的业务逻辑和算法。

鱼友的精彩回答

爱吃鱼蛋的回答

数据类型/结构	简介	是否基础类型	应用场景
String(字符串)	**最基本的数据类型**，用于存储字符串、整数或浮点数。	是	**缓存数据**，如页面缓存、对象缓存等；计数器，如网站访问量、用户在线时长等；分布式锁，如限流、秒杀等。
List(列表)	一组有序的元素，可以在头部或尾部添加元素，适用于存储有序的元素集合。	是	**消息队列**，如异步任务、日志收集等；**队列、栈**等数据结构的实现，如任务队列、聊天记录等。
Set(集合)	一组无序的元素，可以进行交集、并集、差集等操作。	是	**对象存储**，如用户信息、商品信息等；**缓存数据**，如页面缓存、对象缓存等。
Hash(哈希表)	一种键值对的集合，适用于存储对象。	是	去重、标签、好友列表等。
Sorted Set(有序集合)	一组有序的元素，每个元素都有一个分值，可以按照分值排序和范围查询。	是	**排行榜**，如热门商品、网站排名等；**分数排序**，如商品价格、评分等。
Bitmaps(位图)	一种特殊的字符串，可以进行位运算，用于存储二进制数据。	否	存储用户**签到**信息、在线状态等。
HyperLogLog(基数统计)	一种基于概率算法的数据结构，用于统计元素的数量。	否	统计网站的 UV、PV 等指标。
Geo(地理位置)	一种存储地理位置信息的数据结构，可以进行地理位置相关的操作。	否	存储商家、用户的**地理位置信息**，**实现附近商家推荐、地理位置搜索**等。
Streams(流)	一种类似于消息队列的数据结构，可以进行发布和订阅操作。	否	实现**消息队列、事件流**等。
Lua scripting(Lua 脚本)	可以通过 Lua 脚本扩展 Redis 的功能。	否	**分布式锁，如限流、秒杀、限流、自动补全**等。

10、Redis 为什么快?

官方解析

Redis 之所以快，主要有以下几个方面的原因：

1. **内存存储**：Redis 的数据都是存储在内存中的，相比于硬盘存储的数据库，内存存储速度更快。
2. **单线程模型**：Redis 使用单线程模型处理所有的请求，避免了多线程之间的线程切换和竞争等开销，提高了处理请求的效率。
3. **非阻塞 I/O**：Redis 使用非阻塞 I/O 处理网络通信，当一个客户端请求到来时，Redis 不会一直等待客户端的响应，而是会先处理其它的请求，这样就避免了 I/O 阻塞带来的性能问题。
4. **精简高效的数据结构**：Redis 内置了多种高效的数据结构，如哈希表、跳表等，这些数据结构的实现非常精简高效，减少了 Redis 对内存和 CPU 的占用，从而提高了 Redis 的性能。
5. **持久化策略**：Redis 支持多种持久化策略，如 RDB（快照）和 AOF（追加式文件）等，这些策略可以将内存中的数据保存到硬盘中，以保证数据的持久性和安全性。同时，Redis 可以将数据以压缩的方式存储在硬盘中，减少了硬盘的占用，提高了数据的读写速度。

综上所述，Redis 之所以快，主要得益于其内存存储、单线程模型、非阻塞 I/O、高效的数据结构和灵活的持久化策略等方面的设计和实现。

鱼友的精彩回答

Starry 的回答

1、**纯内存操作**

Redis 是基于内存的数据存储系统，绝大部分请求是纯粹的内存操作。

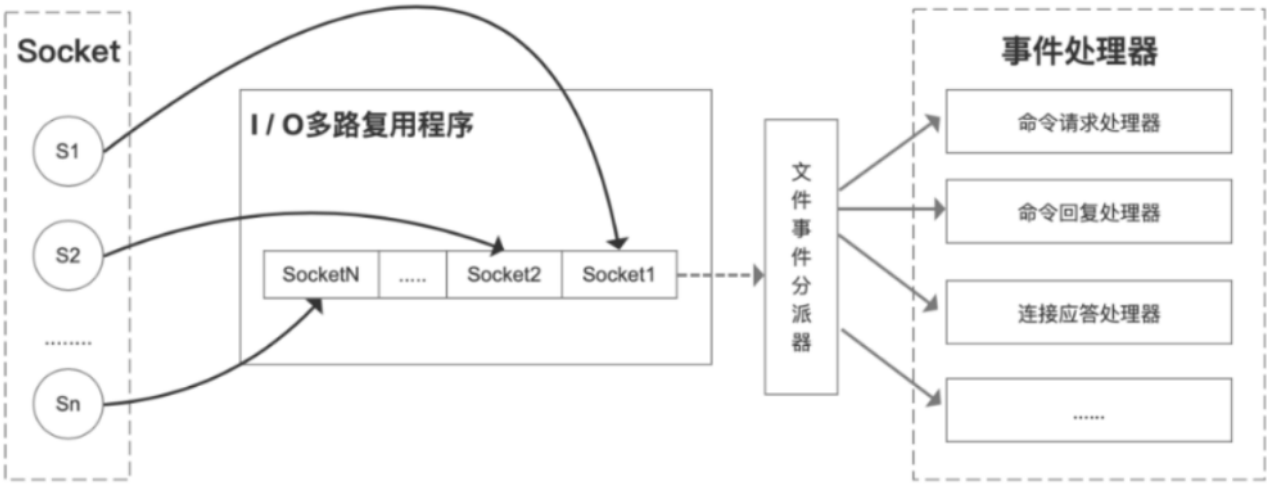
2、**单线程操作，避免了频繁的上下文切换**

Redis 的单线程操作是指，Redis 使用一个主线程来处理所有的客户端请求和数据操作，不会创建新的线程来处理请求。这种**单线程模型的优点是可以避免多线程并发访问共享数据时的竞争和死锁问题**，从而提高了 Redis 的性能和稳定性。此外，由于 Redis 的内存访问速度非常快，因此单线程处理请求也能够保证足够的性能。

3、**采用了非阻塞 I/O 多路复用机制**

为了实现单线程模型，Redis 使用了 IO 多路复用技术。IO 多路复用是指操作系统提供的一种 IO 模型，可以让一个进程同时监听多个 IO 事件（如读写事件），并在有事件发生时通知进程，从而实现并发处理 IO 事件。

具体来说，在 Redis 中，客户端的请求是由一个单线程来处理的，而 IO 操作却是通过 epoll 多路复用技术实现的。



高性能 IO 多路复用

Redis 单线程情况下，内核会一直监听 socket 上的连接请求或者数据请求，一旦有请求到达就交给 Redis 线程处理，这就实现了一个 Redis 线程处理多个 IO 流的效果。

select/epoll 提供了基于事件的回调机制，即针对不同事件的发生，调用相应的事件处理器。所以 Redis 一直在处理事件，提升 Redis 的响应性能。

HeiHei 的回答

Redis

- 核心业务部分（处理命令）使用的是单线程
- 抛开持久化不谈，Redis 是纯内存操作，执行速度非常快，数据存在内存中，绝大部分请求是纯粹的内存操作，非常快速，跟传统的磁盘文件数据存储相比，避免了通过磁盘 IO 读取到内存这部分的开销。
- 数据结构简单，对数据操作也简单。Redis 中的数据结构是专门进行设计的，每种数据结构都有一种或多种数据结构来支持。Redis 正是依赖这些灵活的数据结构，来提升读取和写入的性能。
- 使用基于 IO 多路复用机制的线程模型，可以处理并发的链接。
- redis6.0 后引入多线程，因为 Redis 的瓶颈不在内存，而是在网络 I/O 模块带来 CPU 的耗时，所以 Redis6.0 的多线程是用来处理网络 I/O 这部分，充分利用 CPU 资源，减少网络 I/O 阻塞带来的性能损耗。

11、如何使用 Redis 实现分布式锁?

官方解析

使用 Redis 实现分布式锁的基本思路是使用 Redis 的原子性操作来确保在多个客户端之间只有一个客户端可以成功获取锁。以下是使用 Redis 实现分布式锁的步骤：

1. 获取锁：客户端尝试获取锁，可以使用 Redis 的 SET 命令结合NX和EX选项来实现。SET 命令用于设置一个键值对，NX 选项表示只有在键不存在时才设置，EX 选项表示设置一个过期时间（单位为秒）。 示例：SET lock_key unique_value NX EX 30。这个命令表示如果 lock_key 不存在，就设置它的值为 unique_value，并设置 30 秒的过期时间。如果设置成功，表示获取锁成功；如果设置失败，表示锁已被其他客户端持有，需要等待或重试。
2. 持有锁：在持有锁的客户端执行临界区代码。在此期间，其他客户端无法获取锁。
3. 释放锁：在临界区代码执行完毕后，需要释放锁以供其他客户端使用。为了避免误解锁的情况（例如，由于执行时间过长导致锁过期，然后被其他客户端获取），在释放锁时需要检查锁的值是否与获取锁时设置的值相同。这可以通过 Redis 的 EVAL 命令来实现，使用 Lua 脚本进行原子性操作。 示例：EVAL "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end" 1 lock_key unique_value。这个命令表示检查lock_key的值是否为unique_value，如果是，则删除 lock_key 以释放锁；否则不执行任何操作。
   通过以上步骤，可以实现一个基本的 Redis 分布式锁。在实际应用中，还需要考虑一些其他因素，例如获取锁的重试策略、锁的续期等。

鱼皮的评论

题解更倾向于以原生命令的方式去实现，其实还有很多现成的第三方库，比如 Redisson。这里要关注分布式锁可能出现的各种异常情况。

鱼友的精彩回答

猿二哈的回答

redis 实现分布式锁

1. redis 实现分布式锁的方式有两种：
2. 通过 redis 提供的 setnx 进行实现，往 redis 中使用 setnx 插入 key 时，如果 key 存在，则返回 0，可以通过插入 key 的返回值进行判断来实现分布式锁
3. 通过使用 Redission（客户端）来实现分布式锁。可以调用 Redission 提供的 api，即 lock()，unlock()方法进行加锁和锁的释放。此外， Redission 还提供了 watchdog，即看门狗，来对加锁的 key 每隔 10 s对该 key 进行续时，（从而避免锁的过期）
4. Redission 的所有指令是通过 lua 脚本进行实现的，lua 脚本可以保证所有指令执行的原子性

关键词：setnx，Redission，lock，unlock，watchdog，lua，原子性

回家养猪的回答

基于 Redis 的分布式锁实现思路：

- 利用 setnx 获取锁，并设置过期时间，保存线程标识
  - uuid+ 线程 id 因为不同服务器线程 id 有可能一样, 所以拼接 uuid
- 释放锁时先判断线程标识是否与自己一致，一致则删除锁
  - 判断锁是否为自己的然后再释放, 这个过程必须是原子性的否则有可能释放别人的锁, 这就需要 lua 脚本.

特性：

- 利用 set nx 满足互斥性
- 利用 set ex 保证故障时锁依然能释放，避免死锁，提高安全性
- 利用 Redi s集群保证高可用和高并发特性

可重入

- 和 jdk 的可重入锁的原理是一致的
- 使用 has h类型, key 为 userId 或者商品 id , field 为线程 id, value 为数字. 加锁解锁都需要使用 lua 脚本
    - 加锁
        - 判断 key 是否存在, 不存在则直接加锁, 加过期时间 (key 为 userId 或者商品 id , field 为线程 id, value 为 1)
        - 存在则说明有方法已经加了锁, 此时判断 field 的线程 id 是否则自己一致(是否为同一线程), 不同则返回, 相同则 value+1, 然后设置过期时间
    - 加锁后执行流程
        - 方法 1 加锁后, value=1, 方法1调用方法2
        - 方法 2 加锁, redisson 判断锁标识是否是自己(field 字段是否为同一线程), 若是则 value++, value 为 2
        - 同理, 方法 3 加锁, value=3
- 释放锁
    - 判断锁是否是自己线程的. 不是则退出. 是则 value-1
    - 判断重复次数 value 是否为 0, 如果为 0 则释放锁. 不为 0 说明锁其他人还在用, 则重置有效期.
- 释放锁执行流程
    - 方法 3 执行完毕, 释放锁, value--, value=2
    - 方法 2 释放锁, value=1
    - 方法 1 释放锁, value=0, 此时删除 redis 的该数据, 锁完全释放.
    - 可重试：利用信号量和 redis 的 pubsub 发布订阅机制实现等待、唤醒，获取锁失败的重试机制
- 先直接获取锁, 如果获取失败, 并不是直接重试, 因为现在立即重试大概率其他线程正在执行业务. 获取锁失败会先暂时等待. (CPU 占用率不会很高, 性能不错)
- 获取锁成功的线程在释放锁时会发布一条消息.
- 当其他线程得到该消息时, 就会重新获取锁, 如果再次获取锁失败, 就会再次等待.
- 但是不是无限制的等待, 因为他会有一个等待时间, 超过该时间则不重试直接返回 false

超时续约：利用 watchDog,每隔一段时间(releaseTime/3), 重置超时时间

- 看门狗机制会创建一个守护线程, 当锁快到期但是业务线程没执行完时为锁增加时间 (续命).
- 当然看门狗也不会无限地增加超时时间, redisson 一个参数用来设置加锁的时间, 超过这个时间后锁便自动解开了，不会延长锁的有效期。

主从一致性问题: 使用多个独立的 Redis 节点,

- 获取锁时, 往每一个 redis 节点都写入 key. 即便其中一台 redis 宕机, 其他 redis 依旧有锁信息.

- 并且必须在所有节点都获取到锁, 才算获取锁成功.
- Redisson 分布式联锁 RedissonMultiLock 对象可以将多个 RLock 锁对象关联为一个联锁, 可以把一组锁当作一个锁来加锁和释放。

爱吃鱼蛋的回答

为什么

对于分布式系统来说不能够使用 Java 自带的 synchronized 或 Lock 等实现上锁，因为分布式情况下不同节点的 JVM 不同，导致锁对象不同，会造成**锁失效**问题。 而使用 Redis 实现分布式锁可以很好的解决分布式所带来的的问题，并且因为 Redis 的特性，**锁的性能并不低**。 同时由于加锁是使用到了 Redis 中的原子命令 SET key value [EX seconds] [NX]，因此在实现过程中并不需要自行编写 Lua 脚本来保证**锁的原子性**，简化了开发难度。

解释一下 SET key value [EX seconds] [NX]命令：

- EX 表示该命令会设置一个过期时间，单位为秒，后面的 seconds 便是具体的数值；
- NX 表示只有 key 不存在的时候才会设置成功，使用 Redis 实现分布式锁的核心便是这个特性。

实现步骤

下面是使用 Redis 实现分布式锁的步骤：

1. 生成分布式锁的 key，一般为业务相关的业务 key；
2. 生成分布式锁的 value，一般为1或者线程 id；
3. 生成分布式锁的过期时间，避免锁过程中宕机锁无法解开；
4. 通过不同语言的 API 实现 SET key value [EX seconds] [NX]命令；
5. 通过命令执行结果返回的标识判断是否加锁成功，一般 true 为上锁成功；
6. 释放锁时，通过删除分布式锁的 key 实现。

Java 实现

这里提供不可重入的分布式锁的实现方式：

```
@Component
public class RedisLock {

    @Resource
    private StringRedisTemplate redisTemplate;

    /**
     * 尝试获取分布式锁
     * @param key 锁的键值
     * @param value 锁的值
     * @param expireTime 锁的过期时间
     * @param timeUnit 锁的过期时间单位
     * @return 是否获取到锁
     */
```

```
    public boolean tryLock(String key, String value, long expireTime, TimeUnit
timeUnit) {
        // 利用 setIfAbsent 方法实现分布式锁
        Boolean success = redisTemplate.opsForValue().setIfAbsent(key, value,
expireTime, timeUnit);
        return Boolean.isTrue(success);
    }

    /**
     * 释放分布式锁
     * @param key 锁的键值
     */
    public void unlock(String key) {
        redisTemplate.delete(key);
    }
}
```

好处

使用 Redis 实现分布式锁主要有以下好处：

- **高性能**：Redis 是一个高性能的内存数据库，可以快速地完成获取和释放锁的操作；
- **可靠性**：Redis 的持久化机制和主从复制机制可以保证锁的可靠性，即使 Redis 节点出现故障，也可以保证锁的可用性；
- **可重入性**：可以支持可重入锁，即同一个线程可以重复获取同一个锁，而不会被其他线程所影响；
- **灵活性**：可以支持多种锁的实现方式，如阻塞式、非阻塞式、公平锁、非公平锁等，可以根据业务需求选择最合适的锁；
- **分布式环境下的协作**：可以支持分布式环境下的多个进程或者节点之间的协作，实现分布式系统的同步和协作。

补充：这种方式可能会出现业务还未完成但锁已经释放的问题(如果业务执行时间超过了锁的过期时间)，这个问题可以使用 Redssion 解决(看门狗机制)

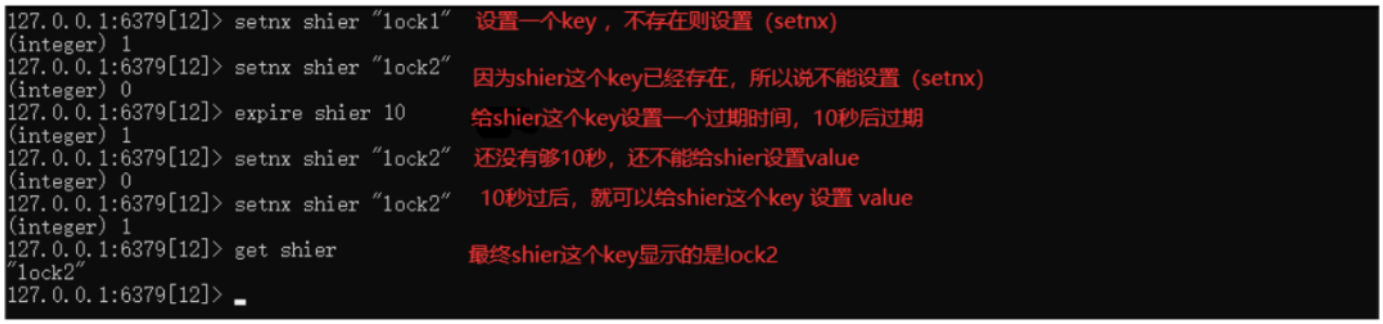
猫十二懿的回答

使用 Redis 实现分布式锁的核心思想是利用 Redis 的原子性操作，对 Redis 中特定的 key 进行 setnx（set if not exists）操作，如果成功返回true 表示获取到了锁，否则返回 false 表示没有获取到锁。在获取到锁的情况下，需要设置过期时间和在释放锁之前判断当前锁是否是自己持有的，以防出现死锁等问题。

具体实现步骤如下：

1. 生成一个唯一的锁标识 key，并设置锁的过期时间。
2. 使用 setnx 命令尝试获取锁，如果返回结果为 1，则表示获取锁成功，进入下一步；否则等待一段时间重试，直到获取到锁为止。
3. 在持有锁的时间内，执行相关业务逻辑操作，并定时更新锁的过期时间，防止锁时间过长而导致锁自动失效。
4. 释放锁，通过比较锁的持有者是否是自己来判断是否能够释放锁。

通过 redis 命令实现分布式锁：



Java 实现一个分布式锁：

```
public class RedisDistributedLock {

    private static final String LOCK_PREFIX = "redis_lock_";
    private static final int EXPIRE_TIME = 5; // 锁过期时间5秒

    private RedisTemplate<String, Object> redisTemplate;
    private String lockKey;
    private String lockValue;

  // 构造函数
    public RedisDistributedLock(RedisTemplate<String, Object> redisTemplate, String
lockKey) {
        this.redisTemplate = redisTemplate;
        this.lockKey = LOCK_PREFIX + lockKey;
        this.lockValue = UUID.randomUUID().toString();
    }

    // 获取锁
    public boolean lock() {
        try {
            String result = redisTemplate.execute((RedisCallback<String>) connection ->
{
                JedisCommands commands = (JedisCommands)
connection.getNativeConnection();
                return commands.set(lockKey, lockValue, "NX", "EX", EXPIRE_TIME);
            });
            return StringUtils.isNotBlank(result);
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    // 释放锁
    public boolean unlock() {
        try {
            String script = "if redis.call('get',KEYS[1]) == ARGV[1] then return
redis.call('del',KEYS[1]) else return 0 end";
```

```
            Long result = redisTemplate.execute((RedisCallback<Long>) connection -> {
                JedisCommands commands = (JedisCommands)
connection.getNativeConnection();
                return (Long) commands.eval(script, Collections.singletonList(lockKey),
Collections.singletonList(lockValue));
            });
            return result != null && result > 0;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }
}
```

12、如何用 Redis 中的 HyperLogLog 统计页面 UV?

官方解析

在 Redis 中，HyperLogLog 是一种**基数估算算法**，可以用于快速计算一个集合中的不同元素数量的近似值。

在使用 HyperLogLog **统计页面** UV 时，我们可以将每个访问者的 IP 地址作为一个元素加入到 HyperLogLog 中，然后通过计算 HyperLogLog 中元素数量的近似值来估计页面的唯一访问者数量。

以下是一个示例代码，展示如何使用 Redis 的 HyperLogLog 实现页面 UV 统计：

```
import redis

# 连接 Redis 服务器
r = redis.Redis(host='localhost', port=6379)

# 记录访问者 IP 地址
ip_address = '192.168.0.1'

# 将 IP 地址加入到 HyperLogLog 中
r.pfadd('page_views', ip_address)

# 获取 HyperLogLog 中元素数量的近似值
uv = r.pfcount('page_views')

# 输出页面 UV
print('页面 UV：', uv)
```

在上面的示例代码中，我们使用 Redis 的 pfadd() 方法将每个访问者的 IP 地址加入到名为 page_views 的 HyperLogLog 中。然后，我们使用 pfcount() 方法获取 page_views 中元素数量的近似值，并将其作为页面 UV 的估计值输出到控制台。

需要注意的是，由于 HyperLogLog 是一种**基数估算算法**，其结果是一个**近似值**，并不是精确值。因此，在实际使用中，需要根据具体情况调整参数和精度等级，以获得更准确的结果。

鱼友的精彩回答

维萨斯的回答

HyperLogLog 是用于做**基数统计**的结构,在输入量非常大的时候,非常好用

- 12k的大小可以记录 2^64 次方个不同元素
- 误差极小 0.81%

我用的可是上流的 SpringBoot,为什么要用 Jedis?

使用 HyperLogLo g统计 UV

Application.yaml

```
server:
  port: 8080
spring:
  redis:
    host: 111.111.111.111
    port: 6379
    database: 0
    password: password
```

IpCountService

```
public void IPCount() {
    //... 得到访问的 IP 地址
    String ip = 得到 ip;
    Long hll = redisTemplate.opsForHyperLogLog().add("hll", ip);
    //... 打日志等后续操作
}
public long uv() {
        // PFCOUNT
        return redisTemplate.opsForHyperLogLog().size("hll");
}
```

CodeJuzi 的回答

使用 Redis 中的 HyperLogLog 统计页面 UV?

```
import redis.clients.jedis.Jedis;

public class PageUVCounter {

    private Jedis jedis;

    public PageUVCounter(Jedis jedis) {
```

```
        this.jedis = jedis;
    }

    public void countUV(String pageName, String userId) {
        String hllKey = "uv:" + pageName;
        jedis.pfadd(hllKey, userId);
    }

    public long getUV(String pageName) {
        String hllKey = "uv:" + pageName;
        return jedis.pfcount(hllKey);
    }

}
```

使用 Jedis 客户端实现了 **PageUVCounter** 类。**countUV** 方法用于统计页面UV，它使用Redis的 **pfadd** 命令将用户 ID 添加到HyperLogLog 中。**getUV** 方法用于获取页面UV，它使用 Redis 的 **pfcount** 命令获取 HyperLogLog 中的基数计数值，即页面的 UV 数量。

注意的点：

- HyperLogLog 是有误差的，误差范围在 0.81% 以内，因此在实际使用中需要根据误差范围进行适当的调整。
- HyperLogLog 中存储的是用户 ID 的哈希值，因此需要确保用户 ID 的哈希分布均匀，以避免误差过大。

heart000_1 的回答

使用 Redis 中的 HyperLogLog 可以很方便地统计页面的 UV（Unique Visitors），其主要步骤如下：

1. 创建一个 HyperLogLog 数据结构：使用 Redis 的 PFADD 命令创建一个 HyperLogLog 数据结构，用于记录所有访问过该页面的用户的信息。例如，使用以下命令创建一个名为 "page_uv" 的 HyperLogLog：

```
PFADD page_uv user1 user2 user3
```

上述命令将 user1、user2 和 user3 这三个用户信息添加到名为 "page_uv" 的 HyperLogLog 中。

2. 统计 HyperLogLog 的基数：使用 Redis 的 PFCOUNT 命令可以统计 HyperLogLog 的基数，即访问过该页面的唯一用户数。例如，使用以下命令可以统计名为 "page_uv" 的 HyperLogLog 的基数：

```
PFCOUNT page_uv
```

上述命令将返回访问过该页面的唯一用户数。

3. 定期合并 HyperLogLog：为了保证统计结果的准确性，需要定期合并多个 HyperLogLog。使用 Redis 的 PFMERGE 命令可以将多个 HyperLogLog 合并为一个 HyperLogLog。例如，使用以下命令可以将名为 "page_uv1" 和 "page_uv2" 的 HyperLogLog 合并为一个名为 "page_uv" 的 HyperLogLog：

```
PFMERGE page_uv page_uv1 page_uv2
```

上述命令将名为 "page_uv1" 和 "page_uv2" 的 HyperLogLog 合并为一个名为 "page_uv" 的 HyperLogLog，用于统计访问该页面的唯一用户数。

需要注意的是，使用 HyperLogLog 统计 UV 可能会出现误差，误差的大小取决于 HyperLogLog 的精度和数据规模。为了减小误差，可以增加 HyperLogLog 的精度（通过增加 HyperLogLog 的位数），或者将多个 HyperLogLog 定期合并。

> 作者：编程导航知识星球 + 星球鱼友们

计算机网络

1、简述计算机网络七层模型和各自的作用?

官方解析

计算机网络七层模型是一个把网络通信协议分为七个层次的标准模型，其目的是为了让计算机网络的设计和管理更加灵活和模块化。这个模型被称为OSI模型（Open System Interconnection Model），它由国际标准化组织（ISO）于1984年发布，是一个开放的标准模型。

每个层次都有自己的独立功能和责任，这种分层的方式使得每个层次都可以独立工作，同时还能够很好地协调上下层之间的数据传输，而不需要依赖于其他层次的实现细节。以下是每个层次的具体功能和责任：

- 物理层：主要负责通过物理媒介传输比特流，如电缆、光纤、无线电波等。物理层规定了物理连接的规范，包括电缆的类型、接口的规范等。
- 数据链路层：主要负责把数据分成数据帧进行传输，并对错误进行检测和纠正。数据链路层还负责物理地址的分配、数据流量控制、错误校验等。
- 网络层：主要负责数据在网络中的传输，包括路由选择、分组转发、数据报文的封装等。网络层还处理数据包的寻址和控制流量等。
- 传输层：主要负责数据传输的可靠性和流量控制等，同时还包括分段、组装、连接建立和断开等功能。传输层的最重要的两个协议是 TCP 和 UDP。
- 会话层：主要负责建立、管理和终止会话，提供会话控制和同步等服务。会话层还负责处理多个应用程序之间的数据交换。
- 表示层：主要负责数据格式转换、加密解密、压缩解压等服务。表示层使得应用程序可以使用不同的数据格式和编码，同时还提供了数据的安全性和完整性保护等服务。
- 应用层：主要提供各种服务和应用程序，如电子邮件、文件传输、远程登录、Web 浏览等。应用层服务可以使用不同的协议实现，如 HTTP、SMTP、FTP、TELNET 等。
  OSI模型是一种理论模型，在实际应用中较少使用。现在比较常用的是TCP/IP模型，它只有四层：应用层、传输层、网络层和数据链路层。

鱼友的精彩回答

一切总会归于平淡的回答

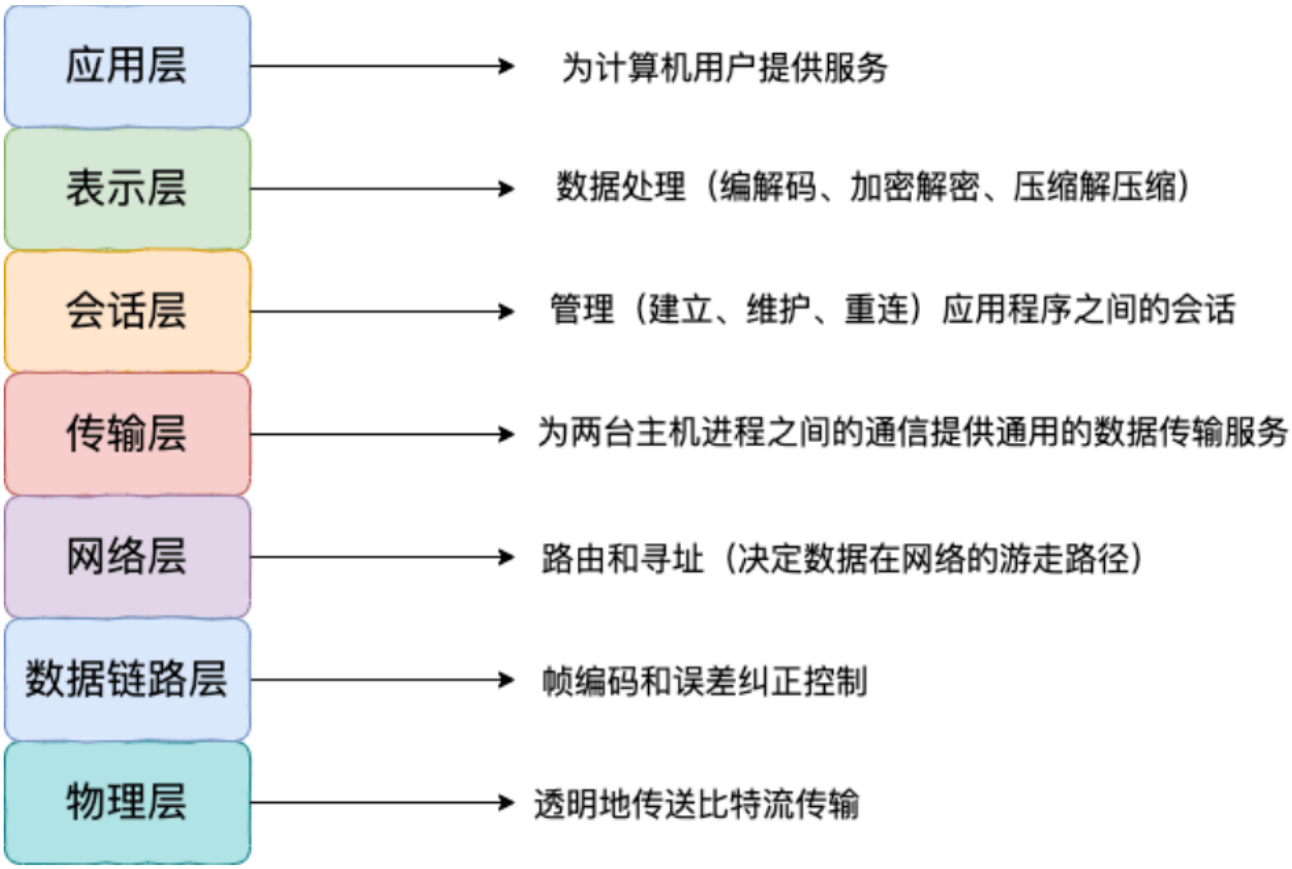
当我们在浏览器中输入网址并访问一个网站时，首先经过物理层将电信号转换为比特流，再通过数据链路层将比特流转换为数据帧，通过 MAC 地址寻找到下一跳设备进行传输。在网络层中，通过 IP 地址寻找到目标主机，路由选择最优路径进行数据传输。在传输层中，通过 TCP 协议保证数据传输的可靠性，同时控制数据流量。在应用层中，通过 HTTP 协议实现浏览器和服务器之间的通信，完成网页的展示。

当面试官问到计算机网络七层模型及其各自的作用时，他很可能想要了解你对计算机网络的基础知识是否掌握，并且是否能够清晰地解释每个层次的功能和作用。

此外，他也可能希望了解你是否熟悉网络通信中的一些概念和术语，以及你是否能够在实际应用中使用这些知识来解决网络通信问题。

小熊佩萁的补充

OSI 七层模型 是国际标准化组织提出一个网络分层模型，其大体结构以及每一层提供的功能如下图所示：



每一层都专注做一件事情，并且每一层都需要使用下一层提供的功能比如传输层需要使用网络层提供的路由和寻址功能，这样传输层才知道把数据传输到哪里去

行云的补充

举例说明：用 QQ 聊天为例子

- 用户在QQ界面输入文字，这就是应用层。
- QQ 为了安全，要给文字加密，这就是表示层做的。
- 会话层是找到对方的实体，也就是对方的 QQ 进程。
- 传输层要找到对方的端口，就是 QQ 传输信息用的是哪个端口。
- 网络层负责通过路由器要找到对方的网络地址。
- 数据链路层要通过物理地址找到对方主机。
- 物理层负责二进制比特流的传输。

2、HTTP 是哪一层的协议？简述它的作用?

官方解析

HTTP 是应用层协议，主要用于在 Web 浏览器和 Web 服务器之间传递数据。它是一种无状态的协议，即服务器不会保存关于客户端的任何信息，每次客户端发送请求，服务器都会返回响应。HTTP 协议通常基于 TCP 协议，使用 TCP 的 80 端口作为默认的传输端口。HTTP 协议主要作用包括：

1. 建立连接：客户端与服务器建立 TCP 连接，然后发送 HTTP 请求，服务器接收请求并处理。
2. 发送请求：客户端发送HTTP请求到服务器，包括请求方法（GET、POST、PUT等）、请求头（如User-Agent、Accept等）和请求正文（可选）等信息。
3. 处理请求：服务器接收并解析 HTTP 请求，执行请求操作（如查询数据库等），并将处理结果返回给客户端。
4. 返回响应：服务器返回HTTP响应，包括响应状态码（如200 OK、404 Not Found等）、响应头（如Content-Type、Cache-Control等）和响应正文（可选）等信息。
5. 关闭连接：客户端接收到响应后，关闭TCP连接。

HTTP 的请求报文主要包括以下几个部分：

1. 请求行：包括请求方法、URL 和 HTTP 协议版本。
2. 请求头：包括一些请求头部信息，例如 User-Agent、Host 等。
3. 空行：表示请求头结束。
4. 请求体：实际请求的数据，例如表单提交的数据等。

HTTP 的响应报文主要包括以下几个部分：

1. 状态行：包括 HTTP 协议版本、状态码和状态消息。
2. 响应头：包括一些响应头部信息，例如 Server、Content-Type、Content-Length 等。
3. 空行：表示响应头结束。
4. 响应体：实际响应的数据，例如网页的 HTML 代码、图片、音频等。

HTTP 的状态码指示了服务器对请求的处理结果。常见的状态码包括 200 OK（请求成功）、301 Moved Permanently（永久重定向）、404 Not Found（未找到资源）和 500 Internal Server Error（服务器内部错误）等。

总之，HTTP 协议的作用是规定了 Web 应用程序中客户端和服务器之间的通信方式和数据传输格式，是支持 Web 应用程序开发的基础协议。

鱼友的精彩回答

维萨斯的回答

Http（超文本传输协议）是应用层的协议

从概念

- 超文本：支持超越文本的内容，如图片、视频、音频等文件
- 传输： A <---> B 两点之间传输数据
- 协议：规范

作用：在计算机世界进行[两点传输][图片、视频、音频]等文件的协议

从使用 （ 随便回忆自己一个 项目实现的Web功能 + 计网拓展 ）

- 建立连接：通常基于 TCP 协议，通过 TCP 建立好两点之间的连接之后，进行传输
- 发送请求：发送请求（Get、Post、Put），发送请求头、请求体（全部都在 Http 请求里）
- 处理请求：收到Http请求，解析后服务端处理，然后给客户端
- 响应请求：发送响应状态码、响应头、响应体
- 关闭连接：Http响应完后，TCP也就关闭了（ 针对Http/1.0 ）

%的回答

HTTP（Hypertext Transfer Protocol）是应用层协议，是互联网上使用最广泛的协议之一。它基于客户端-服务器模型，通过在客户端和服务器之间传输文本数据来进行通信。HTTP通常使用TCP作为传输层协议，也可以使用TLS/SSL进行加密。

HTTP的作用是定义了客户端和服务器之间的通信方式，使得客户端可以向服务器请求资源，并且服务器可以向客户端发送响应结果。HTTP使用URL（Uniform Resource Locator）来定位资源，通过请求方法（如GET、POST、PUT、DELETE等）来描述对资源的操作，通过请求头和响应头来传递附加信息，如编码格式、内容类型、Cookie等。

HTTP 的主要特点包括以下几个方面：

1. 简单易用：HTTP 协议采用文本格式传输数据，易于人类阅读和编写，使用简单。
2. 无状态：HTTP 是一种无状态协议，每次请求和响应之间相互独立，服务器不会保存任何客户端信息，客户端需要自行维护会话状态。
3. 可扩展性：HTTP 允许通过扩展头部信息和请求方法等方式进行扩展，支持自定义数据传输格式和协议。
4. 非连接型：HTTP 是一种非连接型协议，每个请求和响应之间相互独立，不存在长期的连接状态。

HTTP 协议是应用层协议中非常重要的一种，它的作用是为 Web 应用程序提供了标准的通信方式，使得客户端和服务器之间的交互变得更加简单、高效和灵活。

3、HTTP 有哪些常见的状态码?

官方解析

HTTP（超文本传输协议）常见的状态码有以下几种：

1xx（信息类状态码）：指示已经接收到请求，正在继续处理。

2xx（成功状态码）：指示请求已经被接收、理解和接受。

- 200 OK：请求已成功处理。
- 201 Created：请求已经被实现，而且有一个新的资源已经依据请求的需要而建立。
- 204 No Content：服务器已经成功处理了请求，但是没有返回任何实体内容。

3xx（重定向状态码）：需要进行附加操作以完成请求。

- 301 Moved Permanently：请求的网页已永久移动到新位置。
- 302 Found：请求的网页已经临时移动到新位置。
- 304 Not Modified：客户端发送了一个带条件的请求，服务器端允许请求访问资源，但是请求未满足条件。

4xx（客户端错误状态码）：请求包含错误语法或不能被执行。

- 400 Bad Request：请求报文存在语法错误。
- 401 Unauthorized：表示发送的请求需要有通过 HTTP 认证的认证信息。
- 403 Forbidden：表示对请求资源的访问被服务器拒绝。
- 404 Not Found：请求的资源不存在。

5xx（服务器错误状态码）：服务器在处理请求的过程中发生了错误。

- 500 Internal Server Error：服务器遇到了一个未曾预料的状况，导致无法完成对请求的处理。
- 502 Bad Gateway：充当网关或代理的服务器，从远端服务器接收到了一个无效的请求。
- 503 Service Unavailable：服务器暂时处于超负载或正在停机维护，无法处理请求。

状态码是服务器对客户端请求结果的反馈，根据状态码可以快速定位问题所在，进行相应的处理。

> 鱼皮补充：这题建议大家分类记忆，比如 4xx 一般是客户端的问题，5xx 一般是服务端的问题；还可以再幽默一点，比如我们学校的抢课系统经常 503~

鱼友的精彩回答

维萨斯的回答

啥波一口诀 (也不用完全记忆)

一信 二成 三重 四五败

1xx 一信

提示信息,是协议处理的中间状态,还需要后续操作 (感觉一般也不会遇到)

2xx 二成

- 200 : 常见的成功状态码
- 204 : 200状态码 - 无Body版
- 206 : 200状态码 - part版 (表示响应返回的body数据不是资源的全部)

3xx 三重

- 301 : 永久重定向
- 302 : 301状态码 - 临时版 (临时重定向)
- 304 : 不具有跳转意义,表示资源未修改,重定向到缓存 (304 Not Modified)

4xx 四败

客户端!!!

- 404 : 资源不存在或者未找到 (404 NOT FOUND)
- 400 : 请求报文有错误 (记忆方法: 400最小 发生在 Http 第一步(发送请求)中)
- 401 : 请求需要认证
- 403 : 禁止访问 (403 Forbidden)

5xx 五败

服务端!!

- 500 : 笼统错误码
- 502 : 服务器网关或者代理出现问题 (502 Bad GateWay)
- 503 : 服务器正忙,无法处理请求 (503 Unavailable)

Ming 的回答

HTTP 会通过状态码来表达请求响应的情况，常用的状态有以下的内容：

五大类 HTTP 状态码		
	具体含义	常见的状态码
1××	提示信息，表示目前是协议处理的中间状态，还需要后续的操作；	
2××	**成功，报文已经收到并被正确处理；**	200、204、206
3××	**重定向，资源位置发生变动，需要客户端重新发送请求；**	301、302、304
4××	**客户端错误，请求报文有误，服务器无法处理；**	400、403、404
5××	**服务器错误，服务器在处理请求时内部发生了错误。**	500、501、502、503

1xx：提示信息，中间状态，实际用到的比较少

2xx：表示服务器成功处理了客户端的请求

- 200 OK
- 204 No Content 也是成功，但响应头没有 body 信息
- 206 Partial Content 分块下载，断点续传，表示返回的 body 只是一部分

3xx：表示客户端的请求资源发生了变动，需要客户端用新的 URL 重新发送请求，也就是重定向

- 301 Moved Permanently 永久重定向，请求的资源不存在了
- 302 Found 临时重定向，请求资源还在
- 304 Not Modified 不具有跳转的含义，缓存重定向，用于缓存控制

4xx：客户端发送的报文有误，服务器无法处理

- 400 Bad Request 客户端请求的豹纹有错误
- 403 Forbidden 服务器禁止访问
- 404 Not Found 服务器上不存在

5xx：客户端请求报文正确，但是服务器内部发生了错误，属于服务端的错误码

- 500 Internal Server Error 服务器内部错误
- 501 Not Implemented 请求的功能还不支持，即将开业敬请期待
- 502 Bad Gateway 服务器作为网关和代理返回的错误码
- 503 Service Unavaliable 服务器很忙

4、TCP 和 UDP 协议有什么区别，分别适用于什么场景?

官方解析

TCP（Transmission Control Protocol）和 **UDP**（User Datagram Protocol）是两种常用的传输层协议，它们有以下的区别：

1、**连接方面**

TCP 是**面向连**接的协议，而 UDP 是**无连接**的协议。在 TCP 中，发送方和接收方必须先建立连接，然后才能传输数据。UDP 则不需要建立连接，直接发送数据即可。

2、**可靠性**

TCP 保证数据传输的**可靠性**，通过**序列号、确认应答和重传机制**等方式来保证数据的**完整性**和**正确性**。UDP 则不保证数据传输的可靠性，因为它不提供确认和重传机制。

3、**传输速度**

因为 TCP 要保证数据传输的可靠性，所以在传输速度方面相对较慢。而 UDP 则不需要进行复杂的传输控制，因此传输速度更快。

4、**传输内容**

TCP 是一种面向**字节流**的协议，将数据看作是一连串的字节流，没有明确的消息边界。UDP 则是面向**报文**的协议，将数据看作是一系列的报文，每个报文是一个独立的单元，具有明确的消息边界。

基于以上的特点，TCP 和 UDP 适用于不同的场景。TCP 适用于**对传输可靠性要求比较高**的场景，例如**网页浏览**、**文件传输**、**邮件**等。而 UDP 则适用于对传输可靠性要求较低、**传输速度要求较高**的场景，例如**在线游戏**、**视频直播**等。

鱼友的精彩回答

爱吃鱼蛋的回答

TCP 和 UDP 是计算机网络中两种常用的传输层协议，用于实现可靠传输和无连接传输。

TCP（Transmission Control Protocol）是一种**面向连接**的、**可靠的传输协议**。它通过**三次握手四次挥手**进行连接和断开链接，保证数据的**可靠性、完整性和顺序性**，具有较高的传输效率。

TCP 协议适用于要求**可靠传输**的场景，如**文件传输、电子邮件传输**等。

TCP协议的工作流程如下：

- 客户端向服务器发送连接请求（`SYN`）。
- 服务器收到连接请求后，回复确认请求（`SYN+ACK`）。
- 客户端收到确认请求后，回复确认（`ACK`），完成连接。
- 数据传输完成后，客户端和服务器分别发送关闭连接请求（`FIN`）。
- 对方收到关闭请求后，回复确认（`ACK`）。
- 双方都收到对方的关闭请求和确认后，关闭连接。

UDP（User Datagram Protocol）是一种**无连接**的、**不可靠**的传输协议。它不需要建立连接和维护连接状态，具有较高的传输速度和实时性，但不保证数据的完整性和顺序性。

UDP 协议适用于**实时性要求高、数据量小、丢失数据不会影响结果**的场景，如**视频直播**、**语音通话**等。 UDP协议工作流程：

- 客户端向服务器发送数据报。
- 服务器收到数据报后，直接处理数据并回复确认。
- 客户端收到确认后，继续发送下一个数据报。
- 如果数据报丢失或损坏，客户端不会重传，而是直接忽略。

两者的区别主要如下：

1. **连接方式**：TCP 是**面向连接**的协议，UDP 是**无连接**的协议。
2. **可靠性**：TCP 提供**可靠**的传输，保证数据的**完整性**和**顺序性**，而 UDP 不保证数据的完整性和顺序性。
3. **速度**：**UDP 比 TCP 更快**，因为它不需要建立连接和维护连接状态。
4. **传输方式**：TCP 是基于**字节流**的传输方式，UDP 是基于**数据报**的传输方式。

	连接方式	**可靠性**	**速度**	**传输方式**
TCP	面向连接	可靠传输，保证数据的完整性和顺序性	慢，因为需要建立连接和维护连接状态	基于字节流的传输方式
UDP	无连接	不可靠传输，不保证数据的完整性和顺序性	快，不需要建立连接和维护连接状态	基于数据报的传输方式

针对于 TCP 的特点，其应用场景主要有：

1. **文件传输**：通过 TCP 协议传输文件时，确保文件的**完整性**和**安全性**；
2. **邮件传输**：通过 TCP 协议传输邮件时，确保邮件的**完整性**和**可靠性**；
3. **网页浏览**：通过 TCP 协议传输网页时，确保网页的**完整性**和**正确性**；

针对UDP的特点，其应用场景主要有：

1. **视频流传输**：通过 UDP 协议传输视频流时，要求**实时性高**，允许数据的丢失和重复。
2. **语音通话**：通过 UDP 协议传输语音时，要求**实时性高**，允许数据的丢失和重复。
3. **游戏应用**：通过 UDP 协议传输游戏数据时，要求**实时性高**，允许数据的丢失和重复。

猫十二懿的回答

TCP（Transmission Control Protocol）和UDP（User Datagram Protocol）是两种常用的传输层协议，两者的区别比较如下：

	TCP	UDP
可靠性	可靠	不可靠
连接性	面向链接	无连接
报文	面向字节流	面向报文
效率	传输效率低	传输效率高
安全性	容易被攻击、安全性不如UDP	也会被攻击，相对TCP来说安全
可靠性	可靠，无差错、不丢失、不重复有序的传输	不可靠传输，不保证传输有序性
双工性	全双工	一对一、一对多、多对一、多对多
流量控制	滑动窗口	无
拥塞控制	慢开始、拥塞避免、快重传、快恢复	无
传输速度	慢	快
应用场景	对效率要求低，对准确性要求高或要求有链接的场景	对效率要求高、对准确性要求低

HeiHei 的回答

区别：

1. **连接**：

TCP 是面向连接的传输层协议，传输数据前要先建立连接

UDP是不需要连接，即刻传输数据

2. **服务对象**

TCP 是**一对一**的两点服务，即一条连接只有两个端点。

UDP 支持**一对一、一对多、多对多**的交互通信。

3. **可靠性**

TCP 是可靠交付数据的，数据可以**无差错、不丢失、不重复、按需到达**

UDP 是**尽最大努力交付**，不保证可靠交付数据

4. **流量控制、拥塞控制**

TCP 有**拥塞控制**和**流量控制**机制，保证数据传输的安全性

UDP则没有，即使网络非常拥堵了，也不会影响 UDP 的发送速率

5. **首部开销**

TCP 首部长度较长，会有一定的开销，首部在没有使用[选项]字段时是 20

UDP 首部只有 8 个字节，并且是固定不变的，开销较小

6. **传输方式**

TCP 是**流式传输**，**没有边界**，但保证**顺序和可靠**

UDP是**一个包一个包**地发送，是**有边界的**，但可能会**丢包和乱序**

7. **分片不同**

TCP 的数据大小如果大于 **MSS(最大报文段长度)** 大小，则会**在传输层进行分片**，目标主机收到后，也同样在传输层组装 TCP 数据包，如果中途丢失了一个分片，只需要传输丢失的这个分片

UDP 的数据大小如果大于 **MTU(最大传输单元)** 大小，则**会在 IP 层进行分片**，目标主机收到后，在 IP 层组装完数据，接着再传给传输层，**但是如果中途丢了一个分片，在实现可靠传输的 UDP 时则就需要重传所有的数据包**，这样传输效率非常差，所以**通常 UDP 的报文应小于 MTU**

8. **首部字段**

TCP 有可变长的[选项]字段

UDP 头部长度则是不会变化的，无需多一个字段区记录 UDP 的首部长度

9. **包长度字段**

TCP 无。其中 IP 总长度和 IP 首部长度，在 IP 首部格式是已知的。TCP 首部长度，则是在 TCP 首部格式已知的，所以就可以求得 TCP 数据的长度：TCP 数据的长度 = IP 总长度 - IP 首部长度 - TCP 首部长度

UDP 有 2 字节。为了网络设备硬件设计和处理方便，首部长度需要是 4 字节的整数倍

应用场景：

应用层协议	应用	传输层协议
SMTP	电子邮件	TCP
TELNET	远程终端接入	
HTTP	万维网	
FTP	文件传输	
DNS	域名转换	UDP
TFTP	文件传输	
SNMP	网络管理	
NFS	远程文件服务器	

TCP 应用场景适用于**对效率要求低，对准确性要求高或者要求有链接**的场景，而 UDP 适用场景为**对效率要求高，对准确性要求低**的场景

5、HTTP 协议中 GET 和 POST 有什么区别？分别适用于什么场景？

官方解析

HTTP 协议中 **GET** 和 **POST** 是两种常用的请求方法，它们的区别如下：

1. **参数传递方式不同**
    - GET 请求参数是在 **URL** 中以**键值对**的形式传递的，例如：http://www.example.com/?key1=value1&key2=value2。
    - 而 POST 请求参数是在**请求体**中以**键值对**的形式传递的。
2. **参数传递大小不同 GET 请求参数有大小限制**，因为 URL 长度有限制，不同的浏览器和服务器对 URL 长度的限制不同，一般为 2048 个字符。而 POST 请求参数没有大小限制，因为它们是以请求体的形式传递的。
3. **安全性不同**
    - GET 请求的参数是**明文传输**的，因为**参数在 URL 中**，如果涉及敏感信息（如密码），容易被窃取或暴露在浏览器历史记录、代理服务器日志等地方。
    - 而 POST 请求的参数在**请求体中传输**，相对**安全**一些，但是也需要注意**参数加密和防止 CSRF 攻击**等问题。
4. GET 和 POST 适用的场景不同：
    - GET 请求适用于**获取数据**，如**浏览网页、搜索**等。因为 GET 请求参数以**明文形式传输**，容易被拦截和篡改，所以不适用于提交敏感信息的操作。
    - POST 请求适用于**提交数据**，如**登录、注册、发布内容**等。因为 POST 请求参数在**请求体**中传输，相对**安全**一些，可以提交敏感信息，但是需要注意参数加密和防止 CSRF 攻击等问题。

鱼友的精彩回答

维萨斯的回答

既然评论区都有这么多答案了，那还缺我一个吗?

- 关于二者的区别 和 使用场景 （ 详见评论区 ）
    - 参数位置
    - 安全性
    - 幂等
    - 长度
    - 缓存
    - 使用场景

拓展区别：

> - GET 产生一个 TCP 数据包 ， 浏览器会把 **header** 和 **data** 一并发送出去，服务器响应 **200**
> - POST 产生两个 TCP 数据包 ， 浏览器先发送 **header**，服务器响应 **100**（continue），然后再发送 **data**，服务器响应 **200**

在网络良好的情况下，差别不大。 当然也不是所有的 POST 都发两次包，FireFox 在处理 POST 的时候，会一次性直接发送 header 和 data。**（从这里也可以看出来，GET 和 POST 之所以产生区别，是因为浏览器/服务器）**

补充说明(把 get 和 post 结合 Http):

- 关于 get 和 post 甚至是 delete,put 等方法
  - 1、他们都是 Http 发送请求的方法
  - 2、Http 是基于 TCP 的
  - 所以，其实 GET,POST 都是 TCP 连接
  - 既然如此，从 TCP 协议来理解，GET 和 POST 没有本质区别，
- 所以其实
  - **get也可以带RequestBody**
  - **post也可以有?+参数**
- 之所以出现各种差异
  - 是因为**Http 的规定和服务器/浏览器的限制**，让 get 不能让 url 超过一定的长度限制，让 get 的 RequestBody 不被服务器处理，让 Post 的 url 的路径参数（?+参数）不被处理

6、简述 TCP 三次握手、四次挥手的流程？为什么需要三次握手？为什么需要四次挥手?

官方解析

TCP 三次握手和四次挥手是 TCP 协议中建立和终止连接的过程。

三次握手的流程如下：

1. 客户端向服务器发送 SYN 包，表示请求建立连接，此时客户端进入 SYN_SENT 状态。
2. 服务器接收到 SYN 包后，回应一个 SYN-ACK 包，表示同意建立连接，此时服务器进入 SYN-RECEIVED 状态。
3. 客户端接收到 SYN-ACK 包后，向服务器回应一个 ACK 包，表示确认建立连接，此时客户端进入 ESTABLISHED 状态。服务器收到 ACK 包后也进入 ESTABLISHED 状态。

四次挥手的流程如下：

1. 客户端向服务器发送 FIN 包，表示希望关闭连接，此时客户端进入 FIN-WAIT-1 状态。
2. 服务器接收到 FIN 包后，向客户端回应一个 ACK 包，表示已经收到了关闭请求，此时服务器进入 CLOSE-WAIT 状态，客户端收到 ACK 包后进入 FIN-WAIT-2 状态。
3. 服务器关闭连接后，向客户端发送一个 FIN 包，表示可以关闭连接，此时服务器进入 LAST-ACK 状态。
4. 客户端接收到服务器的 FIN 包后，向服务器回应一个 ACK 包，表示已经收到关闭请求，此时客户端进入 TIME-WAIT 状态，等待 2MSL（最大报文生存时间）后，关闭连接。服务器收到 ACK 包后也进入 CLOSED 状态。

三次握手是为了确保客户端和服务器都能够收到对方的请求和回应，防止因为网络原因导致请求或回应丢失而建立不了连接。四次挥手则是为了确保连接的正常关闭，防止因为网络原因导致连接无法关闭或者出现连接断开后还能够接收到数据的情况。

鱼友的精彩回答

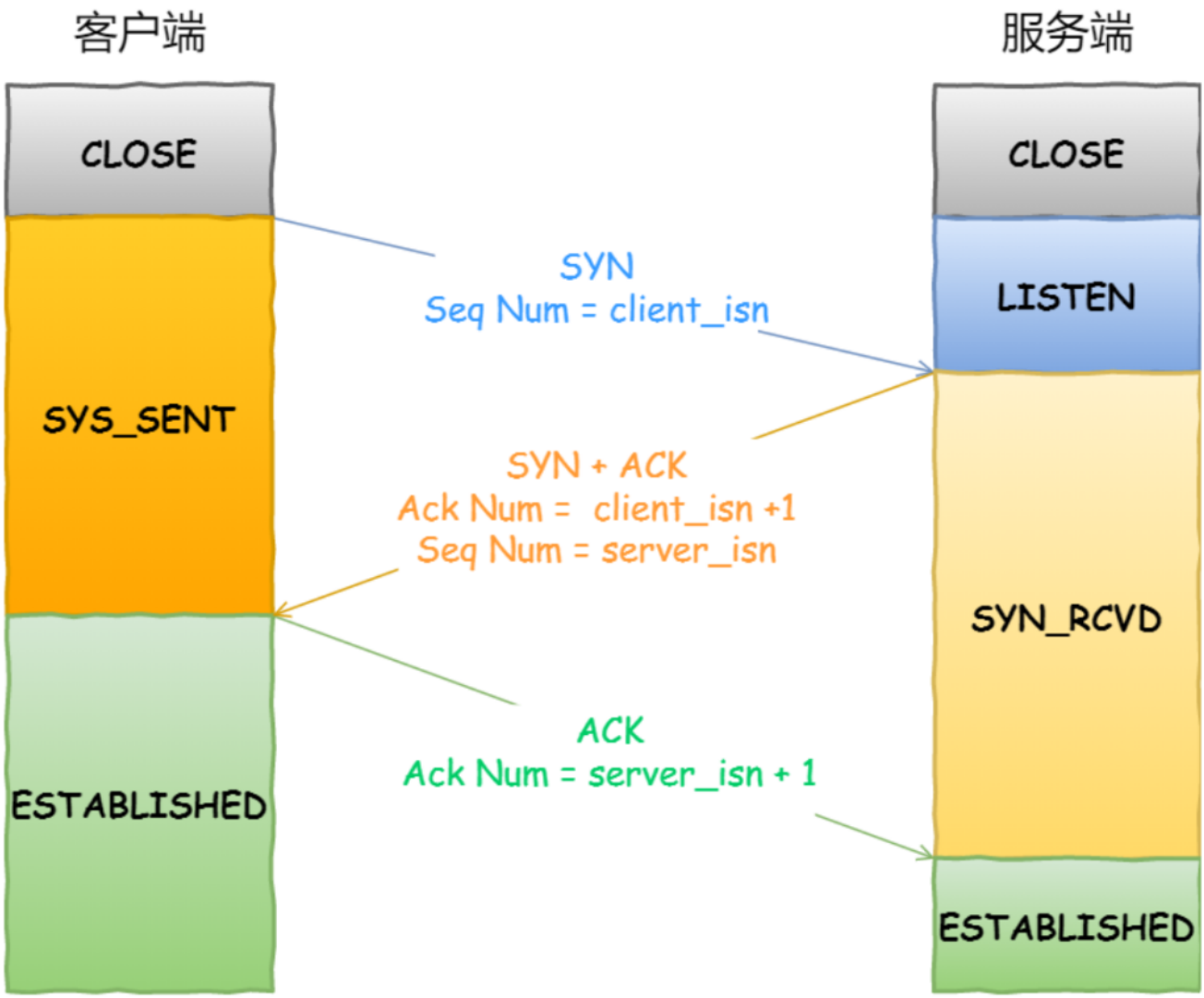
Starry 的回答

TCP（Transmission Control Protocol）是一种面向连接的协议，为了保证数据传输的可靠性，TCP 使用了三次握手和四次挥手的过程。

三次握手的过程如下：

1. 第一次握手：客户端向服务器发送 SYN 报文，请求建立连接。
2. 第二次握手：服务器收到客户端的 SYN 报文，向客户端发送 SYN+ACK 报文，表示可以建立连接。
3. 第三次握手：客户端收到服务器的 SYN+ACK 报文，向服务器发送 ACK 报文，表示连接已经建立。

如图所示：

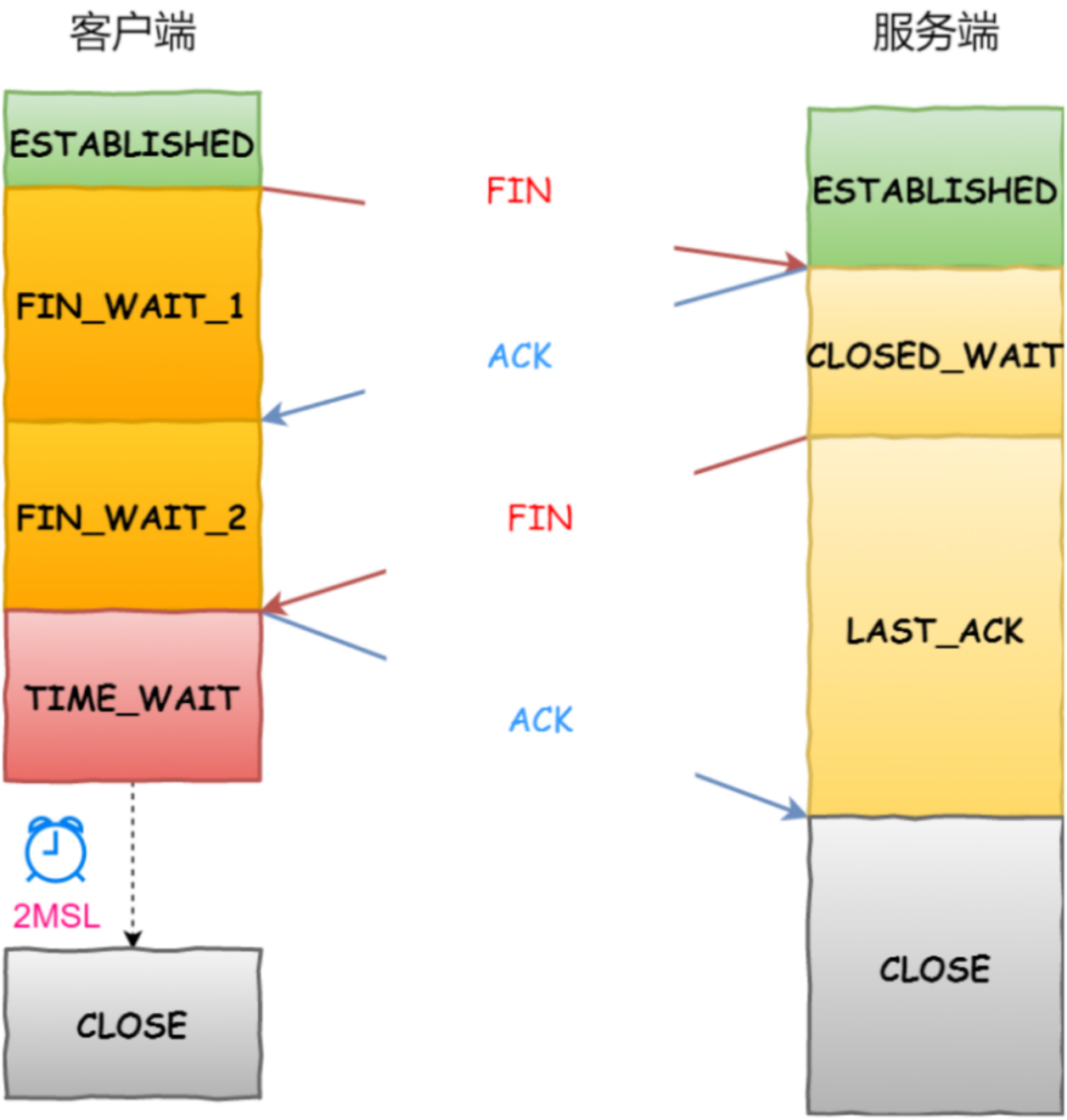


为什么需要三次握手?

三次握手的目的是为了确认双方的收发能力和同步初始序列号。

四次挥手的过程如下：

第一次挥手：客户端向服务器发送 FIN 报文，请求关闭连接。
第二次挥手：服务器收到客户端的 FIN 报文，向客户端发送 ACK 报文，表示收到关闭请求。
第三次挥手：服务器向客户端发送 FIN 报文，请求关闭连接。
第四次挥手：客户端收到服务器的 FIN 报文，向服务器发送 ACK 报文，表示收到关闭请求。
如图所示：



为什么需要四次挥手?

四次挥手的目的是为了保证数据的完整性和可靠性。在关闭连接之前，双方需要确保所有数据都已经传输完毕，因此需要通过四次挥手的过程进行确认和处理。

总结：三次握手的本质是确认通信双方收发数据的能力，四次挥手的目的是关闭一个连接。

行云的回答

TCP 三次握手

目的：

- 建立连接
- 使每一方确认对方的存在
- 允许双方进行参数的协商
- 进行资源的分配

流程

1. A 的 TCP 向 B 发送 连接请求报文段，其首部中的同步位 SYN = 1 ，并随机选择一个序号 seq = x ，表明传送数据时的第一个数据字节序号为 x。
2. B 的 TCP 收到连接请求报文段后，如果同意，则发送连接同意报文。

B 在连接同意报文段中应使 SYN = 1 ，使 ACK = 1 其确认号ack = x + 1 ,自己随机选择一个序号seq = y

3. A 收到此报文后向 B 给出确认，其 ACK = 1 ，确认号 ack = y + 1，seq = x + 1。

A 的TCP通知上层应用进程，连接已经建立

4. B 的 TCP 收到主机A的确认后，也通知其上层应用进程：TCP连接已经建立

为什么需要三次握手

不是两次的主要原因使为了**防止多次连接导致连接混乱。** 比如A主机的网络较差，连续发送了多个连接请求，B收到请求后发送连接同意报文，但是B不知道A是否收到了同意连接请求，就只能重复同意，这些过期的请求可能回导致网络的混乱 设计成三次握手，客户端在接收到服务端的同意连接报文之后，就不需要再重复发送请求连接报文，并且第三次握手客户端会发送报文给服务端，告诉服务端我已经知道你同意连接了，就不需要再同意我重复发的其他请求。 所以三次握手的原因就是避免多次建立重复连接，三次握手少了不能保证建立TCP连接成功，多了会造成资源浪费。

TCP 四次挥手

目的：释放 TCP 连接

流程：

1. 数据传输结束后，通信双方都可以释放连接 现在假设 A 向 B 已经发送完数据，A 就可以发出连接释放报文段，并停止在发送数据，主动关闭 TCP 连接
2. B 收到后。发出确认，意思我收到了，从 A 到 B 这个方向的连接就释放了(此时 A 就不能再发送数据给 B 了)，TCP 连接处于半关闭状态。B 若发送数据，A 仍需要接收
3. 当 B 发送完数据后，就可以释放连接。 B 发出的连接释放报文
4. A 收到连接释放报文后，必须发出确认。

为什么需要四次挥手：

TCP 是全双工通信，可以双向传输数据。任何一方都可以在数据传送结束后发出**连接释放**的通知，待对方**确认**后进入半关闭状态。当另一方也没有数据再发送的时候，则发出连接释放通知，对方确认后就完全关闭了 TCP 连接。

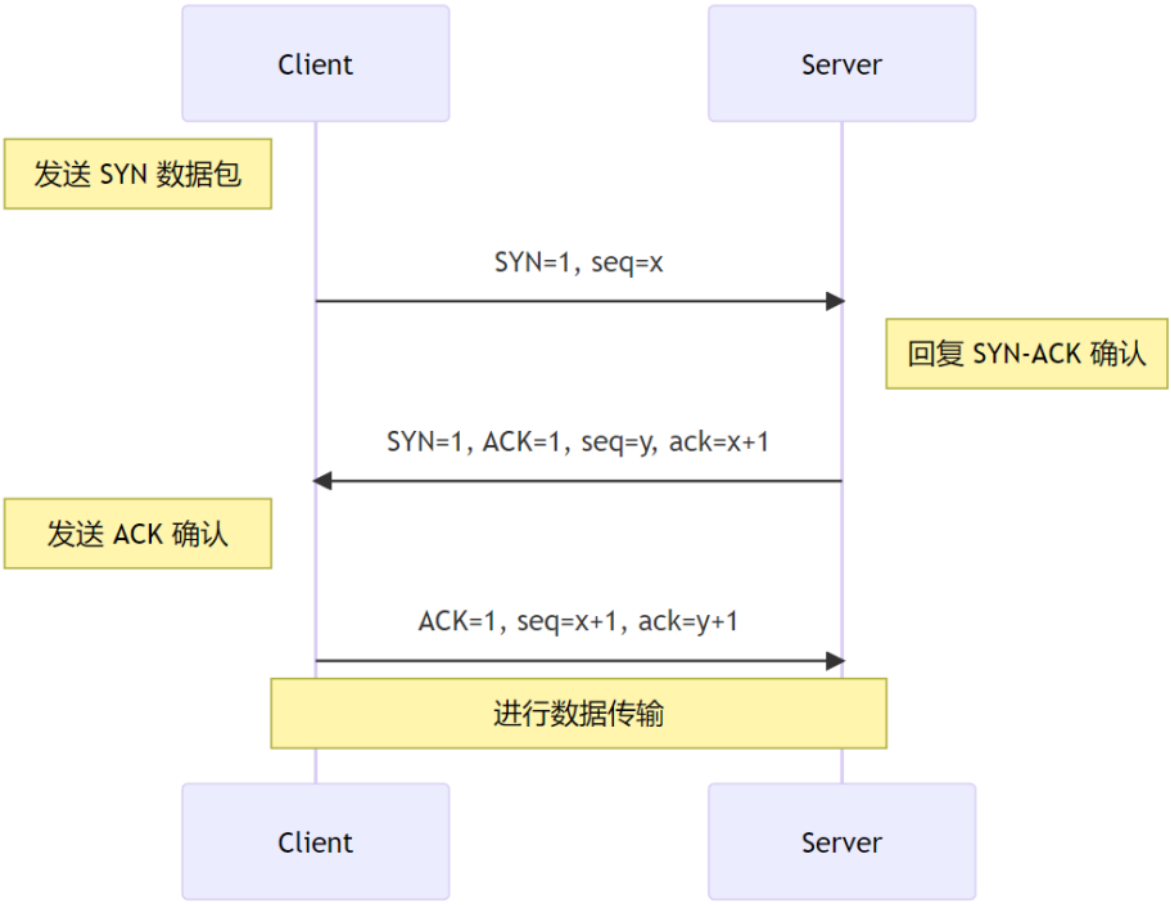
举个例子：A 和 B 打电话，通话即将结束后。

1. **第一次挥手**： A 说“我没啥要说的了”
2. **第二次挥手**：B 回答“我知道了”，但是 B 可能还会有要说的话，A 不能要求 B 跟着自己的节奏结束通话
3. **第三次挥手**：于是 B 可能又巴拉巴拉说了一通，最后 B 说“我说完了”
4. **第四次挥手**：A 回答“知道了”，这样通话才算结束。

Gianhing 的回答

TCP 三次握手：用于建立连接

1. 客户端向服务器发送 SYN 包
   客户端发送一个 SYN 数据包给服务器，请求建立连接，其中包含一个随机生成的序列号 seq = x
2. 服务器回复 SYN-ACK 确认
   服务器收到 SYN 包后，如果可以建立连接，就会回复一个 SYN-ACK 数据包，其中会包含一个随机生成的序列号 seq = y 和一个确认序列号 ack = x + 1，表示收到客户端的请求，并准备好接收数据
3. 客户端发送 ACK 确认
   客户端收到服务器的 SYN-ACK 后，会发送一个 ACK 数据包给服务器，其中会包含一个确认序列号 ack = y + 1，并且自己的序列号 seq = x + 1，表示确认收到，并准备发送数据



为什么需要三次握手?

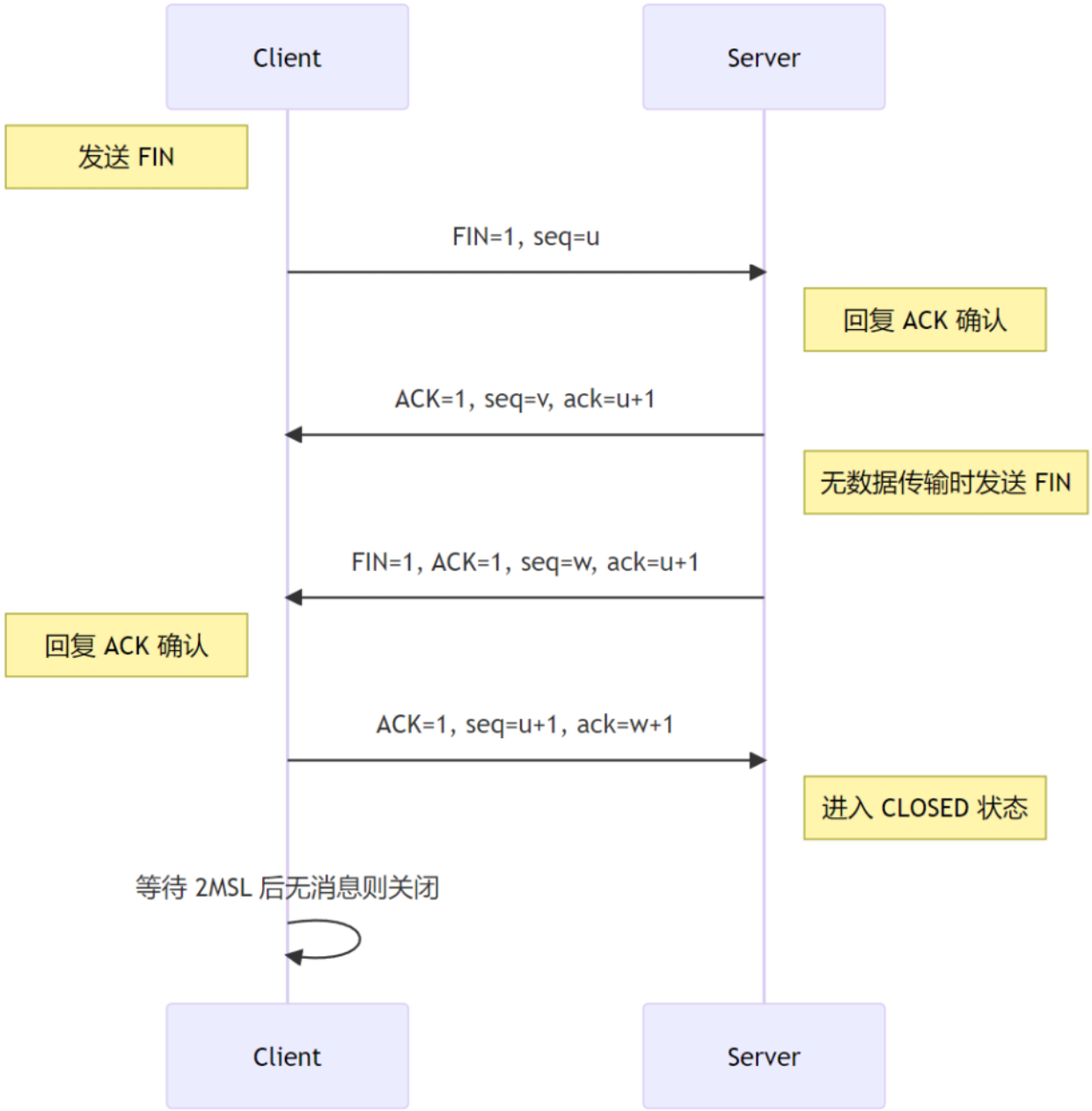
三次握手**为了客户端和服务端都能够确认对方的身份，并确保数据的发送和接收都正常。**

- 防止服务端开启一些无用的连接增加服务端的开销

- 防止重复连接造成的连接错误

TCP 四次挥手：用于释放连接

1. 客户端向服务器发送 FIN
   当客户端不需要数据传输的时候，会向服务器发送一个 FIN 数据包，请求关闭连接
2. 服务器回复 ACK 确认
   服务器收到 FIN 包之后回复 ACK 数据包，表示同意关闭连接，但可能还有数据要传输，并不会立即关闭
3. 服务器发送 FIN
   当服务器也没有数据要传输的时候，会向发送客户端发送一个 FIN 数据包，表示也准备好关闭连接
4. 客户端发送 ACK 确认
   客户端收到 FIN 包之后回复 ACK 数据包，表示收到关闭请求，此时客户端处于 TIME-WAIT 状态，等待 2MSL（最大报文存活时间）后，若没有收到数据则说明服务端已关闭，自己也可以关闭连接了



为什么需要四次挥手?

四次挥手**为了确保数据都能传输完成并正常关闭连接**。任何一方都可以在数据传送结束后发出连接释放的通知，待对方确认后进入半关闭状态。当另一方也没有数据再发送的时候，再发出连接释放通知，对方确认后就完全关闭了 TCP 连接。

> 作者：编程导航知识星球 + 星球鱼友们

操作系统

1、什么是进程和线程？它们有哪些区别和联系？

官方解析

在操作系统中，进程是指一个正在执行中的程序，而线程是进程的一部分，是一个程序中执行的代码片段。

进程是操作系统资源分配的最小单位，一个进程至少包括一个线程，进程拥有自己的内存空间、文件句柄、环境变量等系统资源。进程间相互独立，互不干扰，每个进程都拥有自己的地址空间。进程通信需要通过进程间通信机制（IPC）来实现。

线程是程序执行的最小单位，一个进程中可以包含多个线程，它们共享进程的内存空间和系统资源。多个线程可以并发执行，从而提高了程序的运行效率，同时也会带来线程安全等问题。线程之间的通信可以通过共享内存、信号量等机制实现。

进程和线程的区别和联系如下：

- 资源分配：进程拥有自己的内存空间等系统资源，而线程共享进程的资源；
- 独立性：进程之间相互独立，互不干扰，而线程是进程的一部分，线程之间共享进程的资源；
- 调度：进程间调度的开销比线程大，线程的调度开销小，可以并发执行；
- 并发性：多个进程之间相互独立，多个线程可以并发执行；
- 同步：进程间通信需要通过IPC机制，线程间同步可以通过共享内存、信号量等机制实现。

在实际开发中，多线程应用更加常见，因为线程的开销小，执行效率高，适用于需要并发执行的场景。但需要注意线程安全问题。而多进程应用通常更加稳定，但开销较大，适用于需要独立运行的场景。

鱼友的精彩回答

行云的回答

	进程	线程
资源分配	资源分配和拥有的基本单位	自己基本不拥有系统资源，但可访问所属进程的全部资源
调度	未引入线程的os中，进程是调度和分派的基本单位	进入线程的os中，线程是调度和分派的基本单位，但是线程不能脱离进程运行
地址空间	进程的地址空间之间互相独立	同一进程的各线程间共享进程的地址空间
系统开销	进程切换时设计当前cpu环境的保存及新进程cpu环境的设置，开销较大	线程切换，开销很小
独立性	各进程基本相互独立	同一进程的线程很可能相互影响

- 在Java中，当我们启动main函数时，其实就是启动了一个JVM进程，而main函数所在的线程就是这个进程中的一个线程，也称主线程。
- 多个线程共享进程的**堆**（new的对象）和**方法区**（常量池、类信息、静态变量等）。
- 而也有一些东西是线程私有的，例如程序计数器私有是为了线程切换后能恢复到正确的执行位置、虚拟机栈和本地方法栈是线程私有的，这是为了保证线程中的局部变量不被别的线程访问到

~~Anxiety dream 的回答

进程与线程的区别

1、进程是资源分配最小单位，线程是程序执行的最小单位。

2、进程有自己独立的地址空间，每启动一个进程，系统都会为其分配地址空间，建立数据表来维护代码段、堆栈和数据段，线程没有独立的地址空间，它使用相同的地址空间共享数据；

3、CPU切换一个线程比切换进程花费小

4、创建一个线程比进程开销小

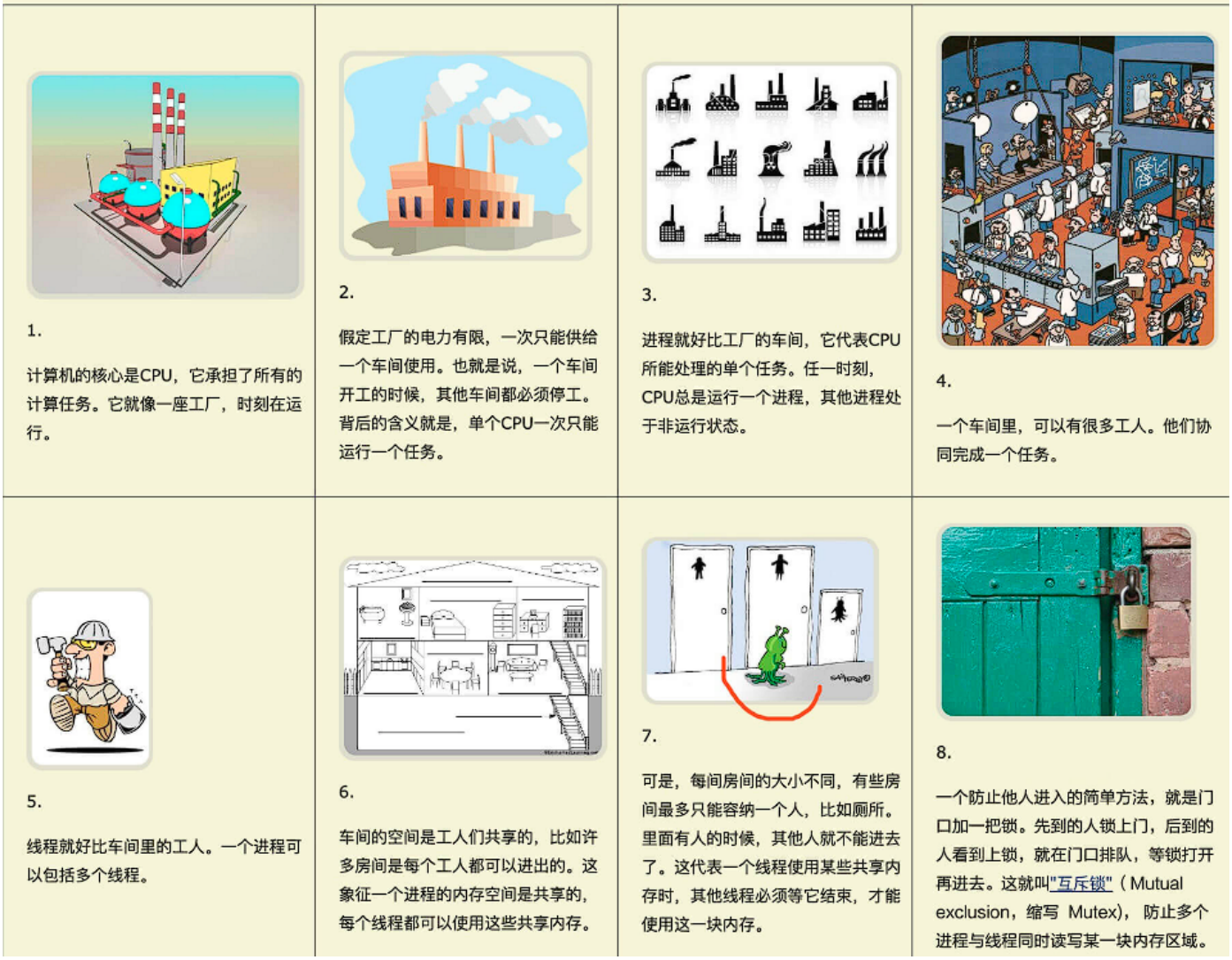
5、线程占用的资源要比进程少很多。

6、线程之间通信更方便，同一个进程下，线程共享全局变量，静态变量等数据，进程之间的通信需要以通信的方式（IPC）进行；

加强理解， 进程 = 火车 ， 线程 = 车厢

- 线程在进程下行进（单纯的车厢无法运行）
- 一个进程可以包含多个线程（一辆火车可以有多个车厢）
- 不同进程间的数据很难共享（一辆火车上的乘客很难到另外一辆火车）
- 同一进程下不同线程间的数据很容易共享（A车厢换到B车厢很容易）
- 进程要比线程消耗更多计算机资源（采用多列火车相比多个车厢更耗资源）
- 进程之间不会互相影响，一个线程挂掉将导致整个进程挂掉（一列火车不会影响到另外一列火车，但是如果一列火车上中间的一节车厢着火了，将影响到所有车厢）

年少轻狂的回答



2、死锁是什么？如何预防和避免死锁？

官方解析

死锁是指两个或多个进程在执行过程中因争夺资源而造成的一种僵局，当进程处于死锁状态时，它们将无法继续执行，而只能相互等待，直到被外部的程序干预或自行放弃。

预防和避免死锁需要采取一些措施，包括：

1. **避免资源独占**：尽量避免一个进程在获得了某些资源后再次请求其他资源，而应该将所有需要的资源一次性申请到位。
2. **避免资源持有和等待**：当一个进程占用了一些资源并等待另一些资源时，其他进程就无法使用这些资源，容易引发死锁。因此，尽可能减少资源持有和等待时间。
3. **避免资源互斥**：有些资源在同一时间只能被一个进程占用，比如打印机、磁带机等，需要采用一些技术手段来避免资源互斥的问题。

4. **引入资源剥夺策略**：当一个进程请求的资源被其他进程占用时，可以采取剥夺资源的策略，即暂停占用该资源的进程，直到该资源被释放后再恢复该进程的执行。
5. **引入进程抢占策略**：当一个进程等待时间过长时，可以采取抢占其资源的策略，即中断正在执行的进程，强制释放其占用的资源。

以上是预防和避免死锁的一些方法，具体选择哪种方法需要根据具体情况进行分析和判断。

鱼皮补充：这是后端非常高频的题目，一定要记住！建议用一些比喻帮助自己记忆

鱼友的精彩回答

Ming 的回答

段子：

- 面试官: 解释下什么是死锁?
- 应聘者: 你录用我,我就告诉你
- 面试官: 你告诉我,我就录用你
- 应聘者: 你录用我,我就告诉你
- 面试官: 滚！

死锁是什么?

多线程编程中，当两个线程为了保护两个不同的共享资源而使用了两个互斥锁，如果应用不当，会造成两个线程都在等待对方释放锁，没有外力的作用下，这些线程会一直互相等待，就没办法继续运行，这就是发生了死锁

产生死锁的四个必要条件

- 互斥条件：多个线程不能同时使用一个资源
- 持有并等待条件：线程A在等待资源2的同时并不会释放自己已经持有的资源1
- 不可剥夺条件：在自己使用之前不能被其他线程获取
- 环路等待条件：两个线程获取资源的顺序构成了环形链

如何预防和避免死锁?

只要破坏其中一个必要条件就可以避免死锁，最常见的并且可行的就是使用资源有序分配法，来破坏环路等待条件

例子1：线程之间如何避免死锁

- 线程A和线程B获取资源的顺序要一样
- 线程 A 是先尝试获取资源 A，然后尝试获取资源 B 的时候
- 线程 B 同样也是先尝试获取资源 A，然后尝试获取资源 B
- 线程 A 和 线程 B 总是以相同的顺序申请自己想要的资源

例子2：MySQL 如何避免死锁

1. 设置事务等待锁的超时时间
2. 开启主动死锁检测

其他：利用工具排查死锁问题

- java：jstack
- C：pstack + gdb

3、线程间有哪些通信方式?

官方解析

线程间通信是多线程编程中非常重要的一个概念。在多线程编程中，有时候需要让线程之间进行数据交换、协作工作。以下是几种线程间通信的方式：

- 共享内存：线程之间通过访问同一块共享内存区域来实现数据交换。
- 消息队列：一个线程向消息队列中放入一条消息，另一个线程从消息队列中取出消息。
- 管道（Pipe）：管道是一种半双工的通信方式，一个进程可以向管道中写入数据，另一个进程可以从管道中读取数据。
- 信号（Signal）：信号是一种异步通信方式，进程收到信号后，会根据信号的类型做出相应的处理。
- 互斥锁（Mutex）：用于同步访问共享资源，防止多个线程同时访问共享资源，产生冲突。
- 条件变量（Condition Variable）：用于线程之间的协调和通信，一个线程可以通过条件变量等待某个条件的出现，另一个线程可以通过条件变量通知正在等待的线程。

以上是常用的几种线程间通信方式，不同的通信方式适用于不同的场景。在实际编程中，需要根据具体情况选择合适的通信方式。

> 鱼皮补充：操作系统的基本知识，除了理论之外，还可以再延伸到你在编程时怎么让多线程共享变量（避免冲突）

鱼友的精彩回答

Gundam 的回答

共享内存：多个线程共享同一块内存空间，通过对内存的读写操作实现线程间的信息交换。可以使用 synchronized 关键字或 Lock 接口等机制来确保线程安全。

消息传递：多个线程之间通过消息传递实现信息交换。在Java中，可以使用wait()、notify()和notifyAll()等方法来实现线程间消息传递。wait()方法会使当前线程等待，直到其他线程调用notify()或notifyAll()方法唤醒它；notify()方法会随机唤醒等待队列中的一个线程；notifyAll()方法会唤醒等待队列中的所有线程。

信号量：通过信号量机制实现线程间的信息交换。Java中的 Semaphore 类就是一个信号量实现。

管道：管道是一种特殊的流，用于在线程之间传递数据。Java 中的 PipedInputStream 和 PipedOutputStream 类就是管道的实现。

RPC调用：远程过程调用（RPC）是一种跨网络进行的远程调用，可以实现在不同的线程或机器之间进行信息交换。

林风补充

线程间通信与进程间通信的区别

线程是轻量级的进程，系统进行资源调度的基本单位是进程，但是因为进程上下文切换开销太大，所以有了线程，节省开销。 线程本身也是共享进程的内存，上下文切换方便。

安全性，线程间通信的安全性相对较低，需要采用同步机制来保证共享变量的正确性；而进程间通信的安全性相对较高，进程之间相互隔离，不会对对方的内存进行非法操作。

4、什么是零拷贝？说一说你对零拷贝的理解?

官方解析

零拷贝（Zero-Copy）是一种高效的数据传输技术，它可以将数据从内核空间直接传输到应用程序的内存空间中，避免了不必要的数据拷贝，从而提高了数据传输的效率和性能。

在传统的数据传输方式中，当应用程序需要从磁盘、网络等外部设备中读取数据时，操作系统需要先将数据从外部设备拷贝到内核空间的缓冲区，然后再将数据从内核空间拷贝到应用程序的内存空间中，这个过程中需要进行两次数据拷贝，浪费了大量的 CPU 时间和内存带宽。

而使用零拷贝技术，数据可以**直接从外部设备复制到应用程序的内存空间中**，避免了**中间的内核空间缓冲区**，减少了**不必要的数据拷贝**，提高了**数据传输的效率和性能**。

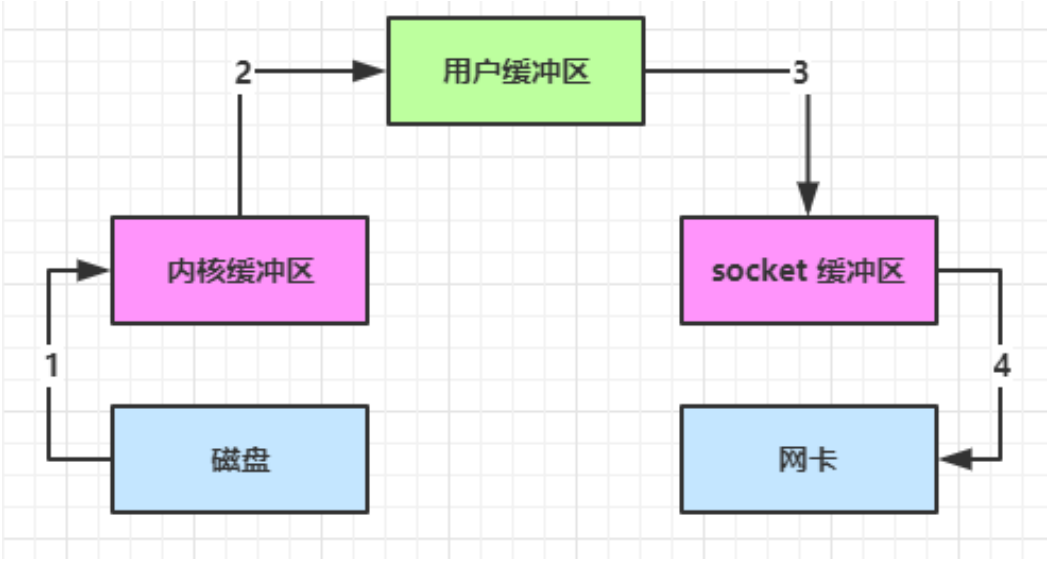
在网络编程中，零拷贝技术可以用于大文件的传输、网络文件系统的读写、数据库查询等场景中，提高数据传输的效率和响应速度。同时，零拷贝技术也可以减少系统内存的开销，提高系统的稳定性和可靠性。

鱼友的精彩回答

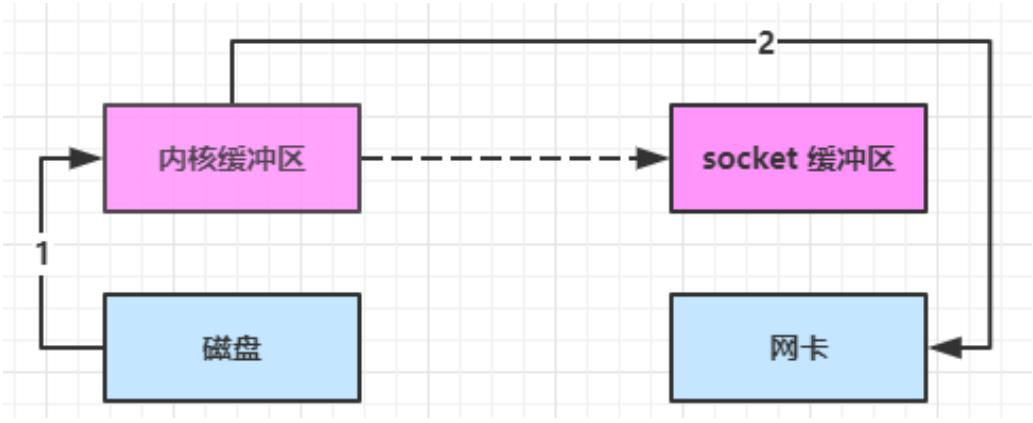
yes.的回答

零拷贝（Zero-Copy）是一种高效的数据传输技术，它可以将数据从内核空间直接传输到应用程序的内存空间中，避免了不必要的数据拷贝，从而提高了数据传输的效率和性能。

传统IO：



零拷贝：



1. java 调用 transferTo 方法后，要从 java 程序的用户态切换至内核态，使用 DMA 将数据读入内核缓冲区，不会使用 cpu
2. 只会将一些 offset 和 length 信息拷入 socket 缓冲区，几乎无消耗
3. 使用 DMA 将 内核缓冲区的数据写入网卡，不会使用 cpu

整个过程仅只发生了一次用户态与内核态的切换，数据拷贝了 2 次。**所谓的【零拷贝】，并不是真正无拷贝**，而是在不会拷贝重复数据到 jvm 内存中，零拷贝的优点有

- **更少的用户态与内核态的切换**
- **不利用 cpu 计算，减少 cpu 缓存伪共享**
- **零拷贝适合小文件传输**

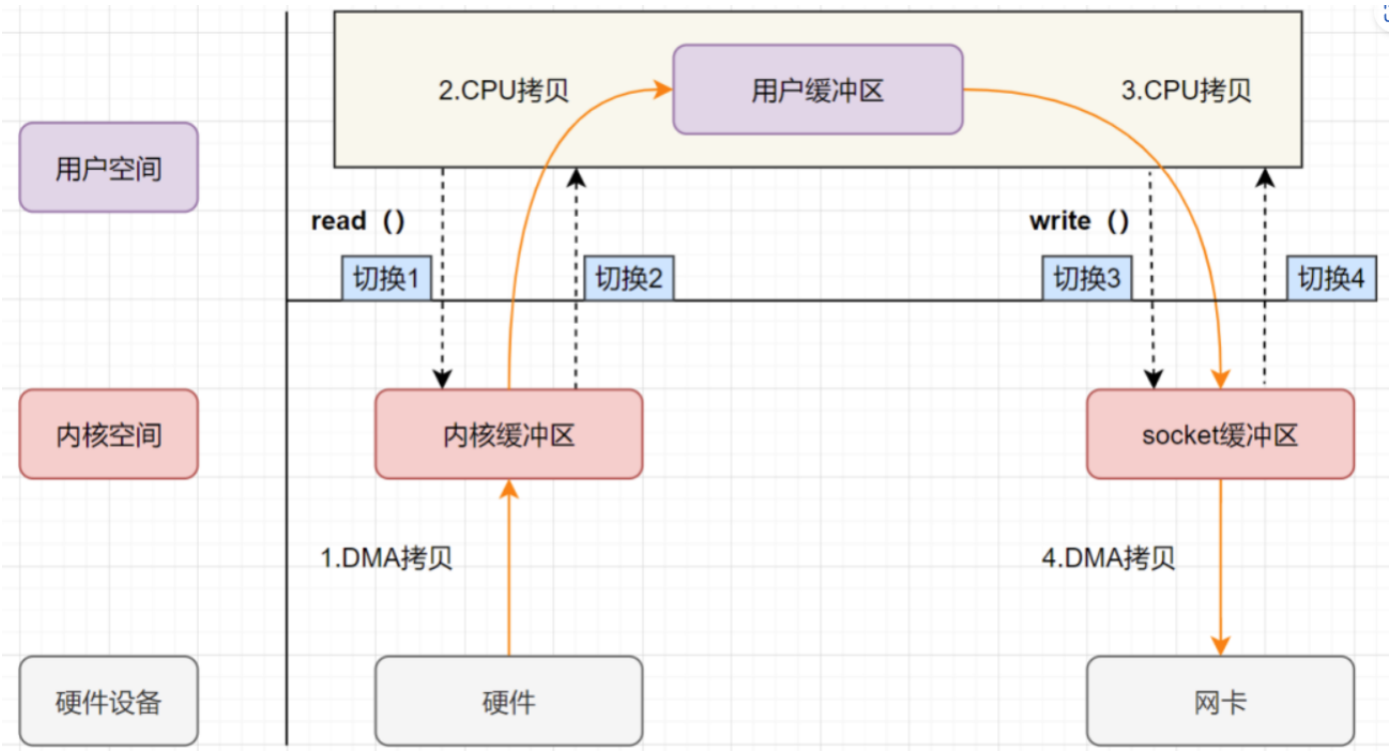
HeiHei的回答

零拷贝
零拷贝（Zero-copy）是一种技术，它可以让计算机在处理数据时，不必在内存之间来回复制数据，从而降低内存复制的开销，提高计算机的性能和效率。具体来说，零拷贝是指在数据传输过程中，**数据可以直接从发送端内存区域被传输到接收端内存区域**，而**无需在传输过程中进行任何复制操作**。从而可以减少**上下文切换以及CPU的拷贝时间**。它是一种 **I/O 操作优化技术**。

传统的 IO 执行流程

传统 IO 流过程包括 read 和 write 过程

- read:把数据从磁盘读到内核缓冲区，再拷贝到用户缓冲区
- write:先把数据写到 socket 缓冲区，再写入到网卡设备



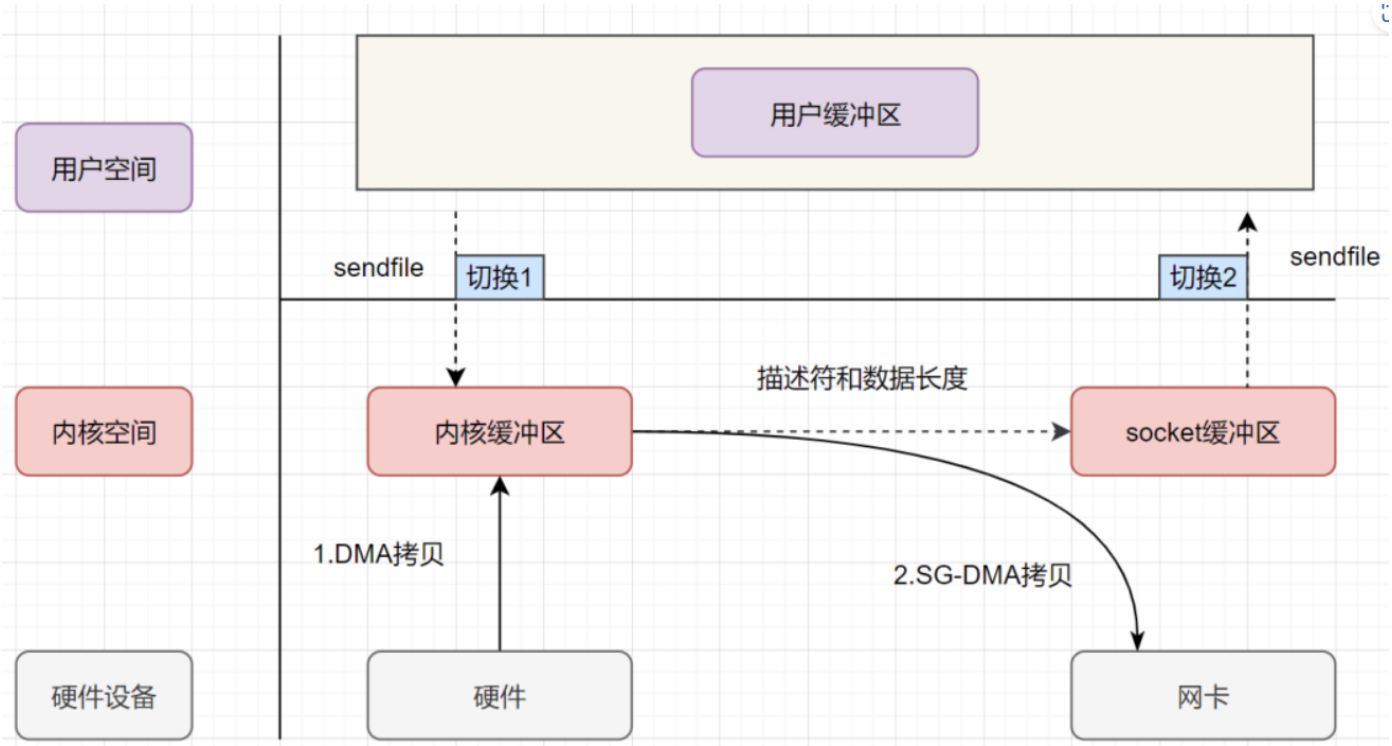
- 用户应用进程调用 read 函数，向操作系统发起 IO 调用， 上下文从用户态转为内核态（切换1）
- DMA 控制器把数据从磁盘中，读取到内核缓冲区。
- CPU 把内核缓冲区数据，拷贝到用户应用缓冲区， 上下文从内核态转为用户态（切换2），read 函数返回
- 用户应用进程通过 write 函数，发起 IO 调用，上下文从用户态转为内核态（切换3）
- CPU 将用户缓冲区中的数据，拷贝到 socket 缓冲区
- DMA 控制器把数据从 socket 缓冲区，拷贝到网卡设备， 上下文从内核态切换回用户态（切换 4），write 函数返回

从流程图可以看出，传统 IO 的读写流程，包括了 **4 次上下文切换（4 次用户态和内核态的切换），4 次数据拷贝（两次 CPU 拷贝以及两次的 DMA 拷贝)**

零拷贝实现的几种方式
这里的零拷贝其实是根据内核状态划分的，在这里没有经过 CPU 的拷贝，数据在用户态的状态下，经历了零次拷贝，所以才叫做零拷贝，但不是说不拷贝。

- mmap + write
- sendfile
- 带有 DMA 收集拷贝功能的 sendfile

sendfile+DMA scatter/gather 实现的零拷贝
sendfile 表示在两个文件描述符之间传输数据，它是在操作系统内核中操作的，避免了数据从内核缓冲区和用户缓冲区之间的拷贝操作，因此可以使用它来实现零拷贝。



- 用户进程发起 sendfile 系统调用， 上下文（切换1）从用户态转向内核态
- DMA 控制器，把数据从硬盘中拷贝到内核缓冲区。
- CPU 把内核缓冲区中的 **文件描述符信息（包括内核缓冲区的内存地址和偏移量**)发送到 socket 缓冲区
- DMA 控制器根据文件描述符信息，直接把数据从内核缓冲区拷贝到网卡
- 上下文（切换2）从内核态切换回用户态，sendfile 调用返回。

可以发现，**sendfile+DMA scatter/gather** 实现的零拷贝，I/O **发生了 2 次用户空间与内核空间的上下文切换，以及 2 次数据拷贝**。其中 2 次数据拷贝都是**包 DMA 拷贝。这就是真正的零拷贝（Zero-copy) 技术**，全程都没有通过 CPU 来搬运数据，所有的数据都是通过 **DMA** 来进行传输的。

航仔丶的回答

零拷贝（Zero-copy）是一种高效的数据传输机制，在追求低延迟的传输场景中十分常用。

它是一种 **I/O 操作优化技术**，指计算机执行 IO 操作时，CPU 不需要将数据从一个存储区域复制到另一个存储区域，从而可以减少上下文切换以及 CPU 的拷贝时间。

传统的数据传输方法需要将数据从一个缓冲区拷贝到另一个缓冲区，而零拷贝技术可以直接在内核空间中完成数据传输，避免了数据复制的过程，从而提高了数据传输的效率。

在传统的数据传输方法中，数据需要经过多次复制才能到达目标缓冲区，这个过程包含了用户空间和内核空间之间的多次上下文切换，这些上下文切换会导致系统的性能下降。而零拷贝技术可以避免这些上下文切换，从而提高了系统的性能。例如，**当应用程序需要将大量的数据写入到磁盘或者网络中时，使用零拷贝技术可以减少数据复制的过程，提高数据传输的效率**。

零拷贝技术的实现主要有以下几种方式：

- **文件描述符传递**：通过文件描述符的传递，将数据从一个进程传递到另一个进程，避免了数据复制的过程。

- **sendfile**()函数：在 Linux 系统中，sendfile()函数可以将一个文件描述符指向的文件内容直接传输到一个 socket 文件描述符中，避免了数据复制的过程。
- **mmap**()函数：mmap()函数可以将一个文件映射到内存中，避免了数据复制的过程。

总之，零拷贝是一种高效的数据传输机制，在追求低延迟的传输场景中应用广泛。使用零拷贝技术可以避免数据复制的过程，减少上下文切换，提高系统的性能。

分布式

1、什么是分布式？为什么需要分布式?

官方解析

分布式是指在多台计算机上协同工作的系统，这些计算机通过网络连接在一起，共同完成一个任务。

分布式系统能够有效地解决单台计算机处理能力不足、系统容易宕机、数据存储容量有限等问题，同时能够提高系统的可靠性、可用性和性能，适用于数据量较大、并发量高、访问频繁的场景。

此外，分布式系统还可以通过横向扩展的方式提高系统的性能和可靠性，同时降低单点故障的风险，提高了系统的可伸缩性，方便进行升级和维护。

在分布式系统中，由于数据和计算任务被分布在多台计算机上，不同计算机之间需要进行通信和协调，因此需要解决分布式一致性、负载均衡、故障恢复、数据共享和安全等问题，同时需要考虑数据的一致性和可靠性。因此，分布式系统的设计和实现比单机系统更加复杂和困难，需要考虑到多个因素的综合影响。

2、什么是网关，网关有哪些作用?

官方解析

网关（Gateway）是在计算机网络中用于**连接两个独立的网络的设备**，它能够在两个不同协议的网络之间传递数据。在互联网中，网关是一个**可以连接不同协议的网络的设备**，比如说可以**连接局域网和互联网**，它可以把**局域网的内部网络地址转换成互联网上的合法地址**，从而使得局域网内的主机可以与外界通信。

在计算机系统中，网关可以用于实现负**载均衡、安全过滤、协议转换**等功能。具体来说，网关可以分为以下几种：

- **应用网关**：用于应用层协议的处理，如 HTTP、SMTP 等。
- **数据库网关**：用于数据库访问的控制和管理。
- **通信网关**：用于不同通信协议之间的数据交换，如 TCP/IP、UDP/IP 等。
- **API 网关**：用于管理和转发 API 请求，实现 API 的授权、限流、监控等功能。

总的来说，网关可以为不同网络提供连接和通信的功能，同时也可以提供安全、性能、可靠性等方面的增强功能，是现代计算机系统中不可或缺的一部分。

> 鱼皮补充：API 网关这里，大家可以提及 Spring Cloud Gateway；应用层网关可以想到 Nginx、HA Proxy 等

鱼友的精彩回答

Starry 的回答

网关（Gateway）是连接两个或多个不同网络的设备，可以实现**协议的转换、数据的转发和安全策略的实现**等功能。简单来说，网关是**设备与路由器之间的桥梁**，由它将不同的网络间进行访问的控制，转换，交接等等。

常见的网关有**应用网关、协议网关、安全网关**等。

网关的作用如下：

- **实现协议的转换**：不同网络之间通常使用不同的协议，通过网关可以实现协议的转换，使得不同网络之间能够相互通信。
- **提供数据转发功能**：网关可以对传输的数据进行过滤、路由、转发等处理，确保数据的可靠传输。
- **实现安全策略**：网关可以对传输的数据进行加密、认证、授权等操作，保证数据的安全性和可靠性。
- **提供缓存功能**：网关可以将一部分数据缓存起来，提高数据的访问速度和响应性能。
- **支持负载均衡**：网关可以将请求分配到不同的服务器上，实现负载均衡，提高系统的可用性和性能。
- **实现访问控制**：网关可以对访问进行控制，防止未授权的访问和攻击，提高系统的安全性。

mos 的回答

在微服务架构里，服务的粒度被进一步细分，各个业务服务可以被独立的设计、开发、测试、部署和管理。这时，各个独立部署单元可以用不同的开发测试团队维护，可以使用不同的编程语言和技术平台进行设计，这就要求必须使用一种语言和平 台无关的服务协议作为各个单元间的通讯方式。

API 网关的定义

网关的角色是作为一个 API 架构，用来保护、增强和控制对于 API 服务的访问。

API 网关是一个处于应用程序或服务（提供 REST API 接口服务）之前的系统，用来**管理授权、访问控制和流量限制**等，这样 REST API 接口服务就被 API 网关保护起来，对所有的调用者透明。因此，隐藏在 API 网关后面的业务系统就可以专注于创建和管理服务，而不用去处理这些策略性的基础设施。

职能

功能

网关可以分为以下几种：

- **应用网关**：用于应用层协议的处理，如 HTTP、SMTP 等。
- 数据库网关：用于数据库访问的控制和管理。
- 通信网关：用于不同通信协议之间的数据交换，如 TCP/IP、UDP/IP 等。 API 网关：用于管理和转发 API 请求，实现 API 的授权、限流、监控等功能。

Antony 的回答

网关，即 Gateway，网关出现的原因是微服务架构的出现，不同的微服务一般会有不同的网络地址，而外部客户端可能需要调用多个服务的接口才能完成一个业务需求。如果让客户端直接与各个微服务通信，会有以下的问题：

- 客户端会多次请求不同的微服务，增加了客户端的复杂性；
- 存在跨域请求，在一定场景下处理相对复杂；
- 认证复杂，每个服务都需要独立认证；
- 难以重构，随着项目的迭代，可能需要重新划分微服务。例如，可能将多个服务合并成一个或者将一个服务拆分成多个。而划分出多个微服务就代表可能出现多个新的访问地址，如果客户端直接与微服务通信，那么重构将会很难实施；
- 某些微服务可能使用了防火墙/浏览器不友好的协议，直接访问会有一定的困难；

那么使用网关的好处就如下：

- 路由
- 负载均衡
- 统一鉴权
- 跨域
- 统一业务处理
- 访问控制
- 发布控制
- 流量染色
- 接口保护
- 统一日志
- 统一文档

常见的网关产品有 Tyk，Kong，Zuul 以及 Spring Cloud Gateway

3、Dubbo 是什么？是否了解过它的架构设计?

官方解析

Dubbo 是一个高性能、轻量级的开源 Java **RPC 框架**，它提供了完整的 RPC 协议栈，包括**服务发布、服务引用、负载均衡、容错、服务治理和服务监控**等功能，同时提供了可扩展的 RPC 协议、数据模型、序列化和网络传输等组件，支持多语言和多协议。

Dubbo 的架构设计主要包括**服务提供者、服务消费者、注册中心和监控中心**四个角色。其中，服务提供者提供服务的实现，并通过注册中心将自己注册到服务治理中心；服务消费者则通过注册中心发现可用的服务，并通过负载均衡策略选择一个服务提供者进行调用；注册中心主要负责服务的注册、发现和路由；监控中心则负责服务的统计和监控。

Dubbo 的架构设计采用了分层架构模式，将不同的功能模块进行分离，以达到模块化和可扩展的目的。同时，Dubbo 还提供了丰富的扩展点和插件机制，用户可以通过自定义扩展点和插件来满足不同的业务需求。

> 鱼皮补充：其实这道题的答案就在 dubbo 官方文档中，大家学 dubbo 的时候建议仔细阅读下官方文档

鱼友的精彩回答

Starry 的回答

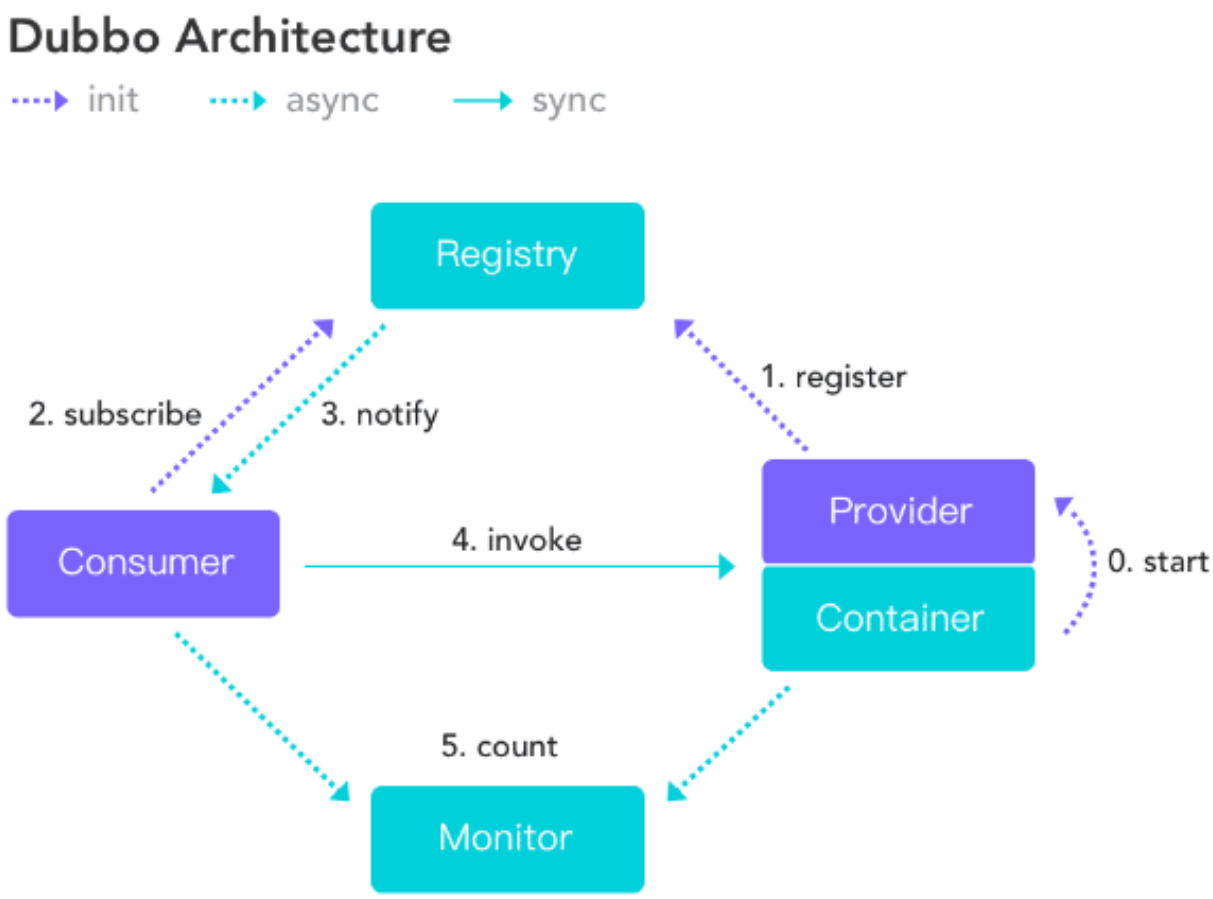
Apache Dubbo 是一款高性能的 **Java RPC** 框架。其前身是阿里巴巴公司开源的一个高性能、轻量级的开源 Java RPC 框架，可以和 Spring 框架无缝集成。

什么是RPC?

RPC 全称为 remote procedure call，即远程过程调用。

比如两台服务器 A 和 B，A 服务器上部署一个应用，B 服务器上部署一个应用，A 服务器上的应用想调用B 服务器上的应用提供的方法，由于两个应用不在一个内存空间，不能直接调用，所以需要通过网络来表达调用的语义和传达调用的数据。

Dubbo 架构图：



各节点角色说明：

节点	角色名称
Provider	暴露服务的服务提供方
Consumer	调用远程服务的服务消费方
Registry	服务注册与发现的注册中心
Monitor	统计服务的调用次数和调用时间的监控中心
Container	服务运行容器

调用关系说明：

1. 服务容器负责启动，加载，运行服务提供者。
2. 服务提供者在启动时，向注册中心注册自己提供的服务。
3. 服务消费者在启动时，向注册中心订阅自己所需的服务。
4. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
5. 服务消费者，从提供者地址列表中，基于负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。
6. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

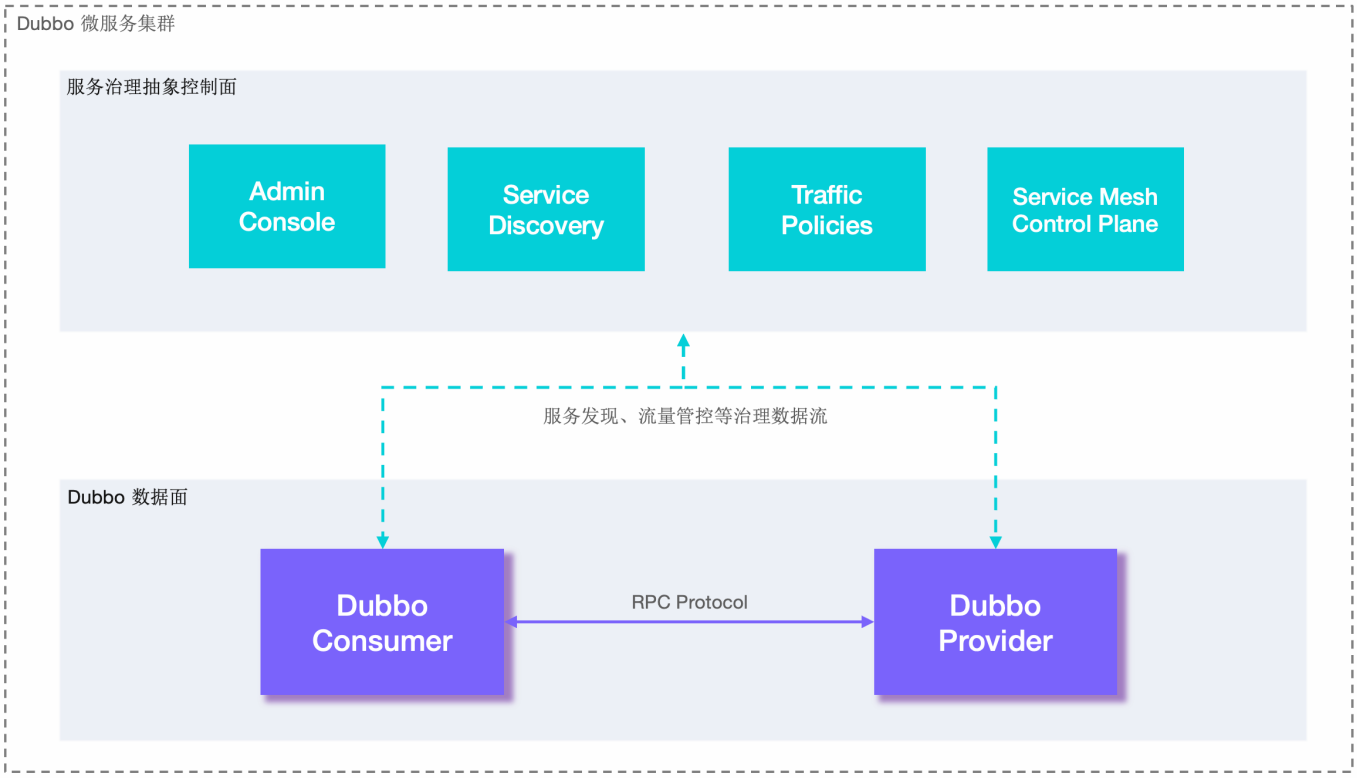
维萨斯的回答

Apahce 提供的 RPC 框架,前身是 alibaba。

- (这一点重要也不重要)
- 重要： 因为实际开发中如果版本没控制好(点名 SpringBoot 较新版本和 alibaba 的旧 Dubbo)，容易报一些莫名奇妙的错误。
- 不重要：Dubbo 现在已经到 3.x 系列，现在大家用 Apache 的 Dubbo3.x 系列基本不会出现版本问题

根据官方文档（March 1, 2023）结构如下

分为 服务治理控制面 + Dubbo 数据面（图一是现在官方给的图



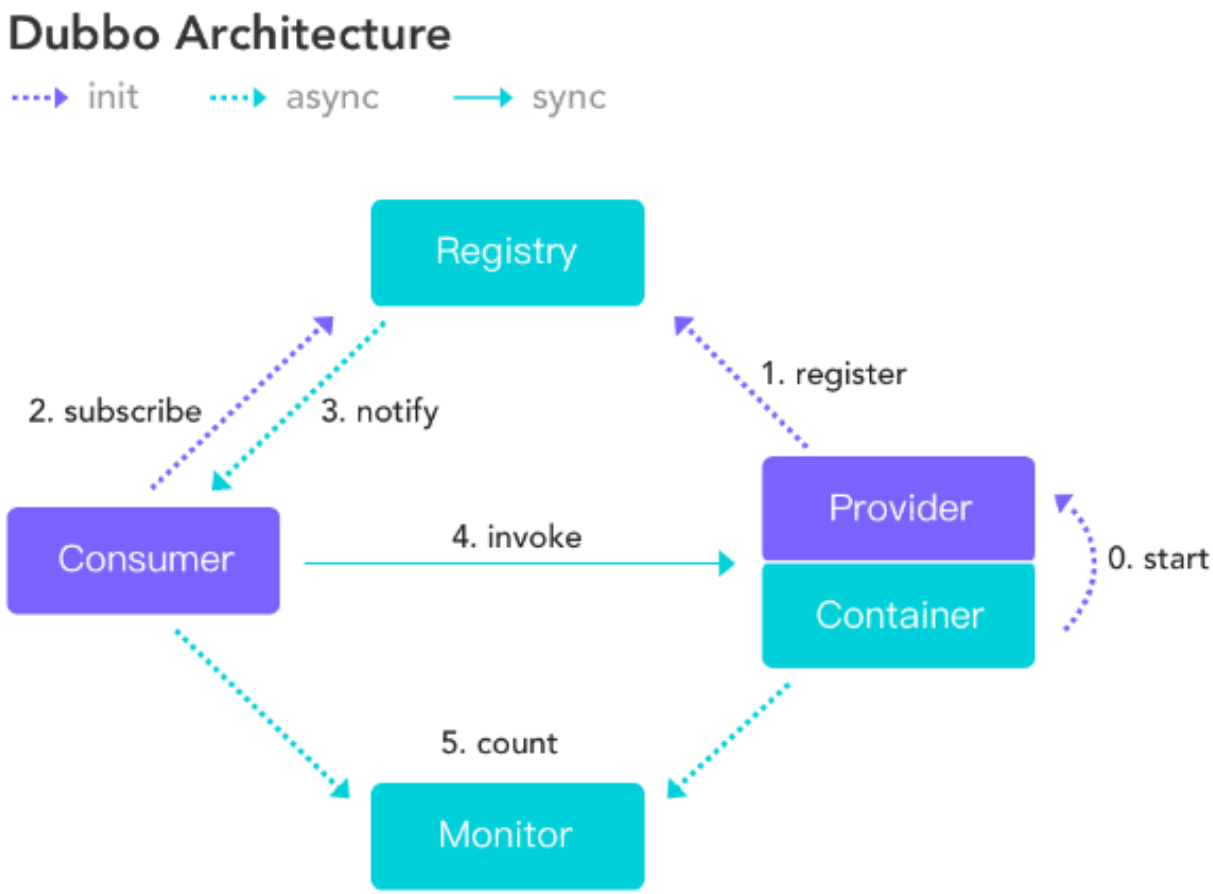
在服务控制面：

- 包含协调服务发现的注册中心、流量管控策略、Dubbo Admin 控制台、负载均衡等一系列的实现

在 Dubbo 数据面：

- 约束了微服务定义、开发与调用的规范，定义了服务治理流程及适配模式 作为 RPC 通信协议实现 解决服务间数据传输的编解码问题

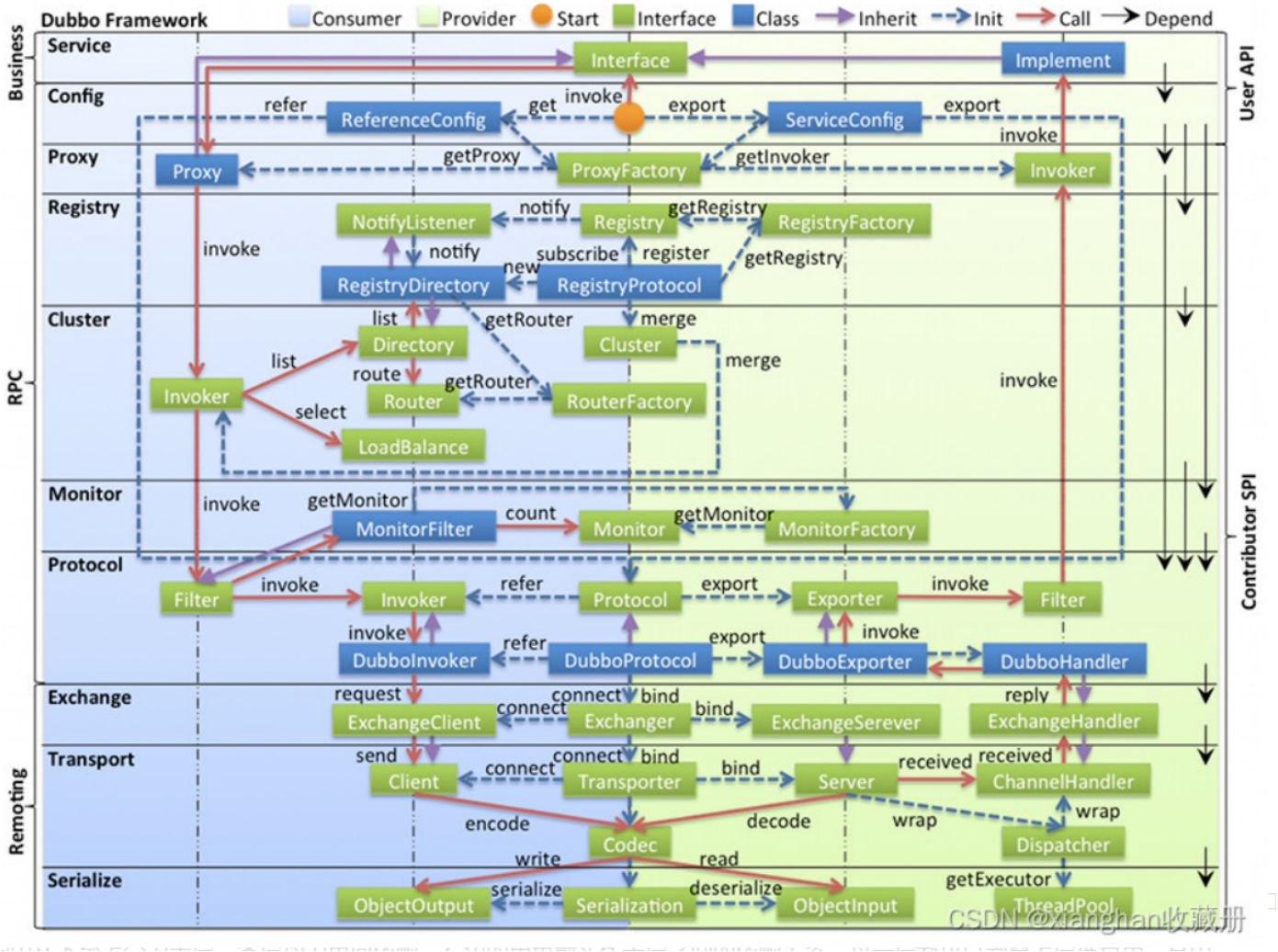
感觉新的太抽象了，还是以前的 Dubbo 框架图典中典



HeiHei 的回答

Dubbo 是一种基于 Java 语言的高性能分布式服务框架，用于简化不同系统之间的通信和协调。它可以帮助用户构建高性能、可扩展、可靠的RPC服务，并支持多种传输协议、负载均衡、服务路由、服务治理、服务监控等功能

架构设计从分层理解：

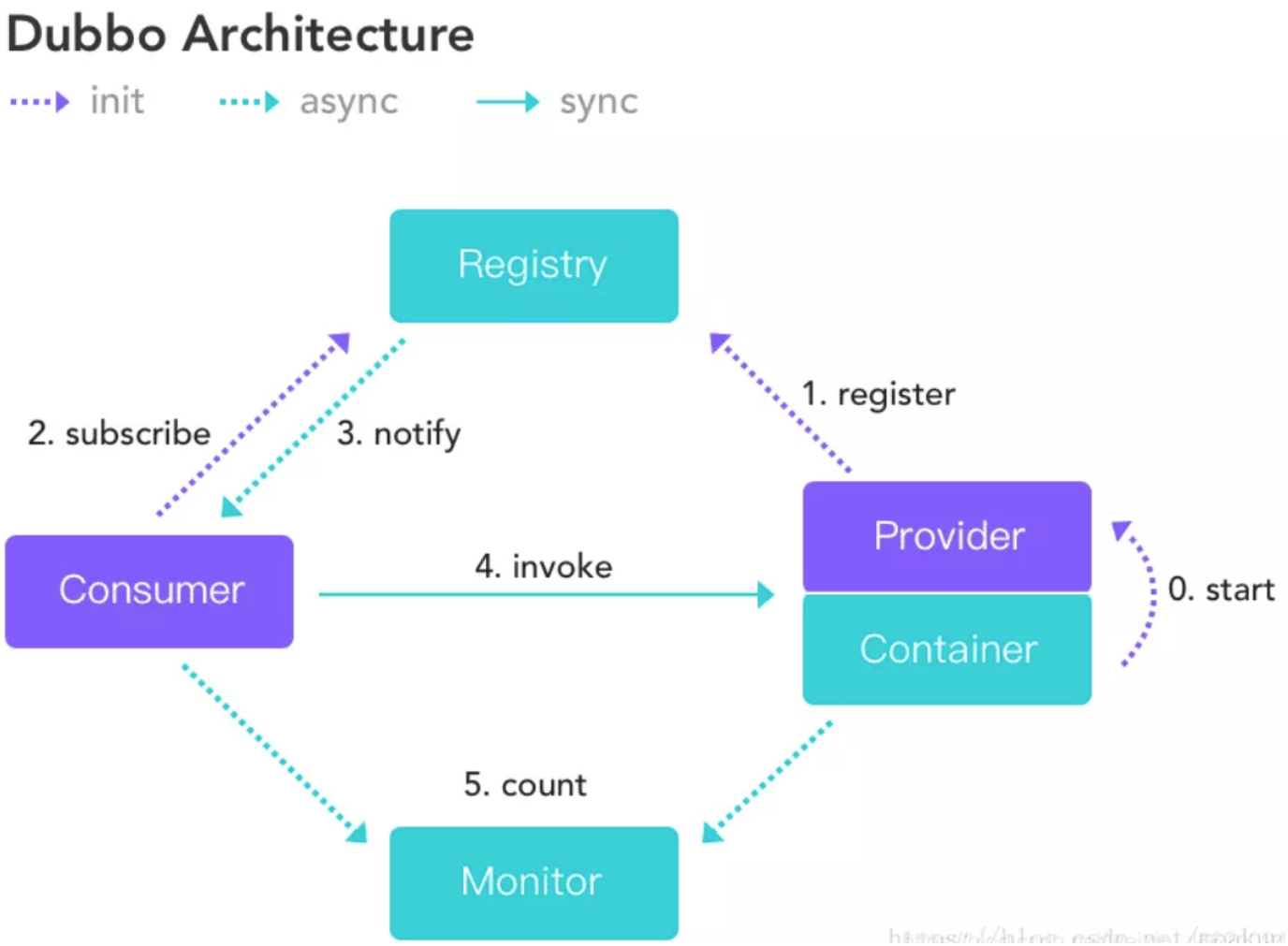


Dubbo 的整体架构可以分为三层：`业务层、RPC 层、Remoting 层`

- **接口服务层**（Service）：与业务逻辑相关，根据 provider 和 consumer 的业务设计对应的接口和实现。
- **配置层**（Config）：用来初始化配置信息，用来管理Dubbo的配置。
- **服务代理层**（Proxy）：服务接口透明代理，provider和consumer都会生成Proxy，它用来调用远程接口。生成服务的客户端Stub和服务端的Skeleton，以ServiceProxy为中心，扩展接口为ProxyFactory。
- **服务注册层**（Registry）：封装服务地址的注册和发现，以URL为中心，扩展接口为RegistryFactory、Resitry、RegistryService。
- **路由层**（Cluster）：封装多个提供者的路由和负载均衡，并桥接注册中心，扩展接口为 Cluster、Directory、Router 和 LoadBlancce。
- **监控层**（Monitor）：PRC调用次数和调用时间监控，以Statistics为中心，扩展接口为 MonitorFactory、Monitor 和 MonitorService。
- **远程调用层**（Protocol）：封装RPC调用的具体过程，以 Invocation 和 Result 为中心，扩展接口为 Protocal、Invoker 和 Exporter。
- **信息交换层**（Exchange）：封装请求响应模式，同步转异步，以 Request 和Response 为中心，扩展接口为 Exchanger、ExchangeChannel、ExchangeClient 和 ExchangeServer。
- **网络传输层**（Transport）：将网络传输封装成统一接口，可以在这之上扩展更多的网络传输方式，扩展接口为 Channel、Transporter、Client、Server 和 Codec。 10.数据序列层（Serialize）：负责网络传输的序列化和反序列化，扩展接口为 Serialization、ObjectInput、ObjectOutput 和 ThreadPool。

架构设计从角色理解：

Dubbo的架构设计主要包括服务提供者、服务消费者、注册中心和监控中心四个角色。其中，服务提供者提供服务的实现，并通过注册中心将自己注册到服务治理中心；服务消费者则通过注册中心发现可用的服务，并通过负载均衡策略选择一个服务提供者进行调用；注册中心主要负责服务的注册、发现和路由；监控中心则负责服务的统计和监控。



节点	角色说明
Provider	暴露服务的服务提供方
Consumer	调用远程服务的服务消费方
Register	服务注册与发现的注册中心
Monitor	统计服务的调用次数和调用时间的监控中心
Container	服务运行容器

> 作者：编程导航知识星球 + 星球鱼友们

4、什么是分布式的 CAP 理论?

官方解析

分布式的 CAP 理论是指在分布式系统中，**一致性**（Consistency）、**可用性**（Availability）**和分区容错性**（Partition Tolerance）这三个指标无法同时满足的问题。具体来说：

- **一致性**（Consistency）：指多个副本之间数据保持一致，即在一个副本上的写操作会立即同步到其他所有副本，所有副本的数据都是最新的，保持**强一致性**。
- **可用性**（Availability）：指系统在任何时候都能对外提供服务，即**系统随时能够响应用户请求**，不会因为节点故障或其他原因而导致服务中断。
- **分区容错性**（Partition Tolerance）：指系统在出现**网络分区**（**节点之间失去联系**）时，仍能够继续工作，保证数据的一致性和可用性。

CAP 理论指出，**一个分布式系统只能同时满足其中的两个指标，无法同时满足三个**。

例如，当出现**网络分区**时，如果要**保证一致性**，就必须**停止对外服务**，从而**失去可用性**；如果要**保证可用性**，就必须**放弃一致性**，从而可能**导致不同节点之间数据不一致**。

因此，在设计分布式系统时，需要根据具体的场景和需求来选择合适的权衡方案，比如选择 **CP（一致性和分区容错性）**或者选择 AP**（可用性和分区容错性）**。

需要注意的是，CAP 理论只是一种理论框架，不能直接应用于实际的分布式系统设计。**在实际应用中，还需要考虑系统的具体业务需求、数据访问模式、节点规模和部署环境等因素**，综合权衡之后再选择合适的分布式架构和技术方案。

鱼友的精彩回答

Gianhing 的回答

CAP 理论是指一个分布式系统中，不可能同时满足以下三个条件：

- **一致性**（Consistency）：所有节点在同一时间的看到的数据是一致的，即写**数据操作时要同时更新相关副本**，保证**强一致性**
- **可用性**（Availability）：系统中能正常接收请求的节点都能在合理时间内返回结果，即**系统在某些节点失效下仍能对外提供服务**
- **分区容错性**（Partition Tolerance）：什么是分区？分布式系统中存在很多节点，这些节点之间通过网络进行通信，**当节点间的通信出了问题（如网络故障、机器故障等），就称系统出现了分区**。而**分区容错性就是出现分区问题时，系统还能继续对外提供服务**

根据 CAP 理论，分布式系统只能满足其中的两个特性。然而实际上，**分区容错性是一定要满足的**，因为不可能只要出现分区问题时整个系统就完全无法使用。因此，在分布式系统中，我们需要考虑的是当出现分区问题时，**选择的是一致性还是可用性，即 CP 还是 AP**。

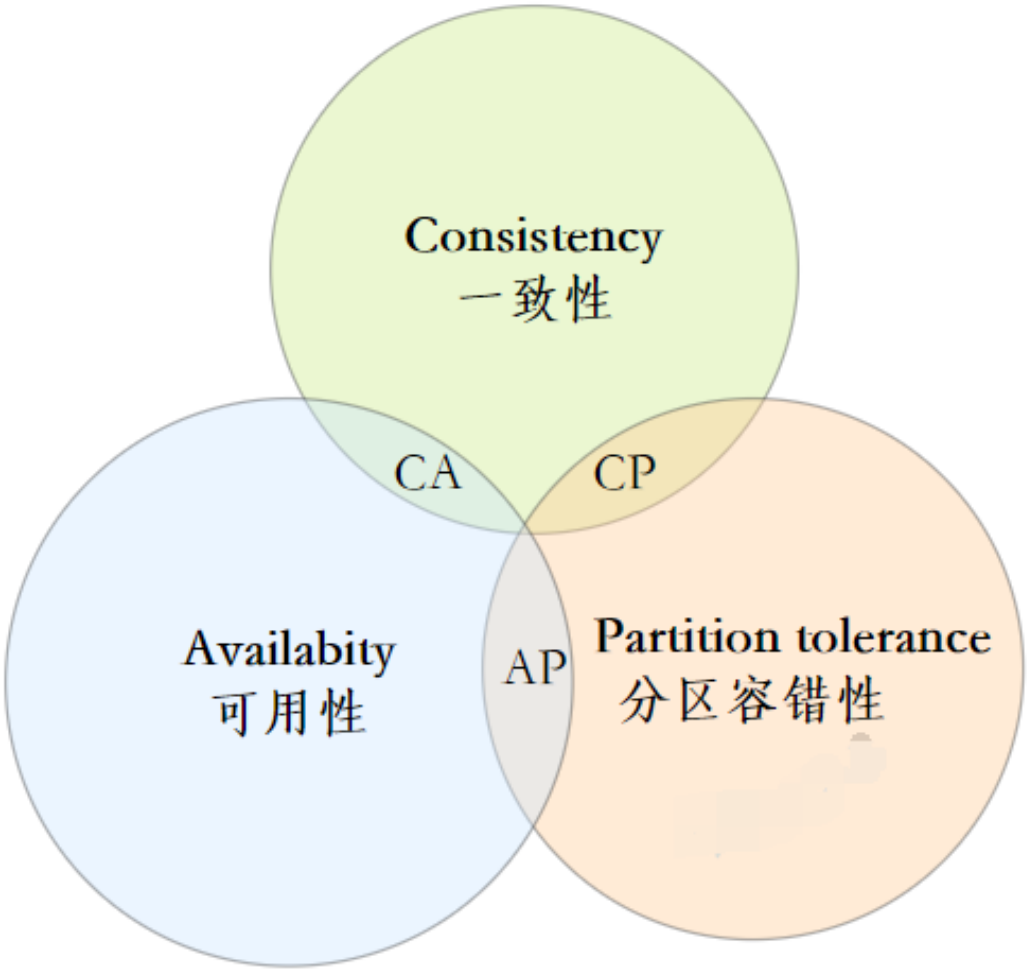
- **CP 架构**：当系统出现分区故障时，客户端发送的任意请求都会被卡死或超时，保证数据的**强一致性**。如 **Zookeeper**

- **AP 架构**：当系统出现分区故障时，**客户端依旧能获取数据，但有的是新数据，有的是旧数据**。如 **Eureka**

> 鱼皮评论：不错

猫十二懿的回答

CAP 原则又称 CAP 定理，指的是在一个分布式系统中，`Consistency（一致性）`、`Availability（可用性）`、`Partition tolerance（分区容错性）`这三个基本需求，最多只能同时满足其中的 2 个。



一致性：数据在多个副本之间能够保持一致的特性。

可用性：系统提供的服务一直处于可用的状态，每次请求都能获得正确的响应。

分区容错性：分布式系统在遇到任何网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务。

CAP 三者不可同得，那么必须得做一些权衡。

CA without P

如果不要求 P（不允许分区），则 C（强一致性）和 A（可用性）是可以保证的。但是对于分布式系统，分区是客观存在的，其实**分布式系统理论上是不可选 CA 的**。

CP without A

如果不要求 A（可用），相当于每个请求都需要在 Server 之间**强一致**，而 P（分区）会导致同步时间无限延长，如此 CP 也是可以保证的。很多**传统的数据库分布式事务**都属于这种模式。

AP wihtout C

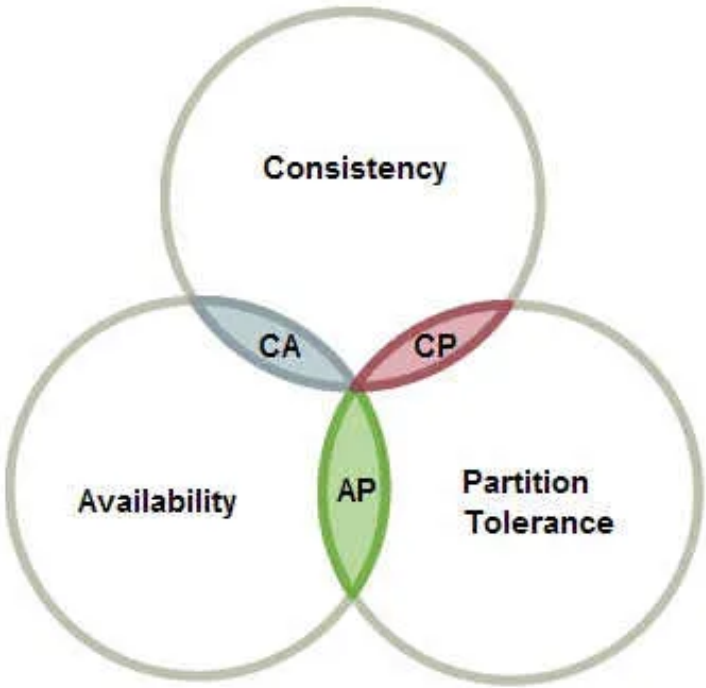
要高可用并允许分区，则需放弃一致性。一旦分区发生，节点之间可能会失去联系，为了高可用，每个节点只能用**本地数据**提供服务，而这样会导致**全局数据的不一致性**。现在众多的 NoSQL 都属于此类。

mos 的回答

CAP即：

- **Consistency**（一致性）
- **Availability**（可用性）
- **Partition tolerance**（分区容忍性）

这三个性质对应了分布式系统的三个指标： 而 CAP 理论说的就是：一个分布式系统，不可能同时做到这三点。如下图：



①**一致性**：对于客户端的每次读操作，**要么读到的是最新的数据，要么读取失败**。换句话说，一致性是站在分布式系统的角度，对访问本系统的客户端的一种承诺：**要么我给您返回一个错误，要么我给你返回绝对一致的最新数据**，不难看出，其强调的是**数据正确**。

②**可用性**：**任何客户端的请求都能得到响应数据**，不会出现响应错误。换句话说，可用性是站在分布式系统的角度，对访问本系统的客户的另一种承诺：**我一定会给您返回数据，不会给你返回错误，但不保证数据最新**，强调的是**不出错**。

③**分区容忍性**：由于分布式系统通过网络进行通信，网络是不可靠的。当任意数量的消息丢失或延迟到达时，系统仍会继续提供服务，不会挂掉。换句话说，分区容忍性是站在分布式系统的角度，对访问本系统的客户端的再一种承诺：**我会一直运行，不管我的内部出现何种数据同步问题**，强调的是**不挂掉**。

CAP 理论指出，**一个分布式系统只能同时满足其中的两个指标，无法同时满足三个。**

例如，**当出现网络分区时，如果要保证一致性，就必须停止对外服务，从而失去可用性；如果要保证可用性，就必须放弃一致性，从而可能导致不同节点之间数据不一致。**因此，在设计分布式系统时，需要根据**具体的场景和需求**来选择合适的权衡方案，比如选择 CP（一致性和分区容错性）或者选择 AP（可用性和分区容错性）。

需要注意的是，CAP 理论只是一种理论框架，不能直接应用于实际的分布式系统设计。在实际应用中，还需要考虑系统的**具体业务需求、数据访问模式、节点规模和部署环境**等因素，综合权衡之后再选择合适的分布式架构和技术方案。

木木的回答

在一个分布式系统中，分布式的 CAP 理论体现在一下三个基本需求，最多只能同时满足其中的两个：

- **一致性**（C:Consistency）：**所有节点在同一时间具有相同的数据**
- **可用性**（A:Availability）：**每个请求都能得到一个响应，无论成功或失败**
- **分区容错性**（P：Partitiontolerance）：**系统能够容忍网络分区的情况，即节点之间无法通信。**

根据 CAP 理论，一个分布式系统只能在以下三种情况中选择其需要满足的两个需求：

- **CP：满足一致性和分区容错性，但牺牲可用性。**例如：当网络发生故障时，为了保证数据不出现不一致，就拒绝服务请求。
- **AP：满足可用性和分区容错性，但牺牲一致性。**例如：当网络发生故障时，为了保证服务不中断，就允许数据出现不一致。
- **CA：满足一致性和可用性，但牺牲分区容错性。**例如：在没有网络故障的情况下，保证数据始终一致和服务始终可用。

常见的例子

- **CP：Zookeeper、HBase、Redis Cluster等。**这些系统在遇到网络分区的情况下，就回牺牲可用性，来保证数据的一致性。
- **AP：Eureka、Cassandra、MingoDB等**，这些系统在遇到网络分区的情况下，会牺牲一致性，可以摆正服务的可用性。
- **CA：单机数据库、单机缓存等**，这些系统在没有网络分区的情况下，可以保证数据的一致性和服务的可用性。

> CA只是一个理想化的状态，在实际的分布式系统中，网络分区是一个不可避免的问题，所以只有CP和AP是真正有意义的选择。

5、什么是 RPC？目前有哪些常见的 RPC 框架？实现 RPC 框架的核心原理是什么？

官方解析

RPC（Remote Procedure Call）是一种**远程调用协议**，允许一台计算机通过网络调用另一台计算机上的服务或方法。它可以让开发人员像调用本地方法一样调用远程方法，将网络通信细节封装起来，提高了分布式系统中各个模块之间的耦合性。

目前常见的 **RPC 框架**有：

1. Dubbo：阿里巴巴开源的分布式 **RPC 框架**，支持多种协议和负载均衡策略。
2. gRPC：Google 开源的高性能 **RPC 框架**，支持多种语言。
3. Thrift：Facebook 开源的跨语言 **RPC 框架**，支持多种传输协议和数据编解码方式。
4. Spring Cloud Netflix：Spring Cloud 的子项目之一，提供了基于 Netflix OSS 开源组件的微服务解决方案，包括服务发现、负载均衡、熔断器等功能。

RPC 框架的核心原理是基于网络传输协议实现的远程方法调用。**RPC 框架**通常由服务提供者和服务消费者两部分组成，服务提供者将本地方法暴露成远程服务，服务消费者通过远程代理对象调用远程方法。

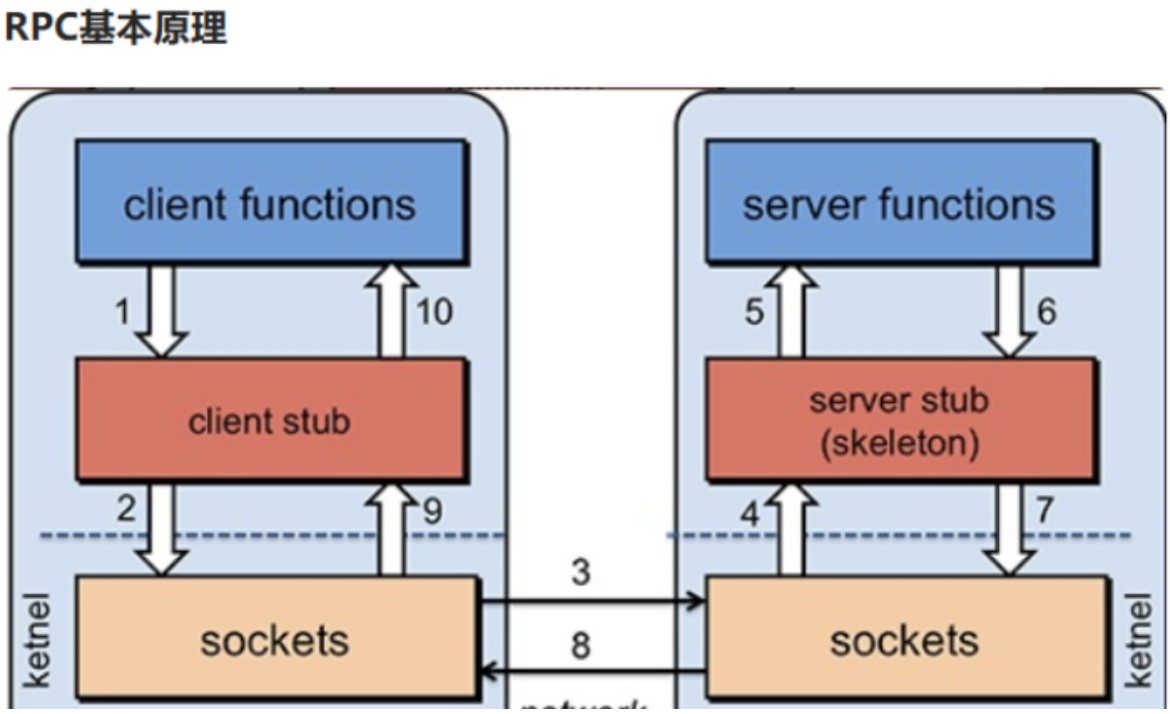
在实现远程方法调用时，需要进行序列化和反序列化操作。序列化将对象转换为二进制数据流，以便于在网络中传输；反序列化则将接收到的二进制数据流转换为对象。

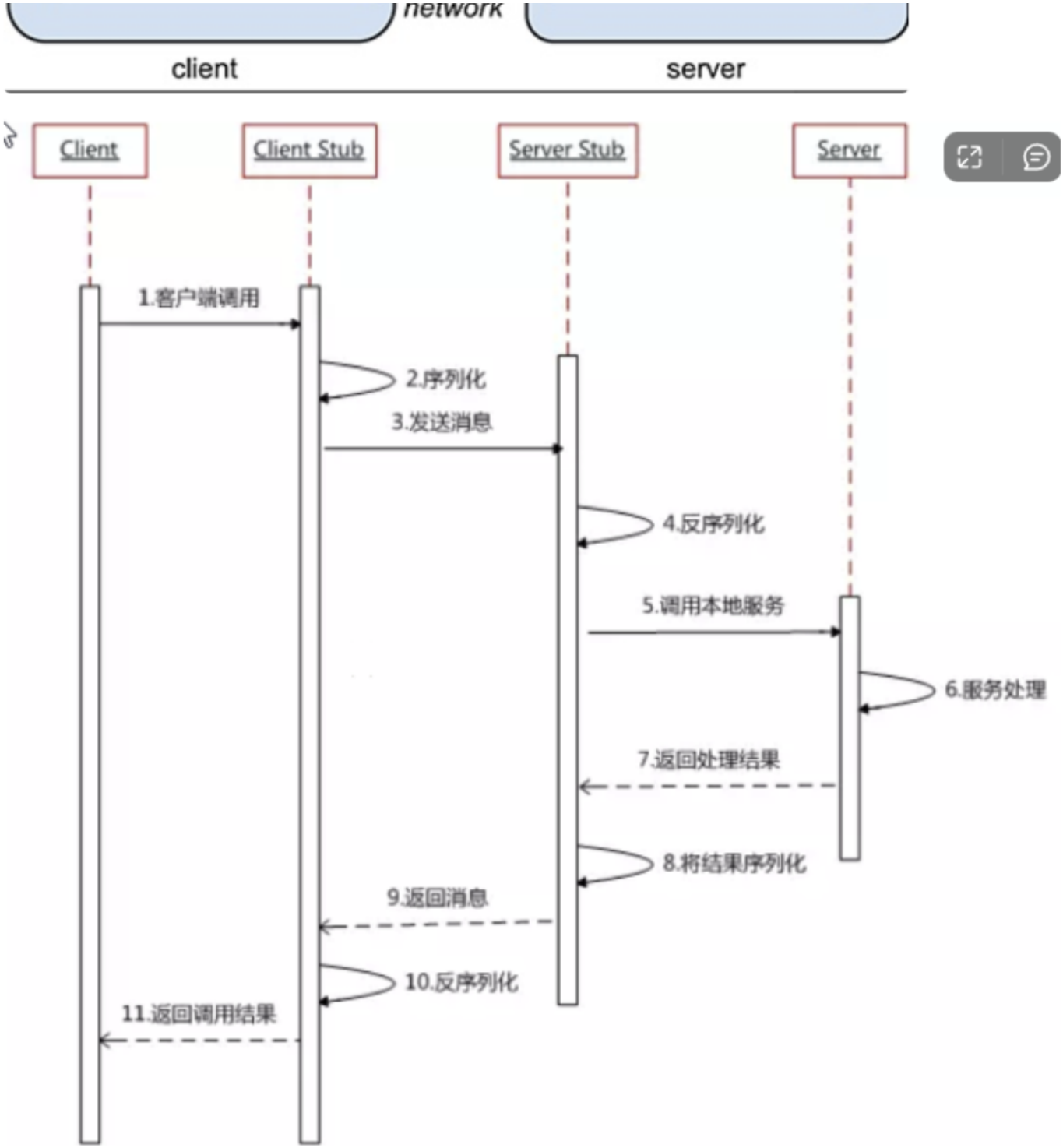
为了提高性能，一些 **RPC 框架**使用了二进制协议，如 Dubbo 使用的 Hessian2 协议和 gRPC 使用的 Protocol Buffers 协议，与基于文本的协议（如 XML 和 JSON）相比，二进制协议具有更小的传输体积和更高的解析速度，能够减少网络传输的开销。

鱼友的精彩回答

林寻的回答

RPC【Remote Procedure Call】是指远程过程调用，是一种进程间通信方式，他是一种技术的思想，而不是规范。它允许程序调用另一个地址空间（通常是共享网络的另一台机器上）的过程或函数，而不用程序员显式编码这个远程调用的细节。即程序员无论是调用本地的还是远程的函数，本质上编写的调用代码基本相同。





木木的回答

什么是 RPC?

RPC 是远程过程调用（Remote Procedure Call）的缩写，是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。

RPC 的主要功能目标是让构建分布式计算/应用更加容易，在提供强大的远程调用能力时不损失本地调用的语义简洁性。

常见的 RPC 框架

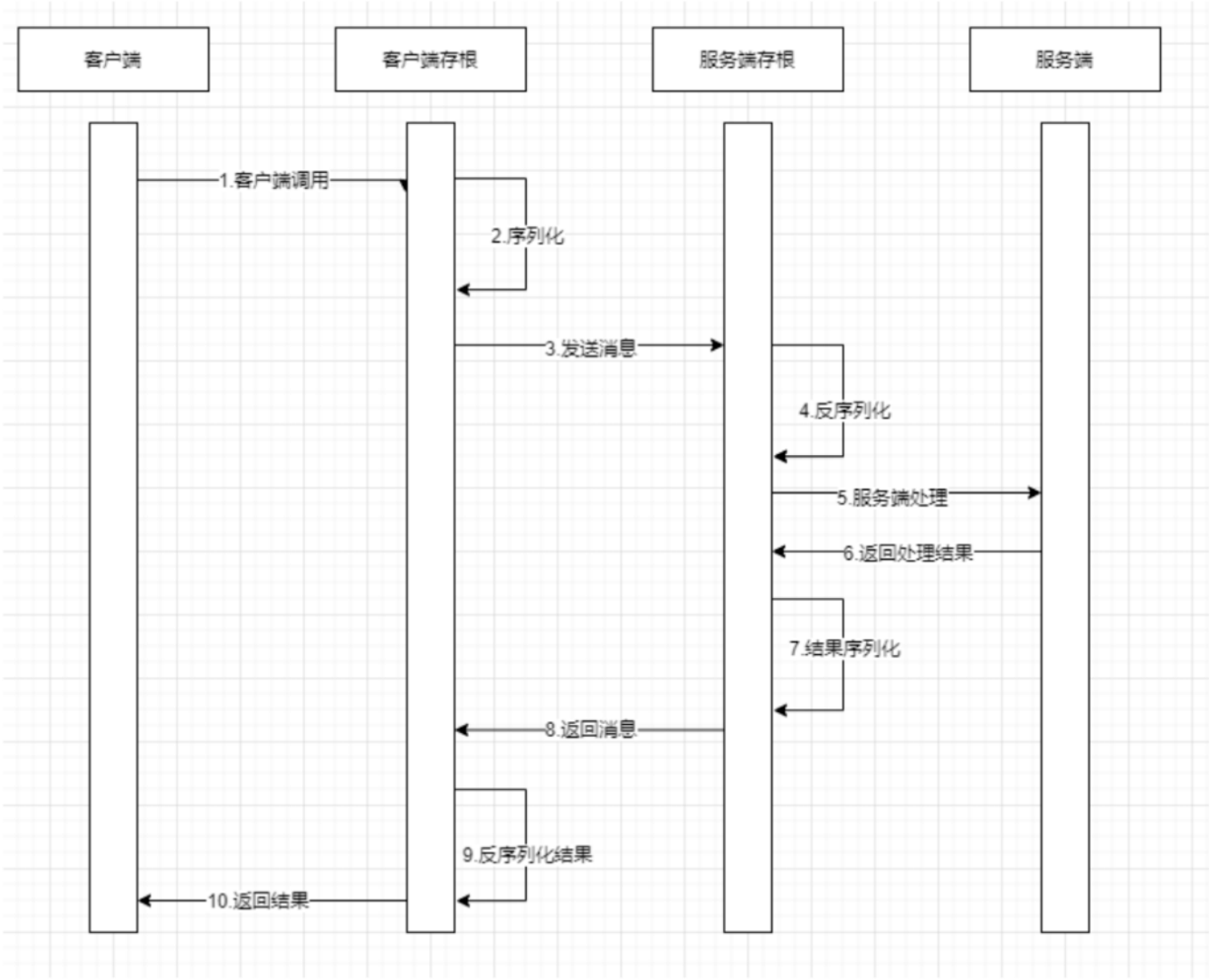
- gRPC：是由 Google 开发的一个基于 HTTP/2和Protocol Buffers 的高性能、跨语言的 RPC 框架，支持多种编程语言，如：java、C++、Python 等。
- Dubbo：是由阿里巴巴开发的一个基于 java 的高性能、轻量级的 RPC 框架，支持多种协议和注册中心，如：zookeeper、Nacos 等等
- Thrift：是由 FaceBook 开发的一个基于二进制协议和多种传输层的 RPC 框架，支持多种编程语言，如：java、c++、Python 等

RPC 框架的核心原理

RPC 框架的核心原理是通过代理、序列化、网络传输、反序列化、反射等技术，实现远程过程调用的透明化。核心流程如下：

1. 客户端通过代理对象（Proxy）调用远程服务，代理对象将调用信息（比如方法名、参数等）进行序列化（Serialzation），转换成字节流；
2. 客户端通过网络传输（Transport）将序列化厚的字节流发送给服务端，服务端收到字节流后进行反序列化（Deserialization），还原成调用信息。
3. 服务端通过反射（Reflection）根据调用信息找到对应的服务信息（Service）和方法（Method），并执行方法，得到返回结果。
4. 服务端将返回结果进行序列化，转换成字节流，通过网络传输发送给客户端，客户端接收到字节流后，进行反序列化，还原成返回结果。
5. 客户端通过代理对象返回结果给调用者，完成远程过程调用。

调用过程流程图如下：



6、什么是注册中心？如何实现一个注册中心？

官方解析

注册中心（Service Registry）是微服务架构中的一个关键组件，负责管理服务实例的信息，包括服务实例的地址、端口、元数据等。它允许服务实例在启动时注册自己，并在关闭时注销。服务消费者可以通过查询注册中心来发现可用的服务提供者，并根据负载均衡策略选择合适的服务实例进行调用。

实现一个注册中心需要以下几个关键功能：

1. **服务注册**：允许服务实例在启动时将自己的信息注册到注册中心，包括地址、端口、元数据等。
2. **服务注销**：允许服务实例在关闭时从注册中心注销自己。
3. **服务发现**：允许服务消费者查询可用的服务实例列表，以便找到合适的服务实例进行调用。
4. **健康检查**：注册中心需要定期检查已注册的服务实例的健康状况，以确保服务实例列表的准确性。如果发现某个服务实例不再可用，注册中心应将其从列表中移除。

实现一个简单的注册中心可以参考以下步骤：

1. 选择一个合适的数据结构（例如哈希表）存储服务实例的信息。
2. 实现一个HTTP API，允许服务实例在启动时向注册中心发送注册请求，将自己的信息添加到数据结构中。

3. 实现一个HTTP API，允许服务实例在关闭时向注册中心发送注销请求，将自己的信息从数据结构中移除。
4. 实现一个HTTP API，允许服务消费者查询可用的服务实例列表。
5. 实现一个定时任务，定期检查已注册的服务实例的健康状况，如果发现某个服务实例不再可用，将其从数据结构中移除。

然而，实现一个可靠、高性能、可扩展的注册中心是一个复杂的任务。在实际应用中，通常推荐使用现有的成熟的注册中心组件，例如 Eureka、Consul、Zookeeper 等。这些注册中心提供了丰富的功能，包括高可用性、数据持久化、动态配置等。

鱼友的精彩回答

小火龙的回答

注册中心是一种**用于服务发现和服务注册**的系统，它用于**管理多个微服务实例的元数据信息**。在一个微服务架构中，可能有数十个、数百个或数千个微服务实例，这些实例可能分布在多台机器上。注册中心通过**维护微服务实例的元数据信息**，如服务的名称、IP地址、端口号、协议等，使得微服务能够相互发现、调用。

一个简单的注册中心可以**使用类似于键值对的方式存储元数据信息**，比如使用Redis、Zookeeper等分布式数据存储系统，通过存储每个服务实例的信息，以及对应的服务名称和版本号，来实现服务发现和注册。通常，注册中心还会提供健康检查、负载均衡、故障转移等功能，以确保服务的高可用性和稳定性。

实现一个注册中心需要考虑以下几个方面：

1. 选择分布式数据存储系统，如 Redis、Zookeeper 等，以存储服务实例的元数据信息。
2. 定义服务注册和发现的接口规范，如 RESTful API 或 RPC 接口。
3. 实现服务注册和发现的逻辑，包括服务注册、服务发现、健康检查、负载均衡、故障转移等功能。
4. 集成到微服务架构中，与微服务框架进行集成，如 Spring Cloud、Dubbo 等，以实现微服务的注册和发现。

一个高可用的注册中心还需要考虑数据的备份和恢复、集群管理、安全认证等方面的问题。

用 java 实现一个简单的的注册中心：

```
import java.util.HashMap;
import java.util.Map;

public class RegistryCenter {
    // 存储服务实例的 map，key 为服务名，value 为该服务的实例列表
    private Map<String, Object> serviceMap = new HashMap<>();

    /**
     * 注册服务实例
     * @param serviceName 服务名
     * @param serviceInstance 服务实例
     */
    public void registerService(String serviceName, Object serviceInstance) {
        if (serviceMap.containsKey(serviceName)) {
            serviceMap.get(serviceName).add(serviceInstance);
        } else {
```

```
            List<Object> serviceInstances = new ArrayList<>();
            serviceInstances.add(serviceInstance);
            serviceMap.put(serviceName, serviceInstances);
        }
    }

    /**
     * 获取服务实例列表
     * @param serviceName 服务名
     * @return 服务实例列表
     */
    public List<Object> getServiceInstances(String serviceName) {
        return serviceMap.get(serviceName);
    }
}
```

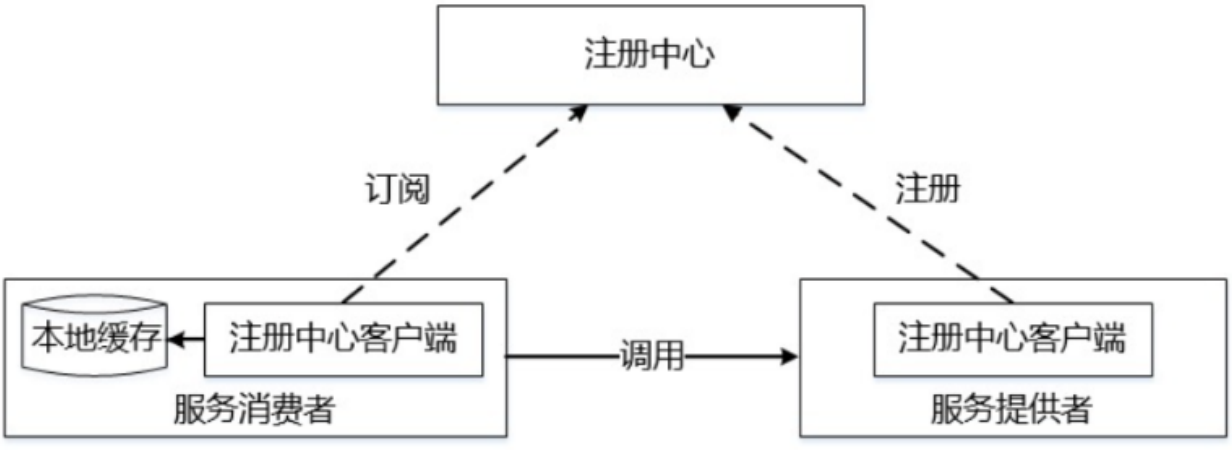
这个注册中心实现了服务实例的注册和查找功能。registerService 方法将一个服务实例注册到注册中心，getServiceInstances 方法可以获取指定服务名的服务实例列表。这个注册中心的实现是一个简单的 HashMap，存储了所有服务实例。实际上，真正的注册中心需要考虑高可用性、数据一致性等问题。

HeiHei 的回答

注册中心：

注册中心是服务实例信息的存储仓库，也是服务提供者和服务消费者进行交互的桥梁。它主要提供了**服务注册和服务发现这两大核心功能**。在一个分布式系统中，不同的服务会以微服务的形式运行在不同的机器上，它们需要相互通信以完成业务逻辑。而注册中心则充当了服务之间的“黄页”，记录了所有可用的服务及其网络地址，方便其他服务进行查找和调用。

当一个新的服务启动时，它会向注册中心注册自己的网络地址和一些元数据信息（例如服务名称、版本号、健康状态等），注册中心会将这些信息存储在自己的数据中心中。当其他服务需要调用这个新服务时，它们可以通过向注册中心查询来获取该服务的地址和元数据信息，然后与该服务建立网络连接。
常见的注册中心包括 ZooKeeper、Consul、Eureka 、Nacos 等等



实现一个注册中心：

1. 设计数据模型：设计注册中心的数据模型，包括服务的元数据信息、服务实例的网络地址等。
2. 实现服务注册：当服务启动时，它需要向注册中心注册自己的元数据信息和网络地址。可以通过 REST API、RPC 等方式实现服务的注册。
3. 实现服务发现：当一个服务需要调用其他服务时，它需要向注册中心查询目标服务的网络地址。可以通过 REST API、RPC 等方式实现服务的发现。
4. 实现健康检查：为了保证服务的可用性，注册中心需要定期检查服务实例的健康状况，并将不健康的实例从服务列表中移除。
5. 实现高可用：注册中心是一个分布式系统的核心组件，需要保证高可用性。可以采用主从复制、集群等方式实现注册中心的高可用性。
6. 实现安全机制：注册中心涉及到服务的元数据信息和网络地址等敏感信息，需要采取合适的安全措

7、什么是分布式的 BASE 理论，它与 CAP 理论有什么联系?

官方解析

BASE 理论是分布式系统中用于描述数据一致性的一个概念。它是一个缩写，分别代表：

1. 基本可用（Basic Availability）：分布式系统在出现故障时，依然能够保证系统的可用性，但可能会出现部分功能或性能降低的情况。
2. 软状态（Soft State）：由于分布式系统中各个节点的状态可能会有一定的延迟，系统允许在一定时间内存在数据不一致的情况。
3. 最终一致性（Eventual Consistency）：在一段时间内，分布式系统中的数据可能不一致，但最终会达到一致状态。这个过程可能需要一定的时间。

CAP理论是另一个关于分布式系统的理论，它指出在分布式系统中，不能同时满足以下三个属性：

1. 一致性（Consistency）：在分布式系统中的所有节点，在同一时刻具有相同的数据。
2. 可用性（Availability）：分布式系统在任何时刻都能对外提供服务，响应用户请求。
3. 分区容错性（Partition Tolerance）：分布式系统在遇到网络分区（部分节点之间通信中断）的情况下仍然能够正常运行。

BASE 理论和 CAP 理论之间的关系是：BASE 理论实际上是对 CAP 理论的一种实践和解释。在 CAP 理论中，由于无法同时满足三个属性，因此在实际的分布式系统设计中，通常需要在一致性和可用性之间做出权衡。BASE 理论就是这种权衡的一种体现，它强调基本的可用性、软状态和最终一致性，而不是追求强一致性。在很多场景中，采用 BASE 理论能够带来更高的系统可用性和更好的性能表现。

鱼友的精彩回答

HeiHei 的回答

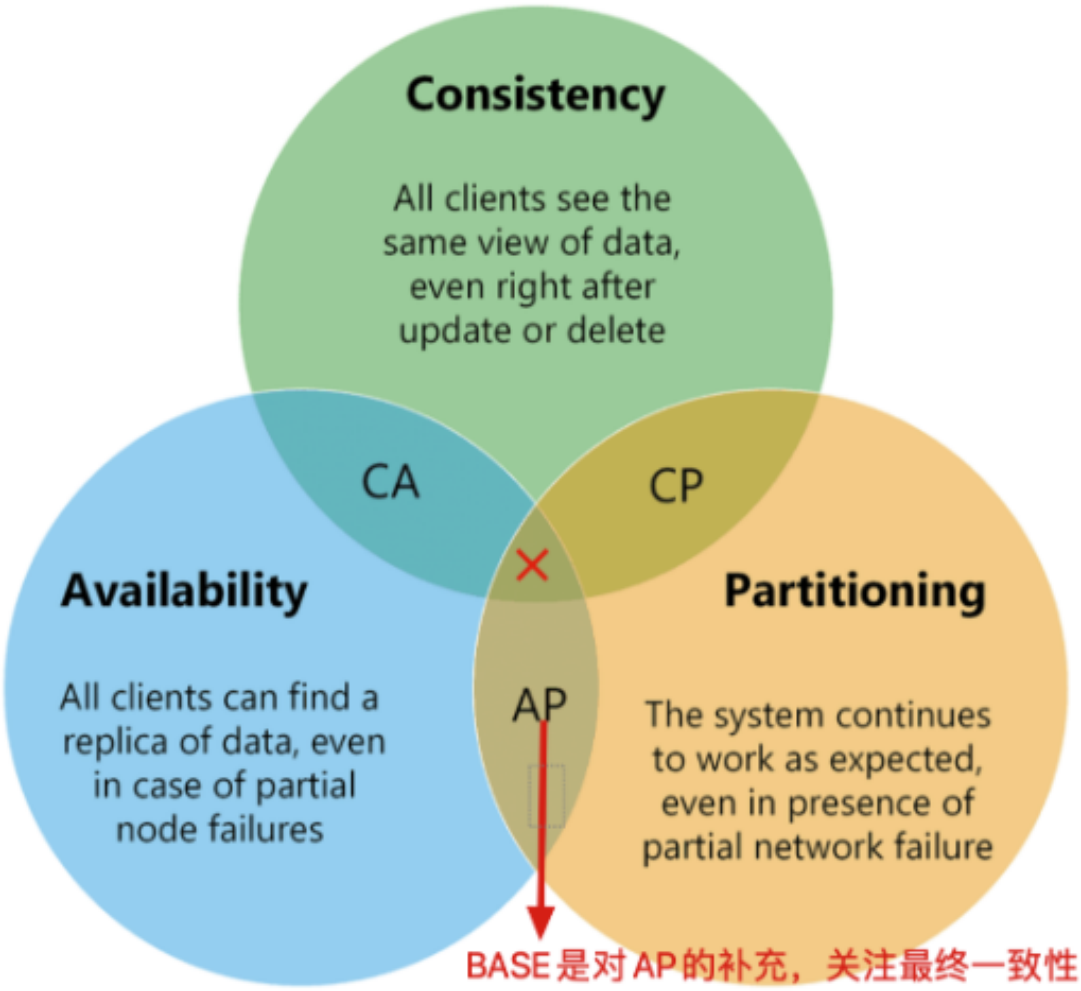
BASE 理论：

BASE 理论是对 CAP 理论的延伸，核心思想是即使无法做到强一致性（Strong Consistency，CAP 的一致性就是强一致性），但应用可以采用适合的方式达到最终一致性（Eventual Consitency）。

1. Basically Available（基本可用）：分布式同再出现不可预知故障的时候，允许损失部分可用性
2. Soft state（软状态）：软状态也称弱状态，和硬状态相对，是指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时
3. Eventually consistent（最终一致性）：最终一致性强调的是系统中所有的数据副本，在经过一段时间上的同步后，最终能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性

CAP 与 BASE 关系

BASE 是对 CAP 中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的结论，是基于 CAP 定理逐步演化而来的，其核心思想是即使无法做到强一致性（Strong consistency），更具体地说，是对 CAP 中 AP 方案的一个补充。其基本思路就是：通过业务，牺牲强一致性而获得可用性，并允许数据在一段时间内是不一致的，但是最终达到一致性状态。



消息队列

1、什么是消息队列？消息队列有哪些应用场景？

官方解析

消息队列是一种异步通信机制，用于在应用程序之间传递消息。它可以将消息暂时存储在队列中，然后按照一定的顺序和条件将消息传递给消费者。

消息队列有以下几个主要的应用场景：

- 异步处理：通过将任务转换为消息，异步地进行处理，可以提高系统的吞吐量和响应速度。
- 系统解耦：在不同的系统或模块之间使用消息队列进行通信，可以实现系统的解耦，提高系统的灵活性和可扩展性。
- 流量控制：消息队列可以对消息进行缓存和限流，保证系统的稳定性和高可用性。
- 应用解耦：在同一个应用程序中，不同的模块之间使用消息队列进行通信，可以实现模块之间的解耦，提高代码的可维护性。
- 日志处理：通过将日志转换为消息，可以实现日志的异步处理，提高系统的性能和可维护性。

常见的消息队列包括 Kafka、RabbitMQ、ActiveMQ、RocketMQ 等。

> 鱼皮补充：一般问到 “应用场景” 的题目，面试官其实更希望你结合自己的项目去说

鱼友的精彩回答

Gundam 的回答

消息队列是一种用于异步通信的机制，用于在不同的应用程序之间传递消息。消息队列通常由**消息生产者、消息队列和消息消费者**三部分组成。

消息生产者将消息发送到消息队列中，而消息消费者则从消息队列中接收消息。消息队列负责存储和管理消息，确保消息传递的可靠性和稳定性。在实现过程中，消息队列还会提供一些额外的功能，如消息过滤、消息路由、消息持久化等。

消息队列的特点：

1. **异步通信**：消息生产者和消息消费者之间采用异步通信模式，发送方无需等待接收方的响应即可继续执行。
2. **解耦合**：消息队列可以将消息生产者和消息消费者解耦合，使得它们之间的关系更加灵活。
3. **可靠性**：消息队列通常会提供一些保证消息传递可靠性的机制，如消息持久化、重试机制等。
4. **缓冲**：消息队列可以缓冲来自多个消息生产者的消息，使得消息消费者可以按照自己的节奏进行消费，从而有效地平衡生产者和消费者之间的处理速度。

消息队列的应用：

1. **异步任务处理**：通过将任务发送到消息队列中，异步处理任务，提高系统的并发性能和吞吐量。
2. **解耦合系统**：将不同的业务逻辑拆分成不同的服务，通过消息队列实现服务之间的通信，提高系统的可维护性和可扩展性。
3. **流量削峰**：将流量通过消息队列分散到不同的服务中，避免单个服务被高并发流量打垮。
4. **日志收集**：通过将日志消息发送到消息队列中，将日志收集和分析与业务逻辑解耦合，提高系统的可靠性和可维护性。

5. **应用解耦**：将不同的应用程序通过消息队列进行集成，实现应用之间的解耦合和数据交换。

行云的回答

消息队列基本架构：

- 有称为生产者的客户端应用程序可以创建消息并将其传递到消息队列。
- 另一个称为消费者的应用程序连接到队列并获取要处理的消息。
- 存储在队列中的消息将被存储，直到消费者检索到它们为止。

消息队列的特点：

- 提供异步通信协议,该协议是一种将消息放入消息队列并且不需要立即响应来继续处理的系统。
- 电子邮件可能是异步通信的最佳示例。发送电子邮件后，发件人将继续处理其他事情，而无需接收者的立即响应。
- 这种处理消息的方式使生产者与使用者脱钩，从而使他们不需要同时与消息队列进行交互。

应用场景：

- 应用解耦合：
  - 场景：A 系统产生一条比较关键的数据，很多系统都需要 A 系统将这个数据发送过来。A 系统要时时刻刻考虑 BCDE 四个系统如果挂了该咋办？要不要重发，要不要把消息存起来?
  - 解决：如果使用 消息队列（MQ），A 系统产生一条数据，发送到 MQ 里面去，哪个系统需要数据自己去 MQ 里面消费。如果新系统需要数据，直接从 MQ 里消费即可；如果某个系统不需要这条数据了，就取消对 MQ 消息的消费即可。这样下来，A 系统压根儿不需要去考虑要给谁发送数据，不需要维护这个代码，也不需要考虑人家是否调用成功、失败超时等情况。
- 异步：
  - 场景：以系统注册用户为例，正常情况下，注册用户信息需要查询数据库需要50ms，然后向用户发送邮件需要50ms,然后再向用户发送短信通知获取验证码又需要50ms,共计需要150ms，我们可以发现这些都是同步进行的，比较浪费时间。
  - 解决：如果采用 mq 之后，注册用户信息需要查询数据库需要50ms，然后其他操作写入到消息队列中，消息队列异步执行邮件以及短信服务，可能只需要20ms,这时共计需要 70ms 则可以完成任务，大大的缩短了任务的执行时间。
- 流量削峰：
  - 场景：如果一个订单系统，每秒钟可以处理一万次下单，在平时正常的时候该系统可以应付正常的请求，正常时段我们下单之后就会收到结果。但是如果突然遇到高峰期，有十万次下单操作给到订单系统中，这时由于超出处理范围则会造成系统崩溃。
  - 解决：可以用到消息队列来做缓冲，将其流量削峰，将一秒内十万次下单存到消息队列中，然后订单系统每秒只拉取一万下单来做处理，将其他订单先积压在消息队列中，等高峰期过去后，系统就会慢慢把订单都处理完。

L Y J 的回答

消息队列：是分布式系统中重要的组件，主要解决应用耦合，异步消息，流量削锋等问题。

实现高性能、高可用、可伸缩和最终一致性架构。

使用较多的消息队列有 ActiveMQ、RabbitMQ、ZeroMQ、Kafka、MetaMQ、RocketMQ。

1、异步处理：

可以将对实时性要求不高的场景进行异步处理，如用户注册后发送注册短信和注册邮件。

传统串行方式，将注册信息写入数据库成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，再返回给客户。

并行方式，将注册信息写入数据库成功后，发送注册邮件和发送注册短信同时进行，三个任务完成后，返回给客户端。可以提高处理时间。

假如三个业务节点每个使用 50 毫秒，不考虑网络等其他开销，则串行方式的时间为 150 毫秒，并行的时间可能是 100 毫秒。

2、应用解耦：

用户下单后，订单系统需要通知库存系统。传统方法是：订单系统调用库存系统的接口。

假如库存系统没有办法完成访问，则整个操作无法完成。但是如果将消息写入消息队列，用户下单后，订单系统持续化处理，将返回用户订单下单成功，库存系统，将采用拉/推的方式，获取下单信息，库存系统根据下单的信息，进行库存操作。如此设计后，订单系统与库存系统将分离开来，订单系统写入消息队列，之后就不再关心其他后续操作，实现了解耦。

3、流量削峰

在秒杀系统中使用广泛，一般因为流量过大，导致流量暴增，然后应用挂掉。解决方案为：在前段加入消息队列，可以控制活动的人数，缓解短时间内高流量压垮应用。

用户的请求，服务器接收后，首先写入消息队列，加入消息队列过长或者长度超过最大数量，则直接抛弃用户请求或者跳转到错误页面。

4、 **日志处理**，将消息队列使用在日志处理中，比如 Kafka 的应用，解决大量日志传输的问题，日志采集客户端，负责日志数据采集，定时写入 Kafka 队列；Kafka 消息队列，负责日志数据的接收、存储、转发；日志处理应用：订阅并消费 Kafka 队列中的日志数据。

5、 **消息通讯**，消息队列一般内置了高效地通信机制，可以实现高效的点对点的消息队列，或者聊天室等。

点对点通讯：客户端 A 和客户端 B 使用同一队列，进行消息通讯。

聊天室通讯：客户端A和客户端B...客户端N，订阅同一个主题，进行消息发布和接收，实现聊天室的效果。

2、有哪些主流的消息队列，它们分别有什么优缺点、各自的适用场景是什么?

官方解析

以下是几种主流的消息队列：

1. **RabbitMQ 优点**：可靠性高、性能优秀、支持多种协议、有完善的管理界面。 缺点：部署和维护较为复杂。 适用场景：适用于高可靠性、高吞吐量、多协议、多语言的分布式系统场景。
2. **Kafka 优点**：性能优秀、可扩展性好、可靠性高、支持多种数据处理模式。 缺点：管理界面不够完善、复杂度较高。 适用场景：适用于高吞吐量、高并发、数据处理流程复杂的场景，例如大数据处理、实时日志处理等。
3. **ActiveMQ 优点**：功能齐全、易于使用、支持多种协议。 缺点：性能相对较低、可靠性不如 RabbitMQ。 适用场景：适用于需要使用多种协议、支持多种消息类型的场景，例如 Web 服务、企业应用集成等。
4. **RocketMQ 优点**：性能优秀、可靠性高、支持海量数据存储和传输。 缺点：社区相对较小、功能不够完善。 适用场景：适用于海量数据存储和传输场景，例如电商、金融等领域。
5. **Redis 优点**：速度极快、支持多种数据结构、支持事务操作、支持发布/订阅模式。 缺点：可靠性不如 RabbitMQ 和 Kafka。 适用场景：适用于对性能要求极高、需要使用多种数据结构和事务操作的场景，例如缓存、计数器、实时消息等。

总的来说，选择适合自己的消息队列需要根据具体业务需求和场景进行综合评估和选择。

鱼友的精彩回答

苏打饼干的回答

名称	优点	缺点	单机吞吐量	分布式	应用
ActiveMQ	时效性ms级，消息可靠性高	官方维护少、性能相对低	万级		Web服务器、企业应用集成 需要多种协议、多种消息类型的场景
Kafka	不易丢失数据 支持多种数据处理模式	Load明显飙高，可能出现消息乱序 社区活跃度低	百万级	√	大数据、日志采集 高吞吐量、高并发、数据处理流程复杂的场景
RocketMQ	**消息零丢失** 支持**十亿级别消息堆积**	支持的客户端语言不多	十万级	√	金融互联网、秒杀 海量数据和传输场景
RabbitMQ	**高并发，支持多种语言** 社区活跃度高	商业版收费	万级	√	数据量中小型 高可靠、多协议、多语言的分布式系统场景
Redis	速度快、支持数据结构丰富、支持事务	可靠性不如RabbitMQ和Kafka		√	缓存、计数器、实时信息 对性能要求极高的场景

猫十二懿的回答

主流的消息队列如下

消息队列	优点	缺点	适用场景
RabbitMQ	开源、可靠性高、支持多种协议和编程语言、性能稳定	需要安装 Erlang 环境、性能较弱	适用于吞吐量不高，但对可靠性和多语言支持有要求的场景
Apache Kafka	高吞吐量、分布式、可水平扩展、支持流处理	适用于大规模数据处理，但需要额外的复杂配置和运维成本	
ActiveMQ	开源、性能稳定、支持多种协议和编程语言、集成广泛	稳定性不如 RabbitMQ、复杂性较高	适用于需要多种集成方式和协议的场景
RocketMQ	分布式、高可用、性能高、支持批量发送和顺序消费	对于消息存储和网络传输对可靠性要求高	适用于高吞吐量和可靠性要求高的场景
Amazon SQS	可靠性高、支持多种协议和编程语言、弹性可扩展	有一定的限制，如不能支持 push 模式	适用于需要可靠性和弹性可扩展性的场景
Google Cloud Pub/Sub	分布式、高可靠性、支持多种协议和编程语言、可以与 Google Cloud 平台集成	可能会受到 Google Cloud 平台的限制	适用于需要与 Google Cloud 平台集成的场景
Redis消息队列	高性能，单机可以处理百万级别的消息。 可以支持多种消费者，从而实现并行消费。 可以支持消息持久化，保证消息不丢失。 支持事务和 Lua 脚本，可以实现复杂的消息处理逻辑	不支持消息的顺序消费。 不支持消息的重复消费。 单机容量受限，无法支持大规模数据处理	秒杀、抢购等场景

飞扬的回答

消息队列是在消息的传输过程中保存消息的容器，简单点理解就是传递消息的队列，具备先进先出的特点，一般用于异步、解耦、流量削锋等问题，实现高性能、高可用、高扩展的架构。

常见的消息队列使用场景有 6 个：**应用解耦,异步处理,流量削锋,日志处理,消息通讯,消息广播.**

主流的消息队列有 ActiveMQ，RabbitMQ、RocketMQ、Kafka。

特性	Kafka	RocketMQ	RabbitMQ	ActiveMQ
单机吞吐量	10万级别，吞吐量高是kafka最大的优点	10万级，RocketMQ 也是可以支撑高吞吐的 MQ	万级，吞吐量比RocketMQ和Kafka要低了一个数量级	万级，吞吐量比RocketMQ和Kafka要低了一个数量级
支持主题数	百级，topic 达到百级时吞吐量会大幅度下降，要尽量保证 topic 数量不要过多，否则需要增加更多机器资源	千级，topic 达到千级时吞吐量会有较小幅度的下降。可以支撑大量 topic 是 RocketMQ 的一大优点	百万级	千级
消息顺序性	分区有序	有序	有序	有序
消息重复	至少一次，最多一次	至少一次，最多一次	至少一次	至少一次
时效性	ms级	ms级	微秒级，RabbitMQ的一大优点	ms级
可用性	非常高，分布式架构，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用	非常高，分布式架构	高，基于主从架构实现高可用性	高，基于主从架构实现高可用性
消息可靠性	经过参数优化配置，理论上消息可以做到0丢失	经过参数优化配置，理论上消息可以做到0丢失	有较低的概率丢失数据	有较低的概率丢失数据
消息回溯	支持（按offset回溯）	支持（按时间回溯）	不支持	不支持
功能支持	功能较为简单，主要支持简单的MQ功能，在大数据领域的实时计算以及日志采集被大规模使用，是事实上的标准	MQ功能较为完善，还是分布式的，扩展性好	基于erlang开发，所以并发能力很强，性能极其好，延时很低	MQ领域的功能极其完备
伸缩性	高伸缩性，每个主题（topic）包含多个分区（partition），主题中的分区可以分布在不同的主机（broker）中	高伸缩性，灵活的分布式横向扩展部署架构，整体架构和 kafka 很像	一般	
管理界面	普通	完善	普通	普通
持久化	消息可以持久化到磁盘	消息可以持久化到磁盘	持久化不好，可以持久化到内存、文件	可以持久化到内存、文件、数据库
消息路由	不支持	不支持	支持	
语言支持	支持多语言，Java优先	支持Java、C++，但C++不成熟	支持几乎所有最受欢迎的编程语言：Java，C，C ++，C #，Ruby，Perl，Python，PHP等	支持多语言，Java优先
社区活跃度	高	一般	高	高

3、有哪些常见的消息队列模型？分别适用于什么场景？

官方解析

消息队列是一种在分布式系统中用于**异步通信**的模型，它允许不同的组件通过将消息发送到队列来实现解耦和灵活性。以下是常见的消息队列模型及其适用场景：

1. **点对点模型**（Point-to-Point Model）：这种模型中，**消息生产者**将消息发送到一个**队列**中，**消息消费者**从该队列中接收消息。**一个消息只会被一个消费者接收**，消费者在处理完消息之后会从队列中删除它。这种模型适用于**需要保证消息只被一个消费者处理的场景**，例如**订单系统**、**日志处理**等。

2. **发布-订阅模型**（Publish-Subscribe Model）：这种模型中，**消息生产者**将消息发送到一个**主题**（Topic）中，多**个消息消费者**可以**订阅**该主题并接收到所有的消息。**每个消息可以被多个消费者接收**，消费者在处理完消息之后不会从主题中删除它。这种模型适用于**需要将消息广播给多个消费者的场景**，例如**新闻订阅、实时数据分析**等。
3. **请求-应答模型**（Request-Response Model）：这种模型中，**消息生产者**发送一个请求消息到一个队列中，**消息消费者**从该队列中接收请求并返回一个响应消息。**一个请求只会被一个消费者接收并处理**，处理完成后返回**响应消息**给消息生产者。这种模型适用于需要**请求-响应**模式的场景，例如**远程过程调用、微服务通信**等。
4. **推拉模型**（Push-Pull Model）：这种模型中，**消息生产者**将消息**推送**到一个**队列**中，**消息消费者**从该队列中**拉取消息**。消息生产者将消息发送到队列中，消费者按需从队列中拉取消息进行处理。这种模型适用于**需要灵活控制消息消费速度的场景**，例如**数据采集、视频流传输**等。
   需要注意的是，以上模型并不是完全独立的，实际使用中可以根据具体场景**组合使用**不同的模型。例如，可以将**点对点模型和请求-应答模型**结合使用，实现**异步的 RPC 调用**。另外，在选择消息队列模型时，还需要考虑消息传输的**可靠性、顺序性、延迟**等因素。

鱼友的精彩回答

小幸运的回答

一、什么是消息队列

我们可以把消息队列比作是一个存放消息的容器，当我们需要使用消息的时候可以取出消息供自己使用。**消息队列是分布式系统中重要的组件，使用消息队列主要是为了通过异步处理提高系统性能和削峰、降低系统耦合性。**

二、常见模型

ActiveMQ，RabbitMQ，Kafka，RocketMQ

1. 点对点模型（Point-to-Point Model）：也被称为队列模型，消息生产者将消息发送到队列中，然后消息消费者从队列中获取消息并处理。适用于需要精确传递消息的场景，如订单处理、任务调度等。
2. 发布/订阅模型（Publish/Subscribe Model）：也被称为主题模型，消息生产者将消息发送到主题中，然后主题将消息广播给所有订阅该主题的消费者。适用于需要将消息广播给多个消费者的场景，如新闻订阅、实时数据更新等。
3. 管道模型（Pipes and Filters Model）：消息生产者将消息发送到管道中，然后管道中的过滤器依次处理消息并将其发送给下一个过滤器。适用于需要将消息按照一定的处理流程进行处理的场景，如日志处理、数据清洗等。

三、具体应用场景

1. RabbitMQ：高吞吐量的消息队列；多种语言客户端库支持；支持多种消息协议；支持复杂的路由规则；支持消息确认机制；适合任务队列、日志处理、消息通信等场景。
2. Apache Kafka：高吞吐量、支持百万级别的消息每秒处理能力；分布式、高可靠、可扩展；支持持久化存储消息；数据复制至多个副本，保证数据可靠性；适合大规模数据流处理、日志系统等场景。
3. Apache ActiveMQ：完全支持 JMS 规范，具有良好的跨语言支持；支持多种消息协议和多种持久化方式；具有较强的安全机制和集群管理能力；适合企业级应用、金融行业等场景。
4. Apache RocketMQ：高吞吐量，低延迟，亿级消息堆积能力；支持事务消息、定时消息等高级特性；支持数据双写，保证数据可靠性；适合大规模数据流处理、金融支付等高可靠性场景。

> 作者：编程导航知识星球 + 星球鱼友们

设计模式

1、设计模式是什么？为什么要学习和使用设计模式?

官方解析

设计模式是一套被反复使用、经过验证的、通用的解决特定问题的设计思想，是一种被设计师反复使用，经过时间验证的、解决特定问题的一种技术方案。

设计模式的主要作用在于：

- 提高代码的可维护性、可扩展性、可读性，提高代码的质量；
- 通过共享经验，提高开发人员的设计能力，缩短学习时间，增强团队合作效率；
- 提高开发效率，缩短开发周期。

设计模式主要分为三大类：

- **创建型模式**：用于描述创建对象的方式；
- **结构型模式**：用于描述如何组合对象，形成更大的结构；
- **行为型模式**：用于描述对象之间的协作和职责分配。

在具体应用设计模式时，需要根据实际场景选用合适的设计模式。常用的设计模式包括**单例模式、工厂模式、观察者模式、适配器模式、装饰器模式、策略模式**等。

> 鱼皮补充：这题不用像报菜名一样说出所有设计模式，最好能举例你是如何在项目中应用它的

鱼友的精彩回答

一切总会归于平淡的回答

设计模式是指在软件开发中经常遇到的一些重复性问题，通过对这些问题的总结、抽象、归纳和提炼，得到的一些解决问题的通用方案。

学习和使用设计模式可以帮助开发人员提高代码的**可重用性、可维护性、可扩展性和可读性**，从而提高开发效率和代码质量。

设计模式的学习和使用，需要结合实际的业务场景进行理解和应用。以下是一些常见的现实业务场景和对应的设计模式：

- 工厂模式：当需要创建多种具有相同特征的对象时，使用工厂模式可以将对象的创建与业务逻辑分离，降低代码的耦合度。 例如，在电商平台上，不同种类的商品都需要进行库存管理和订单管理，可以使用工厂模式来创建对应的库存管理器和订单管理器。

- 单例模式：当需要确保系统中某个类只有一个实例时，可以使用单例模式，保证全局唯一性，避免资源的浪费。 例如，在 Web 应用中，有时需要保证所有请求都使用同一个数据库连接，可以使用单例模式来实现数据库连接池。
- 观察者模式：当一个对象的状态发生改变需要通知其他对象时，可以使用观察者模式，将对象的状态与业务逻辑分离，提高系统的灵活性和可扩展性。 例如，在多人在线游戏中，玩家的行为会影响其他玩家的状态，可以使用观察者模式来实现游戏中的事件处理和状态同步。
- 策略模式：当需要在运行时根据不同的情况采用不同的算法时，可以使用策略模式，将算法与业务逻辑分离，提高代码的可维护性和扩展性。 例如，在电商平台上，可以使用策略模式来实现不同的促销策略，例如满减、打折等。

面试官问到这个问题，可能想了解面试者是否熟悉常见的设计模式，并能够**结合实际业务场景进行理解和应用**，以提高代码质量和开发效率。同时，也想了解面试者是否有足够的设计能力和经验，能够在实际项目中使用设计模式来解决问题。

行云的回答

设计模式：是一套经过反复使用的代码设计经验，目的是为了重用代码、让代码更容易被他人理解、保证代码可靠性。

项目中合理地运用设计模式可以完美解决很多问题，每种模式在现实中都有相应的原理来与之对应，每种模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。

设计模式分为三大类：

- **创建型模式**：共5种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式
- **结构型模式**：共7种：适配器模式、装饰器模式、代理模式、桥接模式、外观模式、组合模式、享元模式
- **行为型模式**：共11种：策略模式、模板方法模式、观察者模式、责任链模式、访问者模式、中介者模式、迭代器模式、命令模式、状态模式、备忘录模式、解释器模式

2、什么是单例模式？使用单例模式有什么好处？有哪些常用的单例模式实现方式？各自的应用场景是什么？

官方解析

单例模式是一种**创建型设计模式**，它确保**一个类只有一个实例**，并提供**一个全局访问点**来访问该实例。**单例模式**的目的是**确保类的一个唯一实例**，因此其他类可以轻松地从一个可知的地方访问它。

单例模式有以下好处：

- **节省系统资源**：在系统中，如果有多个实例会造成资源浪费，而使用单例模式可以减少这种浪费。
- **简化了对象访问**：单例模式提供了一个**全局的访问点**，因此可以简化访问过程。

常用的单例模式实现方式有以下几种：

- **饿汉式单例模式**：在**类加载**时创建单例对象。缺点是**不支持延迟加载**。
- **懒汉式单例模式**：在**第一次使用时**才创建单例对象。缺点是**需要考虑线程安全**问题。

- **双重检查锁单例模式**：在**第一次使用**时创建单例对象，并使用**双重检查锁定**来确保线程安全。
  枚举单例模式：在**枚举类型**中创建单例对象，可以**防止反射和序列化攻击**。

应用场景：

- **数据库连接池**：通过**单例模式**，可以确保系统中只有一个数据库连接池。
- **日志记录器**：可以使用单例模式记录系统日志，这样可以确保系统中只有一个日志记录器。
- **配置文件管理器**：可以使用**单例模式**来**管理应用程序的配置文件**，这样可以避免重复读取配置文件的开销。
- **线程池**：可以使用**单例模式**来确保系统中只有一个线程池。

一个例子是 **Spring** 框架中的 **ApplicationContext**，它是一个全局访问点，提供了一个管理 Bean 的中央注册表。由于 Spring 中的 Bean 只需创建一次，因此 ApplicationContext 使用单例模式确保只有一个实例。

> 鱼皮补充：有些面试官可能会让你手写一种单例模式的实现，建议大家在学习设计模式时，多写代码实践下

鱼友的精彩回答

爱吃鱼蛋的回答

介绍

单例模式是一种设计模式，它的目的是**保证一个类只有一个实例**，并提供一个**全局的访问点**。使用单例模式可以**避免多次创建对象**，**节省内存空间**，同时也可以**保证数据的一致性**。在开发过程中使用单例模式有以下好处：

1. **节省内存空间**。单例模式只创建一个实例，避免了多次创建对象所造成的内存消耗。
2. **简化代码**。单例模式提供了一个**全局的访问点**，可以方便地调用实例的方法，避免了重复的代码。
3. **保证数据的一致性**。由于只有一个实例，可以**避免并发访问时数据不一致**的问题。

实现方式

实现单例模式的方式有很多，下面是一些常见实现**单例模式**的方式：

- **饿汉式单例模式**：在**类加载时就创建了实例**，因此保证了**线程安全**。但是**可能会造成资源浪费**，因为**实例在使用前就已经创建**了。
- **懒汉式单例模式**：实例的创建是在**第一次使用**时才进行的。这种方式需要**注意线程安全**，可以使用 **synchronized 关键字**或者**双重检查锁定**等方式来保证线程安全。
- **静态内部类单例模式**：这种方式可以**保证线程安全**，且实现简单。实例的创建是在**第一次使用**时才进行的。
- **枚举单例模式**：这种方式可以**保证线程安全**，且实现简单。**枚举类型的属性在 Java 中只会被初始化一次**，因此**枚举单例模式可以保证只有一个实例**。
- **双重检查锁定单例模式**：这种方式可以保证线程安全，且实现相对简单。在第一次调用 getInstance()方法时进行了**双重检查**，保证了只有一个实例被创建。
- **线程局部变量单例模式**：这种方式可以**保证线程安全**，且实现简单。在**单个线程内部只会创建一个实例**，因此不会造成资源浪费。

这里提供一种使用了**双重检查锁**的**线程安全**方式实现单例模式

```
public class Singleton {

```

```
    // 私有构造方法
    private Singleton() {}
    // 使用volatile禁止单例对象创建时的重排序
    private static volatile Singleton instance;

   // 对外提供静态方法获取该对象
    public static Singleton getInstance() {
    // 第一次判断，如果instance不为null，不进入抢锁阶段，直接返回实际
        if(instance == null) {
            synchronized (Singleton.class) {
                // 抢到锁之后再次判断是否为空
                if(instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

区别

这些不同的实现方式各有优缺点，在项目开发过程中需要根据**场景需求**选择合适的实现方式

实现方式	线程安全性	实现复杂度	资源浪费
饿汉式单例模式	线程安全	简单	可能会造成资源浪费
懒汉式单例模式	需要注意线程安全	较复杂	可能会造成资源浪费(线程不安全)
静态内部类单例模式	线程安全	简单	不会造成资源浪费
枚举单例模式	线程安全	简单	不会造成资源浪费
双重检查锁定单例模式	线程安全	较复杂	不会造成资源浪费
线程局部变量单例模式	线程安全	简单	不会造成资源浪费

使用场景

单例模式在开发中有许多适用的场景，如：

1. **数据库连接池**：在多线程环境下，为了避免频繁地创建和释放数据库连接，可以使用单例模式来实现数据库连接池，**保证系统中只有一个连接池实例**；
2. **应用程序配置信息对象**：在系统中需要读取应用程序配置信息时，为了避免重复读取和存储配置信息，可以使用单例模式来实现配置信息对象，**保证系统中只有一个配置信息对象实例**；

3. **线程池**：在多线程环境下，为了避免频繁地创建和销毁线程，可以使用单例模式来实现线程池，**保证系统中只有一个线程池实例**。

在我们日常开发中接触到的 Java 和 Spring 也有许多地方应用了单例模式：

1. 在 Java 中，**Runtime** 类就是一个单例模式。这个类提供了**访问 Java 虚拟机的运行时环境的方法**，比如获取内存信息、获取 CPU 核心数等。由于每个 Java 应用程序只有一个 Java 虚拟机实例，因此Runtime类只需要一个实例即可，这就是单例模式的应用。
2. 在 Spring 框架中，ApplicationContext 对象也是一个单例模式。ApplicationContext 对象是 Spring 框架中的**容器对象**，负责管理 Bean 对象的创建、初始化、依赖注入等操作。由于 ApplicationContext 对象的创建和初始化需要消耗大量的系统资源，因此在整个应用程序中只需要一个 ApplicationContext 对象实例即可，这就是单例模式的应用。

注意事项

单例模式虽然好，但是任何设计模式都会存在其弊端，因此在使用过程中需要注意以下问题：

1. **线程安全问题**。这个问题在饿汉式中是一个常见的问题，普通的**饿汉式**方案在多线程情况下容易出现并发问题导致创建多个不同的示例，因此在实现方案中可以使用**加锁**的方式保证线程的安全；
2. **序列化和反序列化问题**。在单例模式中存在着破坏单例的风险，而**序列化**和**反序列化**便是其中之一。当我们使用序列化和反序列化创建单例对象时会发现创建出来的对象都是不一样的，违背了单例模式的初衷，因此在需要序列化和反序列化类中实现实现 Serializable 接口，并且提供一个 readResolve()方法，以保证反序列化后仍然是同一个实例；
3. **反射问题**。使用单例模式时，需要考虑防止**反射攻击**。因为破坏单例的另外一种方式便是**反射**，就算是前面提及到的序列化和反序列化问题，**底层也是通过反射来破坏单例的**。因此在开发过程中可以**在构造方法中判断是否已经存在实例，如果存在则抛出异常，防止通过反射创建新的实例**。

航仔丶的回答

单例模式是一种**创建型设计模式**，它确保**一个类只有一个实例**，并提供**全局访问点**。使用单例模式可以**减少系统性能开销，节约资源**，对于一般频繁创建和销毁对象的可以使用单例模式。因为它限制了实例的个数，有利于垃圾回收。好的单例模式也能提高性能。单例模式常常用于**任务管理器**，**回收站**，**网站的计数器**等场景。

饿汉式

饿汉式是一种在**类被加载时就创建对象的方法**，因此不存在线程安全问题。但是，它可能导致类装载的原因有很多种，在单例模式中大多数都是调用 getInstance 方法，但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 显然没有达到 lazy loading 的效果。

```
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton() {}
    public static Singleton getInstance() {
        return instance;
    }
}
```

2.**懒汉式**

懒汉式是一种在**第一次使用时才创建对象的方法**，因此可能存在**线程安全**问题。在多线程环境下，当多个线程同时调用 getInstance 方法时，可能会创建多个实例，因此需要使用同步锁 **synchronized** 防止多线程同时进入造成 instance 被多次实例化。但是，**使用同步锁会降低程序的效率**。

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

3.**双重检查锁实现懒汉式单例模式**

在第一次调用时创建对象实例，如果对象已经创建则直接返回。**双重检查锁实现懒汉式单例模式**的写法如下：

```
public class SingletonDCL {
    private volatile static SingletonDCL instance;
    private SingletonDCL() {}
    public static SingletonDCL getInstance() {
        if (instance == null) {
            synchronized (SingletonDCL.class) {
                if (instance == null) {
                    instance = new SingletonDCL();
                }
            }
        }
        return instance;
    }
}
```

4.**静态内部类实现单例模式**

在第一次调用时创建对象实例，如果对象已经创建则直接返回。静态内部类实现单例模式的写法如下：

```
public class SingletonInnerClass {
    private SingletonInnerClass() {}
    public static SingletonInnerClass getInstance() {
        return LazyHolder.INSTANCE;
    }
    private static class LazyHolder {
        private static final SingletonInnerClass INSTANCE = new SingletonInnerClass();
    }
}
```

5.**枚举类实现单例模式**

在第一次调用时创建对象实例，如果对象已经创建则直接返回。**枚举类实现单例模式**的写法如下：

```
public enum SingletonEnum {
    INSTANCE;
    public void method() {
        System.out.println("枚举类中定义方法！");
    }
}
```

3、设计模式可以分为哪几类？一共有多少种主流的设计模式？

官方解析

设计模式可以分为三类：

1. **创建型模式**：这类模式关注对象创建的机制，包括单例模式、工厂模式、抽象工厂模式、建造者模式和原型模式等。
2. **结构型模式**：这类模式关注对象之间的组合关系，包括适配器模式、装饰器模式、代理模式、组合模式、桥接模式、外观模式和享元模式等。
3. **行为型模式**：这类模式关注对象之间的通信方式和协作方式，包括模板方法模式、策略模式、命令模式、职责链模式、状态模式、观察者模式、中介者模式和访问者模式等。
   目前主流的设计模式有23种，它们分别是：
4. 单例模式
5. 工厂方法模式
6. 抽象工厂模式
7. 建造者模式
8. 原型模式
9. 适配器模式
10. 装饰器模式
11. 代理模式
12. 外观模式
13. 桥接模式
14. 组合模式
15. 享元模式
16. 策略模式
17. 模板方法模式
18. 观察者模式
19. 迭代器模式
20. 职责链模式

21. 命令模式
22. 备忘录模式
23. 状态模式
24. 访问者模式
25. 中介者模式
26. 解释器模式

鱼友的精彩回答

猫十二懿的回答

1. 创建型模式（Creational Patterns）：这些模式专注于对象的创建，解决了如何创建对象的问题。通常，这些模式都有一个工厂对象，负责创建其他对象。

共五种设计模式：简单工厂模式、工厂方法模式、抽象工厂模式、单例模式和建造者模式。

2. 结构型模式（Structural Patterns）：模式专注于对象的组合和关联，解决了如何组成对象的问题，都涉及到类和对象之间的关系。

共七种设计模式，分别是适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式和享元模式。

3. 行为型模式（Behavioral Patterns）：这些模式专注于对象之间的通信，解决了如何让对象相互协作的问题，都涉及到算法和对象的职责分配。

共十一种设计模式：策略模式、模板方法模式、观察者模式、迭代器模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式和解释器模式。 主流的设计模式共有 23 种，它们是：

4. **简单工厂模式（Simple Factory Pattern）**：一个工厂对象负责根据传入的参数创建不同的对象实例。
5. **工厂方法模式（Factory Method Pattern）**：每个具体的类都有自己的工厂方法，负责创建对象实例。
6. **抽象工厂模式（Abstract Factory Pattern）**：一组相关或相互依赖的对象，由一个抽象工厂对象负责创建。
7. **单例模式（Singleton Pattern）**：确保类只有一个实例，并提供全局访问点。
8. **建造者模式（Builder Pattern）**：将一个复杂对象的构建过程分解为多个简单对象的构建过程。
9. **原型模式（Prototype Pattern）**：一种创建型设计模式，它通过克隆已有对象来创建新的对象，而不是通过实例化对象来创建。
10. **适配器模式（Adapter Pattern）**：将一个类的接口转换成客户希望的另一个接口。
11. **装饰器模式（Decorator Pattern）**：动态地为对象增加新的功能。
12. **代理模式（Proxy Pattern）**：为其他对象提供一种代理以控制对这个对象的访问。
13. **外观模式（Facade Pattern）**：为复杂的子系统提供一个简单的接口。
14. **桥接模式（Bridge Pattern）**：将抽象部分与它的实现部分分离，使它们都可以独立地变化。
15. **组合模式（Composite Pattern）**：将对象组合成树形结构以表示部分-整体的层次结构。
16. **享元模式（Flyweight Pattern）**：通过共享技术来实现大量细粒度对象的复用。
17. **策略模式（Strategy Pattern）**：定义一系列算法，将它们封装起来，并且使用他们可以相互替换。

18. **模板方法模式（Template Method Pattern）**：定义一个算法的骨架，将一些步骤延迟到子类中实现。
19. **观察者模式（Observer Pattern）**：定义对象间的一种一对多的依赖关系，当一个对象状态发生改变时，所有依赖于它的对象都得到通知并自动更新。
20. **迭代器模式（Iterator Pattern）**：提供一种方法顺序访问一个聚合对象中各个元素，而且又不暴露该对象的内部表示。
21. **责任链模式（Chain of Responsibility Pattern）**：为某个请求创建一个对象链，处理该请求的对象沿着该练依次处理，知道一个对象处理它为止。
22. **命令模式（Command Pattern）**：将一个请求封装成一个对象，使发出请求的责任和执行请求的责任分割开。
23. **备忘录模式（Memento Pattern）**：它允许在不暴露对象实现细节的情况下，保存和恢复对象之前的状态。
24. **状态模式（State Pattern）**：允许对象在其内部状态改变时改变它的行为。
25. **访问者模式（Visitor Pattern）**：封装一些作用于某种数据结构中的各元素的操作，它可以在不改变这个数据结构的前提下定义作用于这些元素的新操作。
26. **中介者模式（Mediator Pattern）**：通过封装一系列对象之间的交互，来降低对象之间的耦合度。

> 作者：编程导航知识星球 + 星球鱼友们

4、什么是工厂模式？使用工厂模式有什么好处？工厂模式有哪些分类？各自的应用场景是什么？

官方解析

工厂模式（Factory Pattern）是一种创建型设计模式，它提供了一种封装对象创建过程的方法。工厂模式将对象的创建和使用分离，让一个专门的工厂类负责创建对象实例，而不是在代码中直接使用 new 操作符。这有助于降低代码的耦合度，提高可维护性和可扩展性。

使用工厂模式的好处：

1. 降低耦合度：工厂模式将对象的创建与使用分离，使得客户端代码不直接依赖具体的类，降低了耦合度。
2. 提高可扩展性：当需要添加或修改产品类时，只需修改工厂类，而不需要修改客户端代码，提高了系统的可扩展性。
3. 提高可维护性：通过集中管理对象的创建，提高了代码的可维护性。
4. 提高代码复用性：工厂类可以被多个客户端代码复用，减少了重复代码。

工厂模式可以分为以下几类：

1. 简单工厂模式（Simple Factory Pattern）：一个工厂类根据传入的参数决定创建哪个具体产品类的实例。简单工厂模式适用于产品种类较少且不易变化的场景。 应用场景：例如，一个简单的图形绘制工具，可以根据传入的参数创建不同类型的图形（如圆形、矩形等）。
2. 工厂方法模式（Factory Method Pattern）：定义一个接口或抽象类来创建对象，将实际创建对象的工作推迟到子类中。工厂方法模式适用于产品种类较多且可能增加的场景。 应用场景：例如，一个日志记录器，可以根据不同的需求（如文件日志、数据库日志等）使用不同的工厂子类创建相应的日志记录器实例。

3. 抽象工厂模式（Abstract Factory Pattern）：提供一个接口，用于创建一系列相关或相互依赖的对象，而无需指定它们具体的类。抽象工厂模式适用于产品族的场景。 应用场景：例如，跨平台UI框架，可以为不同平台（如Windows、macOS等）提供不同的UI控件实现，通过抽象工厂来创建对应平台的一系列UI控件。

各种工厂模式的应用场景取决于实际需求，需要根据具体问题来选择合适的工厂模式。

鱼友的精彩回答

维萨斯的回答

循序渐进循序渐进

从定义开始

- 工厂模式
    - 把对象的构造交给一个类,这样在对象构造过程中,调用者不需要知道对象的产生过程。降低了代码的耦合度，同时体现了面向对象的封装的特征
    - 好处 (不是绝对的，仍然需要具体问题具体分析)
        - **松耦合**
        - **封装**
        - **可拓展**
        - **复用**

再看分类 - 从缺点看他的用处

简单工厂模式 （一个工厂生产所有的对象 ）

- 如果它负责的对象太多，简单工厂容易庞大，变成超级类
- 简单工厂的拓展是竖向拓展（拓展需要访问工厂内部，职责不够单一）
- 因此，他适合在'简单场景' - 对象少且固定

工厂方法模式 （ 一个工厂一个对象 ）

- 工厂方法是横向拓展，由于工厂对象1对1，耦合度没有任何减少，要生产类就要知道那个对应的工厂
- 当需要生产新的产品时，无需更改既有的工厂，只需要添加新的工厂即可。保持了面向对象的可扩展性(纵向，横向都很不错)，符合开闭原则。
- 因此，工厂方法模式适用于对象多而且可能增加的场景

抽象工厂模式 （ 抽象接口工厂实现 ）

- 抽象工厂是定义了一个接口，然后由具体工厂去实现抽象方法，突出特点就是运用了多态，耦合够松。
- 缺点也很明显，在接口不变的情况下，无论是横向拓展性还是**松耦合**都很好，但是一旦接口有新的功能增加，所有的工厂实现都需要纵向拓展。即横向拓展能力强，但是纵向拓展能力=0
- 因此，抽象工厂模式适合产品族（即有相似属性）的生产

林寻的回答

创建者模式

二.工厂模式

在 java 中，万物皆对象，这些对象都需要创建，如果创建的时候直接 new 该对象，就会对该对象耦合严重，假如我们要更换对象，所有 new 对象的地方都需要修改一遍，这显然违背了软件设计的开闭原则。如果我们使用工厂来生产对象，我们就只和工厂打交道就可以了，彻底和对象解耦，如果要更换对象，直接在工厂里更换该对象即可，达到了与对象解耦的目的；所以说，工厂模式最大的优点就是：**解耦**。

- 简单工厂模式（不属于 GOF 的 23 种经典设计模式）
- 工厂方法模式
- 抽象工厂模式

1.简单工厂模式

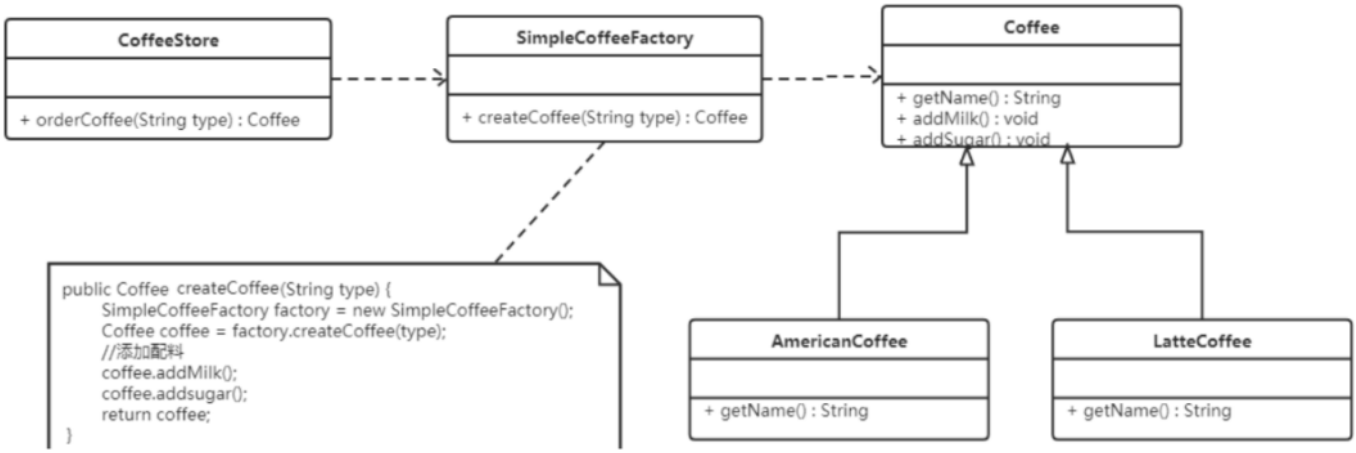
简单工厂不是一种设计模式，反而比较像是一种编程习惯。

1.1 结构

简单工厂包含如下角色：

抽象产品： 定义了产品的规范，描述了产品的主要特性和功能。
具体产品： 实现或者继承抽象产品的子类
具体工厂： 提供了创建产品的方法，调用者通过该方法来获取产品。

1.2 实现



```
public class SimpleCoffeeFactory {
•
    public Coffee createCoffee(String type) {
        Coffee coffee = null;
        if("americano".equals(type)) {
            coffee = new AmericanoCoffee();
        } else if("latte".equals(type)) {
            coffee = new LatteCoffee();
        }
        return coffee;
    }
}
```

工厂（factory）处理创建对象的细节，一旦有了 SimpleCoffeeFactory，CoffeeStore 类中的 orderCoffee()就变成此对象的客户，后期如果需要Coffee 对象直接从工厂中获取即可。这样也就解除了和Coffee实现类的耦合，同时又产生了新的耦合，CoffeeStore 对象和 SimpleCoffeeFactor y工厂对象的耦合，工厂对象和商品对象的耦合。

后期如果再加新品种的咖啡，我们势必要需求修改 SimpleCoffeeFactory 的代码，违反了开闭原则。工厂类的客户端可能有很多，比如创建美团外卖等，这样只需要修改工厂类的代码，省去其他的修改操作。

1.3 优缺点

优点：
封装了创建对象的过程，可以通过参数直接获取对象。把对象的创建和业务逻辑层分开，这样以后就避免了修改客户代码，如果要实现新产品直接修改工厂类，而不需要在原代码中修改，这样就降低了客户代码修改的可能性，更加容易扩展。

缺点：
增加新产品时还是需要修改工厂类的代码，违背了“开闭原则”。

1.4 扩展

静态工厂
在开发中也有一部分人将工厂类中的创建对象的功能定义为静态的，这个就是静态工厂模式，它也不是23种设计模式中的。代码如下：

```
public class SimpleCoffeeFactory {
•
    public static Coffee createCoffee(String type) {
        Coffee coffee = null;
        if("americano".equals(type)) {
            coffee = new AmericanoCoffee();
        } else if("latte".equals(type)) {
            coffee = new LatteCoffee();
        }
        return coffe;
    }
}
```

2.工厂方法模式

针对上例中的缺点，使用工厂方法模式就可以完美的解决，完全遵循开闭原则。

2.1 概念

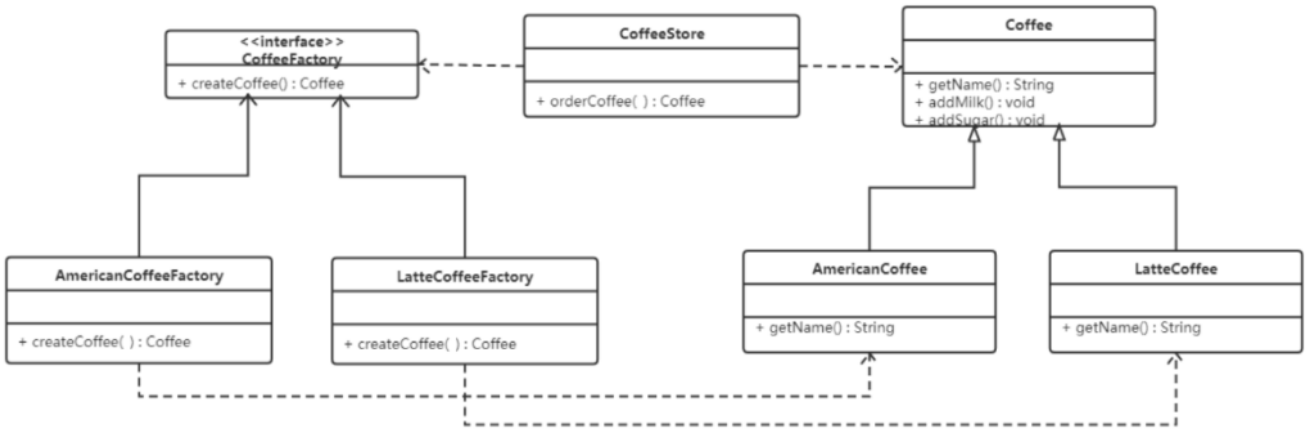
定义一个用于创建对象的接口，让子类决定实例化哪个产品类对象。工厂方法使一个产品类的实例化延迟到其工厂的子类。

2.2 结构

工厂方法模式的主要角色：

抽象工厂（Abstract Factory）：提供了创建产品的接口，调用者通过它访问具体工厂的工厂方法来创建产品。
具体工厂（ConcreteFactory）：主要是实现抽象工厂中的抽象方法，完成具体产品的创建。
抽象产品（Product）：定义了产品的规范，描述了产品的主要特性和功能。
具体产品（ConcreteProduct）：实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间一一对应。

2.3 实现




```
public interface CoffeeFactory {

    Coffee createCoffee();
}
```

```
public class LatteCoffeeFactory implements CoffeeFactory {

    public Coffee createCoffee() {
        return new LatteCoffee();
    }
}

public class AmericanCoffeeFactory implements CoffeeFactory {

    public Coffee createCoffee() {
        return new AmericanCoffee();
    }
}
```

```
public class CoffeeStore {

    private CoffeeFactory factory;

    public CoffeeStore(CoffeeFactory factory) {
        this.factory = factory;
    }

    public Coffee orderCoffee(String type) {
        Coffee coffee = factory.createCoffee();
        coffee.addMilk();
        coffee.addsugar();
        return coffee;
    }
}
```

要增加产品类时也要相应地增加工厂类，不需要修改工厂类的代码了，这样就解决了简单工厂模式的缺点。

工厂方法模式是简单工厂模式的进一步抽象。由于使用了多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。

2.4 优缺点

优点：

- 用户只需要知道具体工厂的名称就可得到所要的产品，无须知道产品的具体创建过程；
- 在系统增加新的产品时只需要添加具体产品类和对应的具体工厂类，无须对原工厂进行任何修改，满足开闭原则；
  缺点：
- 每增加一个产品就要增加一个具体产品类和一个对应的具体工厂类，这增加了系统的复杂度。

3. 抽象工厂模式

前面介绍的工厂方法模式中考虑的是一类产品的生产，如畜牧场只养动物、电视机厂只生产电视机、传智播客只培养计算机软件专业的学生等。

这些工厂只生产同种类产品，同种类产品称为同等级产品，也就是说：工厂方法模式只考虑生产同等级的产品，但是在现实生活中许多工厂是综合型的工厂，能生产多等级（种类） 的产品，如电器厂既生产电视机又生产洗衣机或空调，大学既有软件专业又有生物专业等。

本节要介绍的抽象工厂模式将考虑多等级产品的生产，将同一个具体工厂所生产的位于不同等级的一组产品称为一个产品族，下图所示横轴是产品等级，也就是同一类产品；纵轴是产品族，也就是同一品牌的产品，同一品牌的产品产自同一个工厂。

3.1 概念

是一种为访问类提供一个创建一组相关或相互依赖对象的接口，且访问类无须指定所要产品的具体类就能得到同族的不同等级的产品的模式结构。

抽象工厂模式是工厂方法模式的升级版本，工厂方法模式只生产一个等级的产品，而抽象工厂模式可生产多个等级的产品。

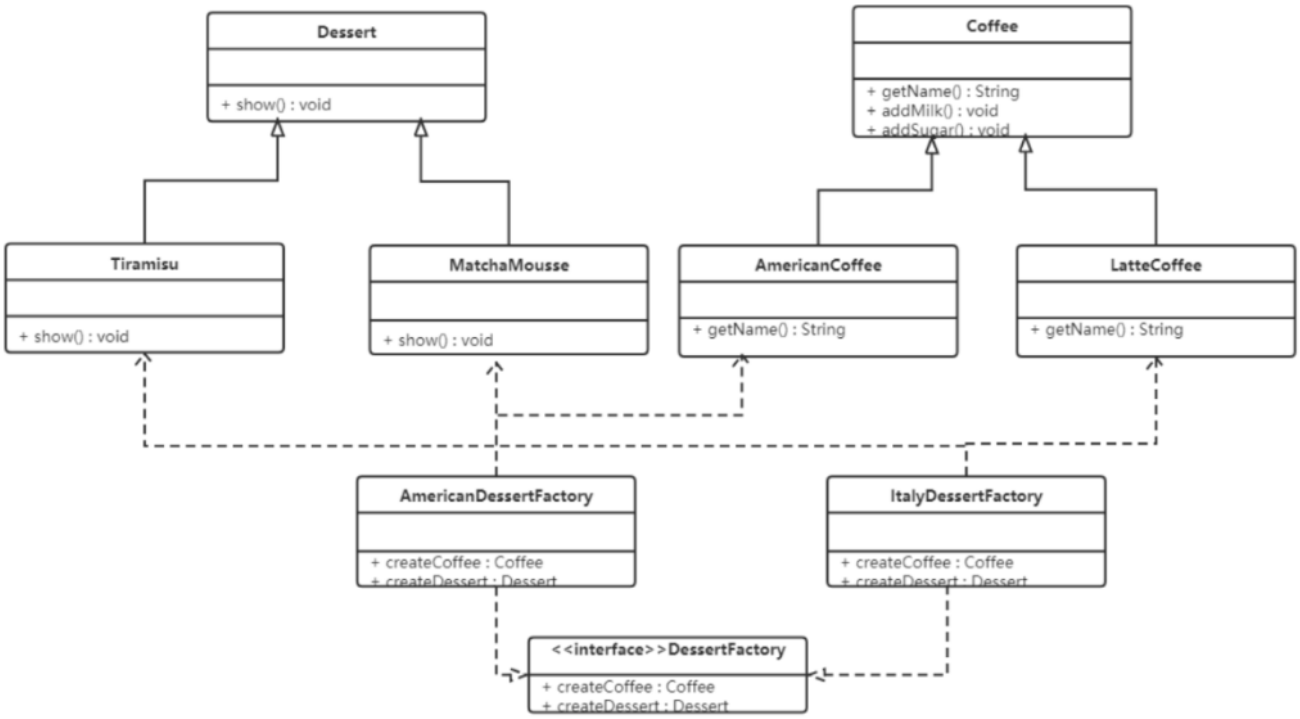
3.2 结构

抽象工厂模式的主要角色如下：

- 抽象工厂（Abstract Factory）：提供了创建产品的接口，它包含多个创建产品的方法，可以创建多个不同等级的产品。
- 具体工厂（Concrete Factory）：主要是实现抽象工厂中的多个抽象方法，完成具体产品的创建。
- 抽象产品（Product）：定义了产品的规范，描述了产品的主要特性和功能，抽象工厂模式有多个抽象产品。
- 具体产品（ConcreteProduct）：实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间是多对一的关系。

3.3 实现

现咖啡店业务发生改变，不仅要生产咖啡还要生产甜点，如提拉米苏、抹茶慕斯等，要是按照工厂方法模式，需要定义提拉米苏类、抹茶慕斯类、提拉米苏工厂、抹茶慕斯工厂、甜点工厂类，很容易发生类爆炸情况。其中拿铁咖啡、美式咖啡是一个产品等级，都是咖啡；提拉米苏、抹茶慕斯也是一个产品等级；拿铁咖啡和提拉米苏是同一产品族（也就是都属于意大利风味），美式咖啡和抹茶慕斯是同一产品族（也就是都属于美式风味）。所以这个案例可以使用抽象工厂模式实现。类图如下：




```
public interface DessertFactory {
•
    Coffee createCoffee();
•
    Dessert createDessert();
}
```

```
//美式甜点工厂
public class AmericanDessertFactory implements DessertFactory {
```

```
    public Coffee createCoffee() {
        return new AmericanCoffee();
    }

    public Dessert createDessert() {
        return new MatchaMousse();
    }
}
//意大利风味甜点工厂
public class ItalyDessertFactory implements DessertFactory {

    public Coffee createCoffee() {
        return new LatteCoffee();
    }

    public Dessert createDessert() {
        return new Tiramisu();
    }
}
```

如果要加同一个产品族的话，只需要再加一个对应的工厂类即可，不需要修改其他的类。

3.4 优缺点

优点：

当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

缺点：

当产品族中需要增加一个新的产品时，所有的工厂类都需要进行修改。

3.5 使用场景

- 当需要创建的对象是一系列相互关联或相互依赖的产品族时，如电器工厂中的电视机、洗衣机、空调等。
- 系统中有多个产品族，但每次只使用其中的某一族产品。如有人只喜欢穿某一个品牌的衣服和鞋。
- 系统中提供了产品的类库，且所有产品的接口相同，客户端不依赖产品实例的创建细节和内部结构。

如：输入法换皮肤，一整套一起换。生成不同操作系统的程序。

3.6 模式扩展

简单工厂+配置文件解除耦合

可以通过工厂模式+配置文件的方式解除工厂对象和产品对象的耦合。在工厂类中加载配置文件中的全类名，并创建对象进行存储，客户端如果需要对象，直接进行获取即可。

第一步：定义配置文件

为了演示方便，我们使用 properties 文件作为配置文件，名称为 bean.properties

american=com.itheima.pattern.factory.config_factory.AmericanCoffee

latte=com.itheima.pattern.factory.config_factory.LatteCoffee

```
public class CoffeeFactory {
•
    private static Map<String,Coffee> map = new HashMap();
•
    static {
        Properties p = new Properties();
        InputStream is =
CoffeeFactory.class.getClassLoader().getResourceAsStream("bean.properties");
        try {
            p.load(is);
            //遍历Properties集合对象
            Set<Object> keys = p.keySet();
            for (Object key : keys) {
                //根据键获取值（全类名）
                String className = p.getProperty((String) key);
                //获取字节码对象
                Class clazz = Class.forName(className);
                Coffee obj = (Coffee) clazz.newInstance();
                map.put((String)key,obj);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
•
    public static Coffee createCoffee(String name) {
•
        return map.get(name);
    }
}
```

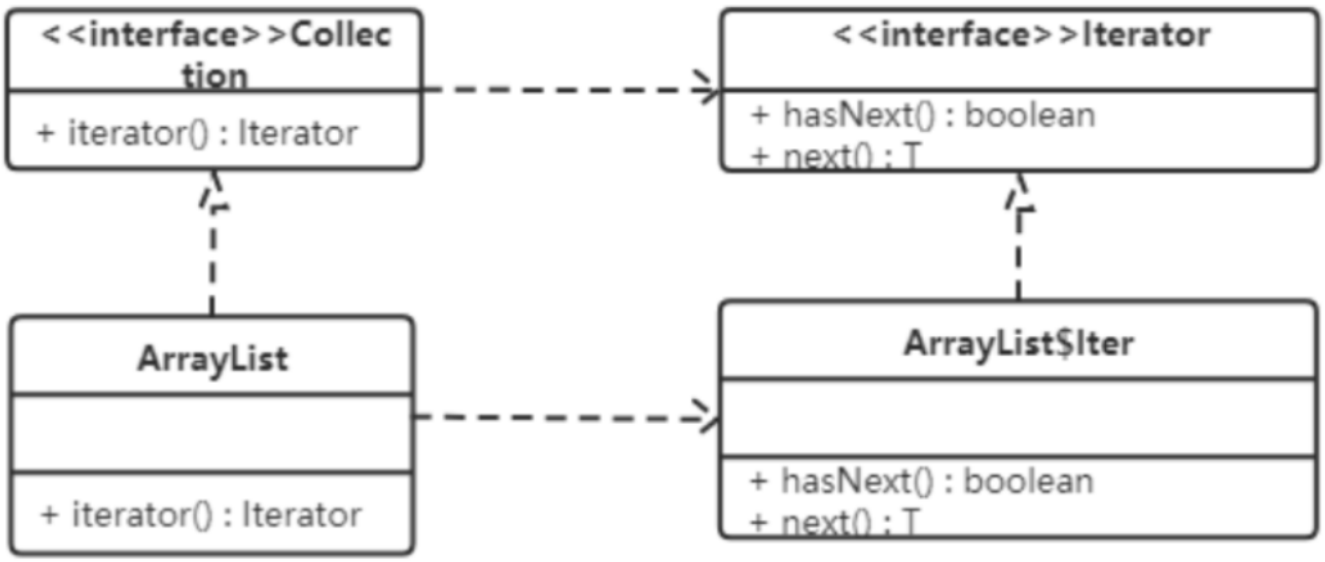
静态成员变量用来存储创建的对象（键存储的是名称，值存储的是对应的对象），而读取配置文件以及创建对象写在静态代码块中，目的就是只需要执行一次。

JDK源码解析-Collection.iterator 方法

```
public class Demo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("令狐冲");
        list.add("风清扬");
        list.add("任我行");
•
        //获取迭代器对象
        Iterator<String> it = list.iterator();
        //使用迭代器遍历
        while(it.hasNext()) {
```

```
            String ele = it.next();
            System.out.println(ele);
        }
    }
}
```

使用迭代器遍历集合，获取集合中的元素。而单列集合获取迭代器的方法就使用到了工厂方法模式。我们看通过类图看看结构



Collection 接口是抽象工厂类，ArrayList是具体的工厂类；Iterator 接口是抽象商品类，ArrayList 类中的 Iter 内部类是具体的商品类。在具体的工厂类中 iterator()方法创建具体的商品类的对象。

另：

1,DateForamt 类中的 getInstance()方法使用的是工厂模式；

2,Calendar 类中的 getInstance()方法使用的是工厂模式；

并发编程

1、并发和并行有什么区别？同步和异步有什么区别？

官方解析

并发和**并行**是两个计算机领域中经常被提到的概念，它们的含义有所不同。

- **并发**（Concurrency）：指的是系统中**同时存在多个正在执行的任务，并且这些任务之间可能会相互影响**。并发通常用来处理**多个任务共享资源**的情况。在单核 CPU 上，多个任务会轮流使用 CPU 时间片，表现为看似同时执行的情况，但实际上只有一个任务正在执行。
- **并行**（Parallelism）：指的是系统中**同时存在多个并且相互独立的任务**，并且这些任务可以在多个处理器上同时执行，真正意义上的同时处理多个任务。
- **同步**（Synchronous）：指的是程序**按照代码的顺序执行**，一行一行地执行，直到当前行执行完成后才能继续执行下一行。同步通常会**阻塞**调用者，直到任务完成才能返回。

- **异步**（Asynchronous）：指的是程序在执行某个任务时，不会一直等待任务完成，而是继续执行下一行代码，**当任务完成后再进行相应的处理**。异步通常**不会阻塞**调用者，可以**提高系统的并发性能**。

总的来说，"并发"和"并行"是针对多个任务的执行方式，"同步"和"异步"是针对任务执行的阻塞方式和返回方式。在实际应用中，可以根据不同的需求来选择合适的并发和同步方式，以提高系统的性能和可靠性。

鱼友的精彩回答

苏打饼干的回答

	含义	场景
并发	多个时间点同一时间间隔发送 宏观上同时发送，微观上交替进行	时间片轮转
并行	真正意义上的同时运行	多个处理器处理 多个独立的任务
同步	为完成某种任务，程序需要在某些位置上协调它们的工作次序而产生的制约关系，又称直接制约关系	进程同步
异步	程序执行某个任务时，不需等待该任务完成，可以直接执行下一个任务，等任务完成后再进行相应处理，提高系统并发性能	并行计算 并发编程

并发和并行针对的是任务的执行方式，同步和异步针对的是任务执行的阻塞和返回方式。

爱吃鱼蛋的回答

先介绍并发和并行的区别：

并发：指多个任务同时在执行，但是它们并不是在同一时刻执行，而是通过**快速切换上下文**来模拟同时执行，优秀的并发可以无限逼近并行，但无法完全做到并行。

例如，在一个 Web 服务器上，多个用户访问同一个网站，服务器会并发地处理这些请求，同时响应每个请求，但是在某一时刻只有一个请求被处理；

并行：指多个任务同时在执行，并且它们真正地同时执行，通常需要多个 CPU 或者多台计算机协同工作。

例如，在一个分布式计算系统中，不同的计算节点可以同时执行不同的任务，并在完成任务后将结果汇总，以加速计算的过程。

接下来是同步和异步：

同步：指调用某个函数或方法时，程序必须等待函数或方法执行完毕才能继续往下执行。

最熟知的便是使用同步机制来控制多个线程之间的访问，如在 Java 中的 synchronized 关键字可以确保同一时间只有一个线程可以访问某个对象的临界区，避免了多个线程同时修改同一个对象导致的数据不一致问题；

异步：指调用某个函数或方法时，程序可以继续往下执行，不必等待函数或方法执行完毕。当函数或方法执行完毕后，程序会得到一个通知或回调来处理结果。在 Java 中可以使用 CompletableFuture 类来异步执行某个任务，从而提高程序的性能。

虽然并发和并行、同步和异步都是计算机领域中常用的概念，但是它们的区别还是很明显的。**并发和并行关注的是任务的执行方式，同步和异步则关注的是数据的处理方式**。

2、什么是 BIO、NIO、AIO?

官方解析

BIO、NIO、AIO 都是 Java 的 IO 模型。

BIO (Blocking IO) 是传统的 IO 模型，它在读写数据时会**阻塞线程**，直到数据读写完成，适用于**并发不高**的场景。

NIO (Non-blocking IO) 是 Java 的新 IO 模型，它在读写数据时不会阻塞线程，而是通过**轮询**的方式检查是否有数据可读写，适用于**并发量较高**的场景。

AIO (Asynchronous IO) 是 JDK 7 开始引入的新 IO 模型，它的读写方式与 NIO 相似，但在读写数据时，不需要自己手动轮询是否有数据可读写，而是**交由系统完成**，适用于**高并发且处理较大数据量**的场景。

总的来说，BIO 的并发处理能力较差，NIO 的并发处理能力较好，但使用起来较为复杂，AIO 的并发处理能力最好，但也是最为复杂的一种 IO 模型。选择适合自己场景的 IO 模型是非常重要的。

> 鱼皮补充：这题比较建议大家举一个例子来区分这些概念，比如：人等待水壶烧水

鱼友的精彩回答

一切总会归于平淡的回答

BIO、NIO、AIO 是 Java 中常用的 I/O 模型，它们都是针对网络编程中 I/O 操作的不同实现方式。

BIO，全称 Blocking I/O，也称为同步阻塞 I/O。

BIO 是最早的 I/O 模型，它是一种阻塞式的 I/O 模型，即当应用程序调用 I/O 操作时，该操作会一直**阻塞线程直到操作完成**，这会导致 **I/O 性能低下**。BIO **适用于连接数较小的场景**，例如单线程的服务器模型。

NIO，全称 Non-Blocking I/O，也称为同步非阻塞 I/O。

NIO 是一种同步非阻塞的 I/O 模型，它的核心是 Selector 和 Channel，利用 Selector 监听多个 Channel 上的事件，当一个 Channel 上的事件到达时，它会被 Selector 转发给注册在这个 Selector 上的其他 Channel，这样可以用一个线程来处理多个请求，提高了 I/O 的效率。NIO 适用于连接数多、连接时间短的场景，例如实时聊天室、在线游戏等。

AIO，全称 Asynchronous I/O，也称为异步非阻塞 I/O。AIO 是 JDK1.7 引入的 I/O 模型，它的特点是异步处理 I/O 操作，当操作完成时会通知应用程序，相比于 NIO 的同步非阻塞 I/O，AIO 无需通过轮询操作完成状态，从而提高了 I/O 的效率。AIO 适用于连接数多、连接时间长的场景，例如 HTTP 长连接、文件操作等。

面试官问到这个问题，可能想了解你对 Java 中不同的 I/O 模型的了解程度，以及你能否根据不同的业务需求选择合适的 I/O 模型。同时，他可能还想了解你在实际开发中使用过哪些 I/O 模型，以及它们的应用场景。

猫十二懿的回答

BIO、NIO、AIO 都是 Java 中网络编程的 I/O 模型。

BIO（Blocking IO ）是JDK1.4之前的传统IO模型，特点就是同步阻塞等待数据，直到数据读取完毕才会返回结果，线程会一直阻塞在 read/write 方法上，不能处理其他的 IO 请求，它的并发性能比较差。

NIO（Non-Blocking IO）是Java 1.4 之后新增的 IO 模型，它支持同步非阻塞式的 IO 操作。NIO 采用了多路复用器来处理 IO 请求，通过一个线程处理多个 IO 请求，实现了高并发处理。NIO 主要有三个核心概念：Selector、Channel、Buffer。Selector 负责监听多个 Channel 上的事件，Channel 可以理解为对原始 IO 的封装，Buffer 则是对数据的封装。

AIO（Asynchronous IO）是Java 1.7 之后新增的 IO 模型，它支持异步非阻塞 IO 操作。与 NIO 不同的是，AIO 在进行读写操作时不需要像 NIO 一样一直轮询，而是通过回调函数的方式在数据准备好后通知应用程序进行数据的读取，这样可以更加高效地利用系统资源，提高吞吐量。但是 AIO 在处理小文件和小数据量时的性能并不如 NIO。

三者区别

	BIO	AIO	NIO
处理IO方式	阻塞IO	非阻塞IO	异步IO
缺点	并发性能差	实现复杂	虽弥补了并发能力，NIO代码比较复杂
适合场景	连接数较小且固定	链接数非常多且链接时间比较长	连接数多且变化较少

白小军的回答

BIO 同步阻塞 IO，即打算约女神，给女神发短信后，没见到女神就一直等在宿舍楼下。

NIO 同步非阻塞 IO，即打算约女神，给女神发短信后，没见到女神就一直发短信。

NIO java中的 NIO，就是打算约女神，你让宿管大妈去挨个看每一个下楼的妹子，女神下楼了大妈就通知你。

AIO 就是打算约女神，你发完短信，你就去玩游戏了，女神下楼了，发短信给你，你才出现。

3、线程的生命周期是什么，线程有几种状态，什么是上下文切换?

官方解析

线程的生命周期通常包括五种状态：**新建（New）**、**就绪（Runnable）**、**运行（Running）**、**阻塞（Blocked）**和**终止（Terminated）**。其中：

- 新建状态是指当线程对象创建后，就进入了新建状态。此时它已经有了相应的内存空间和其他资源，但还没有开始运行。
- 就绪状态是指当线程对象调用 start() 方法后，线程进入了就绪状态。此时它已经具备了运行的条件，只等 CPU 分配资源，让其开始执行。
- 运行状态是指当线程对象获得 CPU 资源后，就开始执行 run() 方法中的代码，线程处于运行状态。
- 阻塞状态是指线程在某些特定情况下会被挂起，暂时停止执行。当线程处于阻塞状态时，它并不会占用 CPU 资源。
- 终止状态是指当线程的 run() 方法执行完毕或者因异常退出时，线程进入了终止状态。此时，该线程不再占用 CPU 资源，也不再执行任何代码。

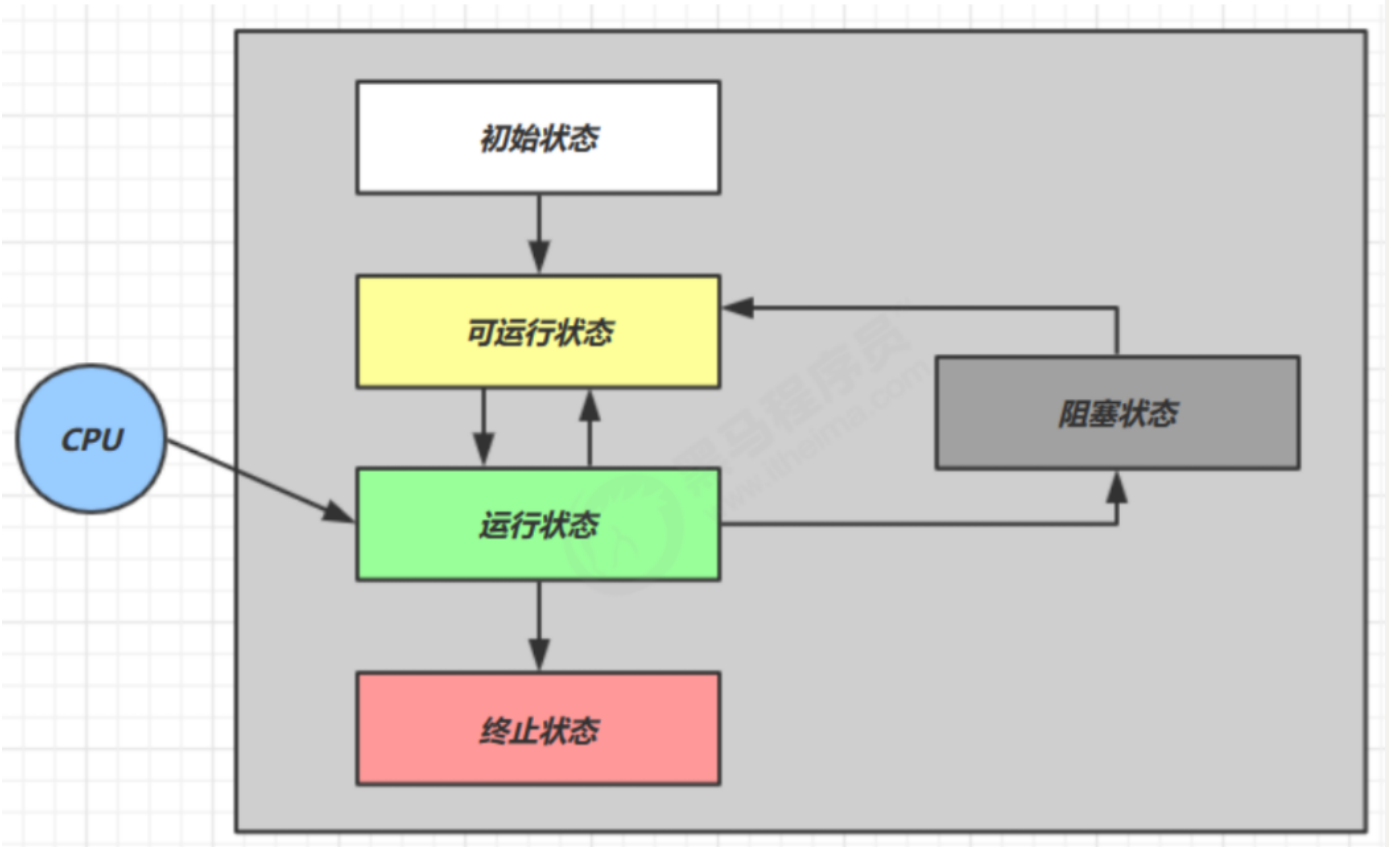
在线程的生命周期中，线程状态的转换通常是由**操作系统调度和控制**的。当线程的状态发生变化时，**需要进行上下文切换**，即保存当前线程的状态和上下文信息，并恢复另一个线程的状态和上下文信息，使其能够继续执行。**上下文切换会带来一定的开销，因此需要尽可能减少线程之间的切换次数**。

鱼友的精彩回答

苏打饼干的回答

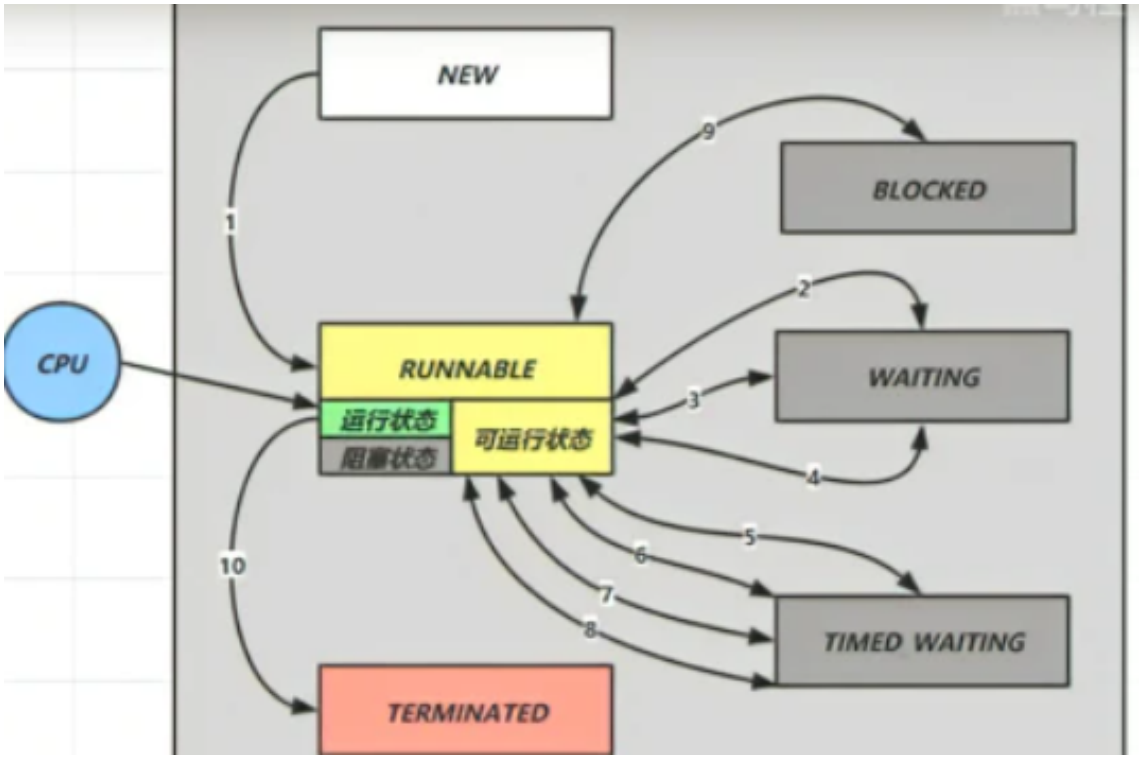
从传统操作系统层面，线程有五大基本状态，包括：**创建、就绪、运行、阻塞和终止状态**

状态类别	说明
创建态	线程**正在被创建时**的状态（分配相应内存空间、资源等）
就绪态	线程**创建完成**以后，具备运行条件，但由于**没有空闲**CPU暂时无法运行 此时线程位于可运行线程池中，**等待获取CPU的使用权**
运行态	线程在CPU**上运行时**的状态
阻塞态	线程运行过程中，请求**等待**某个事件发生或因其他原因，**无法往下执行** 此时操作系统会让这个线程**暂时放弃CPU使用权**，并让其进入阻塞态
终止态	线程**结束生命周期**（释放资源，彻底消失）



从 Java 并发编程的角度来看，线程有六个状态，包括：**新建、就绪、阻塞、等待、超时等待和终止状态**；

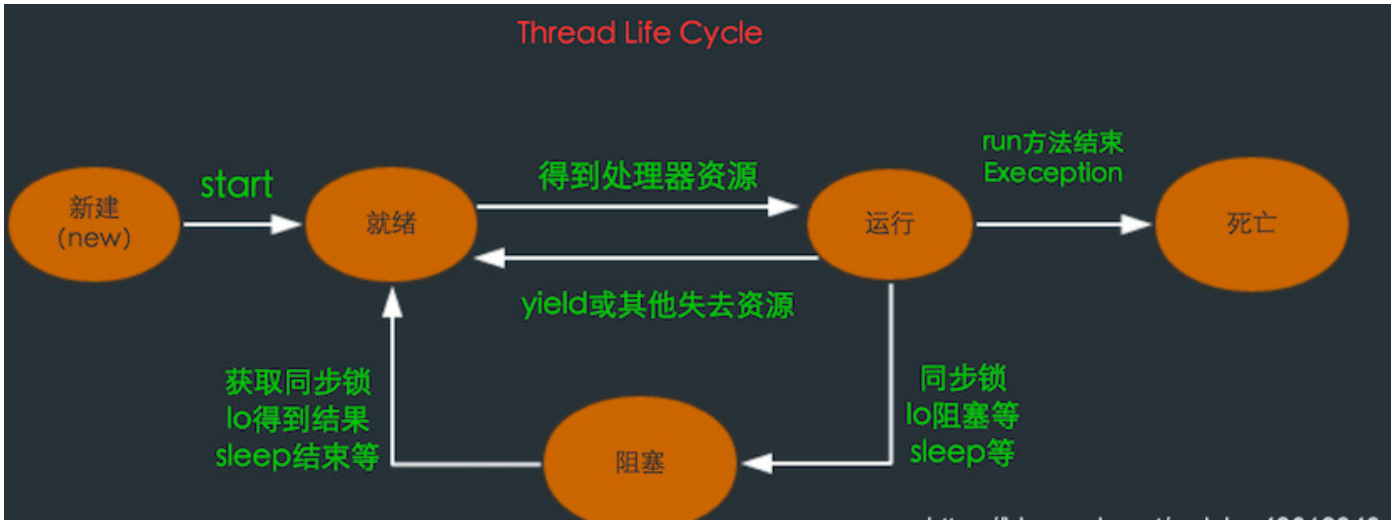
状态类别	说明	类比操作系统
新建态（New）	线程**正在被新建时**的状态 此时线程并未启动	创建态
可运行态（Runnable）	调用线程的 `start()` 方法时，线程进入可运行状态 处于等待系统调度运行或运行的状态	就绪、运行态
阻塞态（Blocked）	当线程等待获取一个 `排它锁` 时，线程会进入阻塞状态 线程不会消耗 CPU 时间，只有在 `获得锁` 之后才能转入可运行状态	阻塞态
等待态（Waiting）	当线程调用 `wait()`、`join()` 或 `sleep()` 方法时，线程会进入等待状态 线程不会消耗 CPU 时间，只有在其他线程唤醒它之后才能转入可运行状态	阻塞态
计时等待态（Timed Waiting）	当线程调用带有超时参数的 `sleep()`、`wait()` 或 `join()` 方法时，线程会进入计时等待状态 线程不会消耗 CPU 时间，只有在超时时间到达或其他线程唤醒它之后才能转入可运行状态	阻塞态
终止态（Terminated）	线程执行完 `run()` 方法或抛出异常退出时，线程会进入终止状态 线程结束自身的生命周期	终止态



上下文切换：指将当前线程的状态保存下来，并将 CPU 资源切换到另一个线程上运行的过程，通常由操作系统进行管理和控制的。需要注意，上下文切换需要花费一定的时间和系统资源，线程的上下文切换次数要尽量减少，以提高系统的性能。

猫十二懿的回答

线程通常有五种状态：**创建，就绪，运行、阻塞和死亡状态。**



线程通常有五种状态：**创建，就绪，运行、阻塞和死亡**状态。

- 新建状态（New）：新创建了一个线程对象。
- 就绪状态（Runnable）：线程对象创建后，其他线程调用了该对象的start方法。该状态的线程位于可运行线程池中，变得可运行，等待获取 CPU 的使用权。
- 运行状态（Running）：就绪状态的线程获取了 CPU，执行程序代码。

- 阻塞状态（Blocked）：阻塞状态是线程因为某种原因放弃 CPU 使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。
- 死亡状态（Dead）：线程执行完了或者因异常退出了 run 方法，该线程结束生命周期。

其中阻塞的情况又分为三种：

(1)、**等待阻塞**：运行的线程执行wait方法，该线程会释放占用的所有资源，JVM 会把该线程放入“等待池”中。进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用 notify 或 notifyAll 方法才能被唤醒，wait 是 object 类的方法

(2)、**同步阻塞**：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则 JVM 会把该线程放入“锁池”中。

(3)、**其他阻塞**：运行的线程执行 sleep 或 join 方法，或者发出了 I/O 请求时，JVM 会把该线程置为阻塞状态。当 sleep 状态超时、join 等待线程终止或者超时、或者 I/O 处理完毕时，线程重新转入就绪状态。sleep 是 Thread 类的方法。

4、synchronized 关键字是什么，有什么作用?

官方解析

synchronized 是 Java 中的一个关键字，用于实现**线程同步**。具体来说，synchronized 用于**修饰方法或代码块**，使得**同一时刻只能有一个线程访问被修饰的代码**，其他线程需要等待当前线程执行完毕后才能访问。

synchronized 主要用于解决**多线程并发访问共享资源**时出现的**线程安全问题**。

如果多个线程同时访问一个共享资源，就会出现多个线程同时修改这个资源的情况，从而导致数据不一致等问题。而使用 synchronized 可以保证同一时刻只有一个线程访问该资源，从而避免了线程安全问题。

synchronized 的作用不仅限于线程同步，它还可以保证**可见性**和**有序性**，即保证在同一个锁上，**一个线程修改了共享变量的值之后，另一个线程能够立即看到修改后的值，并且在多个线程执行顺序上保证了一致性。**

需要注意的是，使用 synchronized 会带来一定的**性能损失**，因为**每次进入同步块时都需要获得锁**，这会增加线程的**等待时间**和**上下文切换的开销**。同时，如果**同步块的代码执行时间很短**，也会增加不必要的**性能开销**。因此，需要根据具体情况来判断是否需要使用 synchronized。

> 鱼皮补充：这题如果能再延伸回答一下 synchronized 的实现机制（锁升级），会更好

鱼友的精彩回答

猿二哈的回答

synchronize 关键字的作用

1. synchronize 是一个解决**并发安全**的的修饰关键字
2. synchronize 可以保证操作的**原子性与可见性**
3. synchronize可以加在**方法**上，表示在多线程的情况下，只有一个线程可以访问这个方法
4. 如果一个对象有多个synchronize 方法，一个线程访问了这个对象的一个 synchronize 方法，其他线程就不能访问这个对象的任何一个 synchronize 方法
5. 多个不同实例对象的 synchronize 方法之间没有关系

6. synchronize 可以加在**代码块**上，在多线程的情况下，如果一个线程访问了这个代码块，其他线程就无法访问这个对象的任何一个 synchronize 方法
7. synchronize 锁 class，表示这个 synchronize 方法对这个类的所有实例都有效，类 static 的类全局效果

关键词：**并发安全，原子性，可见性，加方法，加代码块，加 class**

CodeJuzi 的回答

synchronized 是 Java 中的一个**关键字**，用来实现**同步锁**。

它可以修饰**方法**或**代码块**，**保证同一时刻只有一个线程可以执行被修饰的部分**

synchronized 的作用有以下几点：

- 保证数据的**原子性**和**可见性**，防止多线程操作导致的数据不一致。
- **防止指令重排序**，保证代码执行的顺序和预期一致。
- **实现线程间的通信**，通过 wait 和 notify 方法来实现线程的等待和唤醒。

synchronized的使用方式有以下几种：

- 修饰非静态方法，锁对象是当前实例对象
- 修饰静态方法，锁对象是当前类对象
- 修饰代码块，锁对象是括号里指定的对象

synchronized 和其他同步工具（如 ReentrantLock）相比，有以下优缺点：

- 优点：简单易用，不需要手动释放锁。
- 缺点：不灵活，不能设置超时时间、中断等；效率低，每次都要进入内核态获取锁；不可重入，同一个线程在获取到锁后还要再次获取锁会造成死锁

回家养猪的回答

说一说自己对于 synchronized 关键字的了解

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

在 Java 早期版本中， synchronized 属于**重量级锁**，效率低下，因为**锁监视器（monitor）是依赖于底层的操作系统来实现的**。如果要**挂起或者唤醒**一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从**用户态转换到内核态**，这个状态之间的转换需要相对比较长的时间，这也是为什么早期的 synchronized 效率低的原因。

Java 官方对从 JVM 层面对 synchronized 较大优化，如**自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁**等技术来减少锁操作的开销。所以现在的 synchronized 锁效率也优化得很不错了。

说说自己是怎么使用 synchronized 关键字

修饰非静态方法, 锁的是 this(当前对象), 也就是一个对象用一把锁

修饰静态方法, 锁的是 类.class , 也就是类的所有对象公用一把锁

修饰代码块

尽量不要使用 synchronized(String a) 因为 JVM 中，字符串常量池具有缓存功能

synchronized原理

synchronized 同步语句块的实现使用的是 **monitorenter** 和 **monitorexit** 指令，其中 monitorenter 指令指向同步代码块的开始位置, monitorexit 指令则指明同步代码块的结束位置

当多个线程同时访问该方法，那么这些线程会先被放进**对象的锁池**，此时线程处于 **blocking** 状态

当一个线程获取到了实例对象的**监视器（monitor）锁**，那么就可以进入 **running** 状态，执行方法，此时 **lock record** 中的 **owner** 设置为当前线程，count 加 1 表示当前对象锁被一个线程获取

当 running 状态的线程调用**wait()方法**，那么当前线程释放 monitor 对象，进入 waiting 状态, lock record 中的 owner 变为 null，count 减 1，同时线程进入等待池，直到有线程调用 **notify()** 方法唤醒该线程，则该线程重新获取 monitor 对象进入 owner

如果当前线程执行完毕，那么也释放 monitor 对象，进入 waiting 状态，lock record 中的 owner 变为null，count 减 1

JDK1.6 之后的 synchronized 底层做了哪些优化?

java 的线程模型是 1 对 1 的, 加锁需要调用操作系统的底层原语 mutex, 所以每次切换线程都需要操作系统切换到内核态, 开销很大. 这也是之前 synchronized的问题所在, jdk1.6 对其进行了优化, 从**无锁到偏向锁到轻量级锁到重量级锁** 自旋锁就不需要阻塞, 也就不需要操作系统切换为内核态去做, 所以短时间的自旋开销是比较低的.

JDK 1.6 引入了**偏向锁和轻量级锁**，从而让锁拥有了四个状态：**无锁状态**（unlocked）、**偏向锁状态**（biasble）、**轻量级锁状态**（lightweight locked）**和重量级锁状态**（inflated）。

虚拟机对象头里锁标志位, 就记录了这4中状态.

偏向锁

大多数时候是不存在锁竞争的，常常是一个线程多次获得同一个锁，因此如果每次都要竞争锁会增大很多没有必要付出的代价，为了降低获取锁的代价，才引入的**偏向锁**。

当锁对象第一次被线程获得的时候，使用 **CAS** 操作将线程 ID 记录到对象头的 **MarkWord** 中，这个线程**以后每次进入这个锁相关的同步块就不需要再进行任何同步操作**。

当有**另外一个线程去尝试获取这个锁对象**时，**偏向状态就宣告结束**，此时撤销偏向后恢复到**未锁定状态**或者**轻量级锁**状态。

轻量级锁

轻量级锁考虑的是**竞争锁对象的线程不多**，而且**线程持有锁的时间也不长**的情景。因为**阻塞线程需要 CPU 从用户态转到内核态，代价较大**，如果刚刚阻塞不久这个锁就被释放了，那这个代价就有点得不偿失了，因此这个时候就干脆不阻塞这个线程，让它自旋这等待锁释放。

轻量级锁是相对于传统的重量级锁而言，它使用**自旋 + CAS 操作**来**避免重量级锁使用互斥量的开销**。

长时间的自旋会使 CPU 一直空转, 浪费 CPU, 所以这里自旋是**适应性自旋**, 自旋时间由**上一个线程自旋的时间**决定的.

线程自旋的次数到了阈值, 另外一个线程还没释放锁, 那么就膨胀为重量级锁。

如果**有两条以上的线程争用同一个锁**，那轻量级锁就不再有效，**要膨胀为重量级锁**。

锁消除

锁消除是指**对于被检测出不可能存在竞争的共享数据的锁**进行消除。

锁消除主要是通过**逃逸分析**来支持，如果堆上的共享数据不可能逃逸出去被其它线程访问到，那么就可以把它们当成私有数据对待，也就可以将它们的锁进行消除。

锁粗化

如果**一系列的连续操作都对同一个对象反复加锁和解锁**，频繁的加锁操作就会导致**性能损耗**。

比如连续使用 StringBuffer 的 append() 方法就属于这类情况。如果虚拟机探测到**由这样的一串零碎的操作都对同一个对象加锁**，将会**把加锁的范围扩展（粗化）到整个操作序列的外部**, 这样**只需要加锁一次**就可以了。

系统设计

1、如何设计一个点赞系统?

官方解析

设计一个点赞系统可以分为以下几个步骤：

- **确定需求**：需要明确**点赞的对象是什么**，是否需要**计数**等信息，同时需要考虑**点赞的业务场景**，如**用户点赞、文章点赞**等。
- **数据库设计**：需要设计**点赞相关的数据表**，可以包含**点赞者 ID、被点赞对象 ID、点赞时间**等字段。
- **接口设计**：需要**设计点赞相关的接口**，包括**点赞、取消点赞、查询点赞数**等操作。
- **业务逻辑实现**：在接口中实现点赞相关的**业务逻辑**，包括**判断点赞状态、更新点赞数、更新点赞状态**等操作。
- **安全性考虑**：需要考虑并发访问的情况，可以使用**分布式锁**来保证**数据一致性**和**安全性**。
- **性能优化**：如果点赞系统的访问量很高，可以使用缓存来提高性能，比如使用 **Redis 来缓存点赞数**等信息。
- **监控和日志**：需要对点赞系统进行监控和日志记录，以便及时发现和排查问题。

总之，设计一个点赞系统需要综合考虑**需求、数据库设计、接口设计、业务逻辑实现、安全性、性能优化**等方面，同时需要不断优化和完善。

> 鱼皮补充：题解中的最后一句话是很关键的，大家在回答时，首先要**明确实际的业务场景**，比如这个系统**同时有多少人使用、日均点赞量多少**等，再去考虑用 MySQL、还是 Redis、甚至是 MQ 来实现。

鱼友的精彩回答

Ming 的回答

如何设计一个点赞系统?

从需求上分析

- **点赞类别**：帖子、评论、or 其他
- 点赞限制

    - 仅登录用户还是游客都可以点赞?
    - 可以无限点赞还是每个用户仅限点1次?
- 点赞是否通知用户
    - 通知频率：每次点赞都通知 1 次，还是满 5 次合并通知 1 次
    - 通知样式：点赞帖子和点赞评论，通知的内容要怎么区分显示

从数据库设计上

- 点赞数量：是单独一张表？还是在帖子表直接加一个字段?
    - （前者的好处是，这样如果例如有其他类别的内容需要加点赞功能，就不用再修改数据库，后者的好处是后台统计功能方便查数据，不需要查两张表）
- 点赞记录表：记录点赞操作记录，哪个用户点赞了哪个帖子

从性能考虑上

- 点赞是个高频操作，肯定不能每次都直接操作数据库，需要加一层缓存
- 是单独独立一个key，如 {post_id}_like_num 记录
- 还是随着跟其他数量字段，一起构造一个key 如 {post_info}_num 一个hash字段
- 这里的设计会影响如何保持数据库和缓存一致性的问题

一致性考虑

- 看是否需要强一致性，如果不需要的话，就不需要加锁，这样可以减轻实现负担和提高一定的系统性能

就大概以上是我觉得比较重要，需要考虑的地方，然后编码实现上其实大差不差，主要还是要根据**需求以及后续可拓展性**去考虑

2、如何在 10 亿个数据中找到最大的 1 万个？（提示：最小堆）

官方解析

在 10 亿个数据中找到最大的 1 万个，可以使用最小堆（Min-Heap）算法来实现。

最小堆是一种特殊的二叉树结构，每个节点的值都小于或等于其左右子节点的值。在使用最小堆来查找最大的 1 万个数据时，可以先创建一个大小为 1 万的最小堆，然后将 10 亿个数据逐个加入堆中。当堆的大小超过了 1 万时，将堆顶元素（即最小值）弹出，再将当前元素加入堆中。当遍历完所有数据后，堆中剩余的 1 万个元素就是最大的 1 万个数据。

这种方法的时间复杂度为 O(n log k)，其中 n 是数据总数，k 是要查找的元素数量。因为要维护一个大小为 1 万的最小堆，所以空间复杂度也为 O(k)。

以下是一个示例代码：

```
import heapq

def find_largest_10000(data):
    heap = []
    for d in data:
        if len(heap) < 10000:
            heapq.heappush(heap, d)
        else:
            heapq.heappushpop(heap, d)
    return heap
```

在这个示例代码中，我们使用了 Python 的内置模块 heapq 来实现最小堆。

函数 find_largest_10000 接受一个包含 10 亿个数据的列表，然后返回最大的 1 万个数据。这个函数使用了一个大小为 1 万的最小堆来维护当前找到的最大的 1 万个数据。对于每个数据，如果堆的大小小于 1 万，则直接将数据加入堆中；否则，先将数据加入堆中，再弹出堆顶元素，保证堆的大小不超过 1 万。最后，函数返回堆中剩余的 1 万个元素，这些元素就是最大的 1 万个数据。

鱼友的精彩回答

Gundam的回答

使用最小堆来解决:

首先，我们可以把这 10 亿个数据划分成多个小数据块，每个小数据块包含 1 万个数据。然后，我们可以遍历这些小数据块，对于每个小数据块，我们可以构建一个大小为 1 万的最小堆。当遍历到一个新的数据时，我们可以把它与最小堆的堆顶元素进行比较。如果这个数据比堆顶元素大，那么我们可以把堆顶元素删除，并把这个数据插入到最小堆中。这样，当我们遍历完所有的小数据块之后，最小堆中存储的就是最大的 1 万个数据了。

具体实现过程如下：

1. 初始化一个大小为 1 万的最小堆，把第一个小数据块的数据插入到堆中。
2. 读取下一个小数据块的数据，把这些数据依次与最小堆的堆顶元素进行比较。

- 如果数据比堆顶元素大，那么删除堆顶元素，并把这个数据插入到堆中。
- 如果数据比堆顶元素小，那么忽略这个数据。

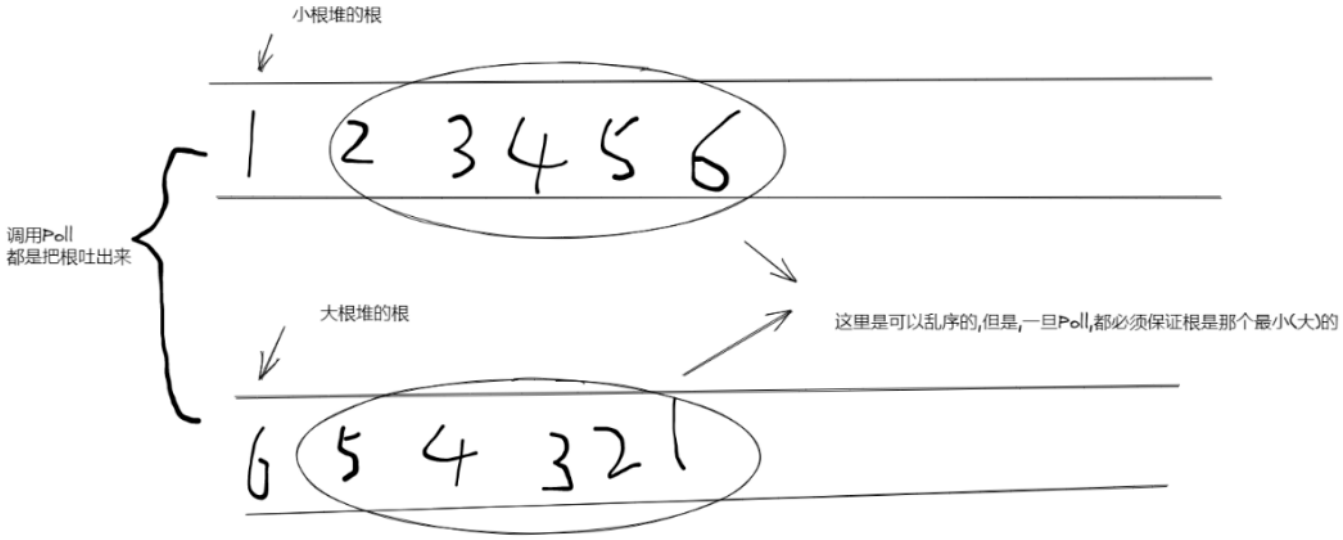
1. 重复步骤 2 直到所有的小数据块都被遍历完。
2. 最终，最小堆中存储的就是最大的 1 万个数据。

这种方法的时间复杂度是 O(NlogK)，其中 N 是数据总数，K 是需要找到的最大的数据数量。因为最小堆的大小是固定的，所以时间复杂度与数据总数 N 没有关系，只与需要找到的数据数量 K 有关系。因此，这种方法可以快速找到最大的 1 万个数据，而且可以处理非常大的数据集。

维萨斯的回答

一句话

大根堆找 TOP 小
小根堆找 TOP 大

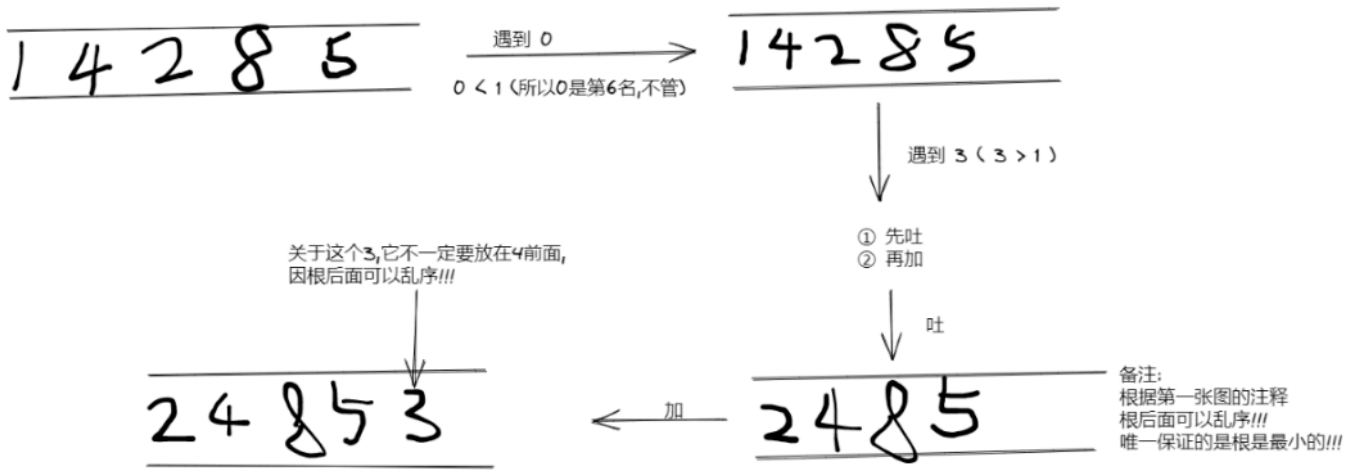


看着图再思考答案的合理性就简单了

- TOP 大 -> 小根堆
- 一边看图一边理解思路
- 先把前 1 w 个推进小根堆
- 然后
    - 具体思考过程:
    - 因为小根堆是: 根最小,那么说明,根后面的都比大,根就是那个第1W名
- 所以,处理后面的数字,当遇到比根还小的 (说明他怎么都是10001名) -> 不管
- 遇到比根大的 (说明他可以抢占第 1 W 名甚至第1名) -> 弹出根,然后把那个新的数字加进来
- 最后小根堆中的那 1 W 个就是 TOP10000 了

给个图最简单了 (如图为 TOP5)

备注: 下面的模拟是我用 IDEA 使用 PriorityQueue 进行 Debug 的,具体根后面是否可以乱序,我不知道其它语言怎么样,但是 JAVA 的优先队列是可以乱序的



JIA 的回答

最容易想到的方法是将数据全部排序，然后在排序后的集合中进行查找，最快的排序算法的时间复杂度一般为 O（nlogn），如快速排序。但是在 32 位的机器上，每个 float 类型占 4 个字节，1 亿个浮点数就要占用 400 MB 的存储空间，对于一些可用内存小于 400 M 的计算机而言，很显然是不能一次将全部数据读入内存进行排序的。其实即使内存能够满足要求（我机器内存都是 8 GB），该方法也并不高效，因为题目的目的是寻找出最大的 10000 个数即可，而排序却是将所有的元素都排序了，做了很多的无用功。

第二种方法为**局部淘汰法**，该方法与排序方法类似，用一个容器保存前 10000 个数，然后将剩余的所有数字——与容器内的最小数字相比，如果所有后续的元素都比容器内的 10000 个数还小，那么容器内这个 10000 个数就是最大 10000 个数。如果某一后续元素比容器内最小数字大，则删掉容器内最小元素，并将该元素插入容器，最后遍历完这 1 亿个数，得到的结果容器中保存的数即为最终结果了。此时的时间复杂度为 O（n+m^2），其中 m 为容器的大小，即 10000。

第三种方法是**分治法**，将 1 亿个数据分成 100 份，每份 100 万个数据，找到每份数据中最大的 10000 个，最后在剩下的 100*10000 个数据里面找出最大的 10000 个。如果 100 万数据选择足够理想，那么可以过滤掉1亿数据里面 99% 的数据。100万个数据里面查找最大的 10000 个数据的方法如下：用快速排序的方法，将数据分为 2堆，如果大的那堆个数 N 大于 10000 个，继续对大堆快速排序一次分成 2 堆，如果大的那堆个数 N 大于 10000 个，继续对大堆快速排序一次分成 2 堆，如果大堆个数 N小于 10000 个，就在小的那堆里面快速排序一次，找第 10000-n 大的数字；递归以上过程，就可以找到第 1 w 大的数。参考上面的找出第 1 w 大数字，就可以类似的方法找到前 10000 大数字了。此种方法需要每次的内存空间为 10^6*4=4MB，一共需要 101 次这样的比较。

第四种方法是 **Hash 法**。如果这 1 亿个书里面有很多重复的数，先通过 Hash 法，把这 1 亿个数字去重复，这样如果重复率很高的话，会减少很大的内存用量，从而缩小运算空间，然后通过分治法或最小堆法查找最大的 10000 个数。

第五种方法采用**最小堆**。首先读入前 10000 个数来创建大小为 10000 的最小堆，建堆的时间复杂度为 O（mlogm）（m 为数组的大小即为 10000），然后遍历后续的数字，并于堆顶（最小）数字进行比较。如果比最小的数小，则继续读取后续数字；如果比堆顶数字大，则替换堆顶元素并重新调整堆为最小堆。整个过程直至 1 亿个数全部遍历完为止。然后按照中序遍历的方式输出当前堆中的所有 10000 个数字。该算法的时间复杂度为 O（nmlogm），空间复杂度是 10000（常数）。

爱吃鱼蛋 的回答

方法

在10亿个数据中找到最大的 1 万个，可以使用以下算法：

1. **Top-K 算法**：Top-K 算法是一种常用的数据挖掘算法，可以在大数据集中找到最大或最小的 K 个元素；
2. **MapReduce 算法**：MapReduce 是一种分布式计算框架，可以将大数据集分成多个小数据集进行处理，然后将结果进行合并。可以使用 MapReduce 实现 Top-K 算法，将数据分成多个部分进行处理，然后将结果进行合并，最后得到最大的 1 万个数据；
3. **快速选择算法**：快速选择算法是一种基于快速排序的算法，可以快速找到第 K 大的数。具体实现可以使用分治法，将数据分成多个小数据集进行处理，然后递归地进行比较和排序，最后找到第 K 大的数。

其中 Top-K 算法通常使用的是最小堆实现。最小堆是一种基于树的数据结构，它的根节点是最小的元素，每个节点的值都小于或等于其子节点的值。在 Top-K 算法中，我们可以使用最小堆来维护当前找到的最大的 K 个元素。具体实现方式是

- 将数据集中的前 K 个元素构建成一个最小堆；
- 遍历剩余的数据：
  - 如果当前数据比堆顶元素小，跳过；
  - 如果当前数据比堆顶元素大，则将堆顶元素替换成当前元素，并重新调整最小堆。
- 重复这个过程，直到遍历完所有数据，最终得到的就是最大的 K 个元素。

实现思路

在Java中，我们可以使用Java 内置的 **PriorityQueue** 优先队列来实现最小堆。具体实现如下：

- 创建一个大小为 k 的 PriorityQueue 对象 queue，用来存储当前找到的最大的 k 个数；
- 遍历 nums 数组，对于每个元素：
  - 如果 queue 的大小小于 k，将其加入 queue 中；
  - 如果 queue 的大小等于 k，比较当前元素与 queue 的堆顶元素的大小，如果当前元素大于堆顶元素，则将堆顶元素弹出，将当前元素加入 queue 中；
- 遍历完成后，queue 中存储的就是最大的 k 个元素；
- 将 queue 中的元素依次弹出，存入结果数组 result 中，返回结果数组。

```
public class TopK {
    public static void main(String[] args) {
        // 定义模拟数据
        int[] nums = {3, 5, 1, 7, 8, 2, 9, 4, 6};
        // 定义需要找的top k
        int k = 3;
        // 拿到top k数组
        int[] result = topK(nums, k);
        for (int i = 0; i < result.length; i++) {
            System.out.print(result[i] + " ");
        }
    }
```

```
    /**
     * top-k算法(最小堆实现)
     * @author: yudan
     */
    public static int[] topK(int[] nums, int k) {
        // 定义优先队列充当最小堆
        PriorityQueue<Integer> queue = new PriorityQueue<>(k);
        // 遍历数组
        for (int num : nums) {
            if (queue.size() < k) {
                // 保证堆中永远都有且只有k个元素
                queue.offer(num);
            } else {
                // 与堆顶元素比较进行更换
                if (num > queue.peek()) {
                    queue.poll();
                    queue.offer(num);
                }
            }
        }
        // 出队将结果放入数组中返回
        int[] result = new int[k];
        for (int i = 0; i < k; i++) {
            result[i] = queue.poll();
        }
        return result;
    }
}
```

优化思路

当然，上面的情况只是最简单的情况，对于10亿数据中找最大的1万个，虽然也能用上面的方式，但是很容易造成处理时间过长甚至OOM问题，因此针对这些问题可以采用以下方式进行优化：

- 对数据进行**分块处理**，每次只处理一部分数据，这样可以避免一次性将所有数据读入内存中；
- 使用**线程池**处理分块数据，采取并行的方式提高处理的速度，但需要考虑并发问题；
- 使用**外部排序**等方法将数据分块排序，然后再合并排序后的结果，这样可以避免一次性对所有数据进行排序；
- **调整JVM的堆内存大小**，增大堆内存的大小可以提高处理大规模数据的能力，但是也会增加系统的负担和风险；
- 使用**分布式计算框架**如Hadoop、Spark等来处理大规模数据，这样可以将数据分布在多台计算机上进行处理，大大提高处理能力和效率。

3、有几台机器存储着几亿的淘宝搜索日志，假设你只有一台 2g 的电脑，如何选出搜索热度最高的十个关键词?

官方解析

对于 **top k 类文本问题**，通常比较好的方案是**【分治+hash/trie 树+小顶堆】**，即先按照 hash 方法把数据集分解成多个**小数据集**，然后使用 **trie 树**或者 **hash** 统计每个**小数据集**中的**词频**，随后用**小顶堆**求出**每个数据集中出现频率最高的前 K 个数**，最后在**所有 top K 中求出最终的 top K**。

拆分成 n 多个文件：以**首字母区分**，不同首字母放在不同文件，如果某个文件长度仍过长，继续按照次首字母进行拆分。这样一来，每个文件的每个数据长度基本相同且首字母相同，就能保证数据被独立的分为了 n 个文件，且各个文件中不存在关键词的交集。

分别词频统计：对于每个文件，使用 **hash** 或者 **Trie** 树进行进行**词频统计**

小顶堆排序：依次处理每个文件，并逐渐更新最大的十个词

鱼皮的评论

题解更倾向于以原生命令的方式去实现，其实还有很多现成的第三方库，比如 Redisson。这里要关注分布式锁可能出现的各种异常情况。

鱼友的精彩回答

会冒泡的可乐的回答

1.读取搜索日志数据

从几台机器上存储的搜索日志数据中，将数据读取到内存中。由于数据量较大，使用多线程可以提高数据处理的效率。具体的实现方案，可以将搜索日志数据按照固定大小分块，每个线程读取一个分块的数据进行处理，并将处理结果写入共享队列中，最后汇总所有处理结果得到最终的统计结果。

2.对关键词进行统计

将每一条搜索日志按照关键词进行分组，并统计每个关键词出现的次数。可以使用Counter 类来进行关键词统计。

3.选取搜索热度最高的十个关键词

将统计结果按照关键词出现次数进行降序排序，选取出现次数最高的前十个关键词，作为搜索热度最高的十个关键词。可以使用sorted函数来进行排序。

具体代码如下：

```
import threading
import queue
from collections import Counter

# 定义线程数和分块大小
NUM_THREADS = 4
BLOCK_SIZE = 1000000

# 定义共享队列
result_queue = queue.Queue()
```

```
# 定义线程函数
def process_data(block):
    keywords_counter = Counter()
    for line in block:
        keyword = line.strip().split(' ')[1]
        keywords_counter[keyword] += 1
    result_queue.put(keywords_counter)

# 读取搜索日志数据
# 其中search_log.txt是搜索日志数据文件的路径，可以根据实际情况进行修改
with open('search_log.txt', 'r') as f:
    # 分块读取数据
    while True:
        block = f.readlines(BLOCK_SIZE)
        if not block:
            break
        # 启动线程进行数据处理
        threads = []
        for i in range(NUM_THREADS):
            start_idx = i * (len(block) // NUM_THREADS)
            end_idx = (i + 1) * (len(block) // NUM_THREADS) if i != NUM_THREADS - 1
else len(block)
            t = threading.Thread(target=process_data, args=(block[start_idx:end_idx],))
            threads.append(t)
            t.start()
        # 等待所有线程处理完毕
        for t in threads:
            t.join()

# 汇总处理结果
final_counter = Counter()
while not result_queue.empty():
    counter = result_queue.get()
    final_counter += counter

# 选取搜索热度最高的十个关键词
top_keywords = final_counter.most_common(10)
```

heart000_1 的回答

假设每个搜索日志的大小为 100 字节，那么几亿条搜索日志的总大小约为 10 GB。由于只有一台 2G 的电脑，因此不能直接将所有搜索日志读入内存进行处理。我们可以采用外排序（External Sorting）的方法来解决这个问题。具体步骤如下：

1. 将所有搜索日志分成若干个大小相等的块，每个块的大小可以根据内存大小来确定，比如每个块的大小为 100 MB。

2. 对于每个块，使用哈希表（Hash Table）来统计每个关键词的搜索次数。由于哈希表的大小不能超过内存大小，因此需要对每个块都单独统计。当哈希表无法容纳更多的数据时，将哈希表中的数据写入一个临时文件。
3. 对于所有临时文件，使用归并排序（Merge Sort）的方法将它们合并成一个大的有序文件。在合并的过程中，统计每个关键词的搜索次数，并将其存入一个大小为 K 的最小堆中，其中 K 为需要选出的热度最高的关键词的个数。最小堆中维护的是搜索次数最大的 K 个关键词。
4. 当所有临时文件都被合并并处理完毕后，最小堆中剩余的 K 个关键词即为搜索热度最高的十个关键词。

需要注意的是，在进行哈希表统计和归并排序的过程中，需要使用外部存储器（如硬盘）来存储中间结果。因此，外排序算法的时间复杂度主要取决于磁盘读写的速度，而不是计算的速度。为了提高性能，可以使用多台机器来同时进行处理，从而加快处理速度。

JVM

1、请你介绍下 JVM 内存模型，分为哪些区域？各区域的作用是什么？

官方解析

JVM 内存模型分为以下几个区域：

1. 程序计数器（Program Counter Register）：每个线程都有自己的程序计数器，用于指示当前线程执行的字节码指令的行号，以便线程执行时能够回到正确的位置。
2. 虚拟机栈（JVM Stack）：也称为 Java 方法栈，用于存储方法执行时的局部变量表、操作数栈、动态链接、方法出口等信息。每个线程在执行一个方法时，都会为该方法分配一个栈帧，并将该栈帧压入虚拟机栈，当方法执行完毕后，虚拟机会将其出栈。
3. 本地方法栈（Native Method Stack）：与虚拟机栈类似，用于存储本地方法的执行信息。
4. 堆（Heap）：用于存储对象实例，是 JVM 中最大的一块内存区域。堆是被所有线程共享的，当创建一个新对象时，对象实例存储在堆中，堆中存储的对象实例都有一个标记用于标记对象是否存活。垃圾回收器会周期性地回收那些没有被标记为存活的对象。
5. 方法区（Method Area）：用于存储已经被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。方法区也是被所有线程共享的。
6. 运行时常量池（Runtime Constant Pool）：是方法区的一部分，用于存储编译期间生成的各种字面量和符号引用，这些内容在类加载后进入常量池中。

其中，程序计数器、虚拟机栈、本地方法栈是线程私有的，堆、方法区、运行时常量池是线程共享的。

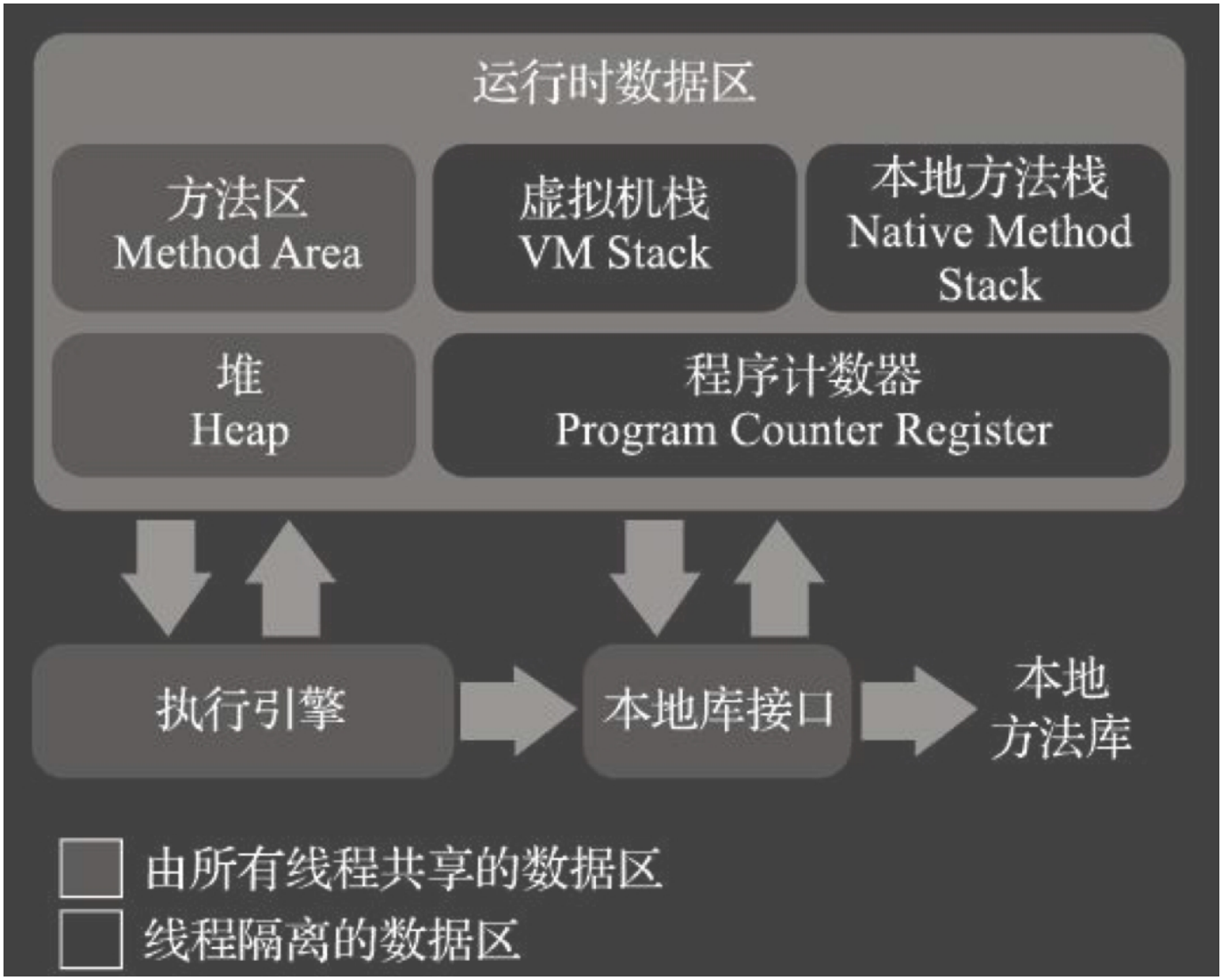
> 鱼皮补充：这题一般是 JVM 必考题，也是 JVM 学习的基础

鱼友的精彩回答

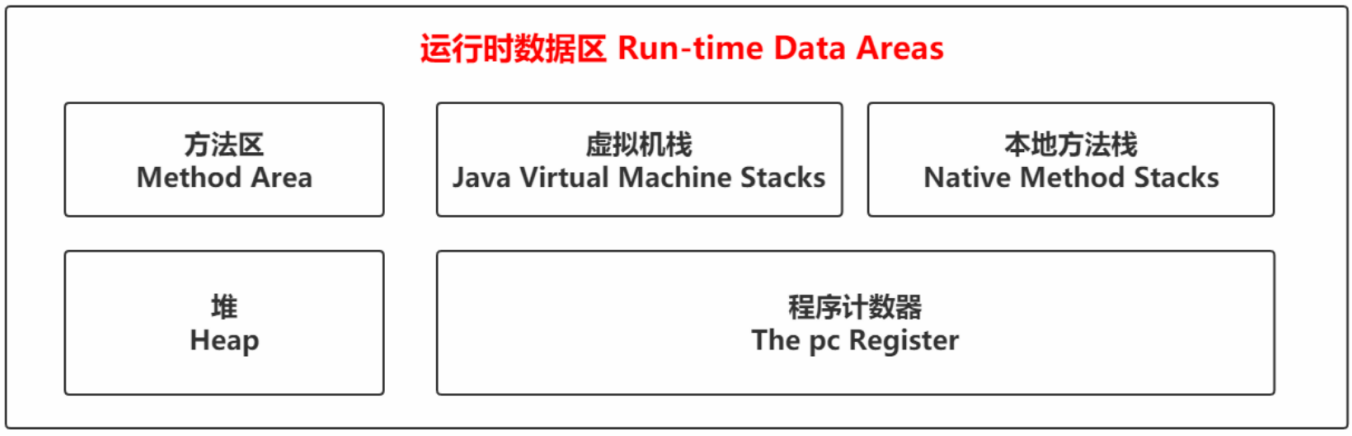
苏打饼干的回答

运行时数据区，包括程序计数器、Java 虚拟机栈、本地方法栈、Java 堆、方法区等。

区域	线程共享	说明	补充
程序计数器	×	记录当前线程所执行的字节码的型号指示器（**程序控制流的指示器**），分支、循环、跳转、异常处理、线程恢复等基础功能都依赖这个指示器完成，字节码解释器工作时就通过改变这个计数器的值选取下一条需要执行的字节码指令	每条线程都有一个独立的程序计数器，确保线程切换后能恢复到正确的执行位置 线程正在执行 Java方法，计数器记录字节码指令地址 线程正在执行 Native方法，计数器值为空(Undefined)
Java虚拟机栈	×	描述 Java方法执行的线程内存模型，每个方法被执行时，JVM会创建一个**栈帧**来存储局部变量表、操作数栈、动态连接、方法出口等信息，该栈帧的出/入栈代表着方法的调用/执行完毕	**局部变量表**存放基本数据类型、对象引用和 `returnAddress` 类型，这些以局部变量槽来作为单元来表示
本地方法栈	×	类似于Java虚拟机栈，面向的是 Native 本地方法(如JNI)	\
Java堆	√	唯一的目的就是存放对象实例	\
Java堆	√	唯一的目的就是存放对象实例	\
方法区	√	又称为"**非堆**"，用于存储已被虚拟机加载的类型信息、常量、静态变量、即时编译器变异后的代码缓存等数据，该区域的内存回收目标主要针对常量池的回收和对类型的卸载，条件相当苛刻导致回收效果较弱	**常量池表**：用于存放编译期生成的 各种字面量与符号引用 **运行时常量池**：方法区的一部分，在类加载后，编译期生成的各种字面量与符号引用会存放于此，其具有动态性，运行期间也可以将心的常量放入池中


运行时数据区
方法区
Method Area
虚拟机栈
VM Stack
本地方法栈
Native Method
Stack
堆
Heap
程序计数器
Program Counter Register
执行引擎
本地库接口
本地
方法库
由所有线程共享的数据区
线程隔离的数据区

我很忙的回答

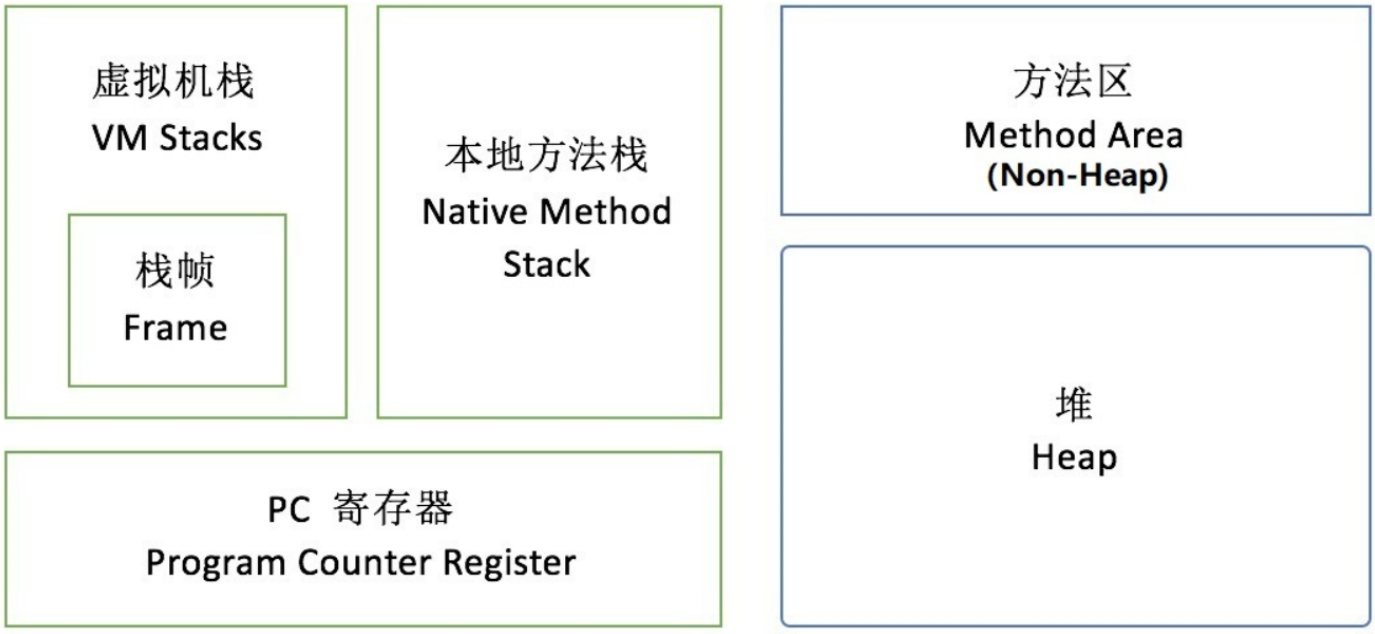

运行时数据区 Run-time Data Areas
方法区
Method Area
虚拟机栈
Java Virtual Machine Stacks
本地方法栈
Native Method Stacks
堆
Heap
程序计数器
The pc Register

> **值得说明的**
>
> ```
> JVM运行时数据区是一种规范，真正的实现
> 在JDK 8中就是Metaspace，在JDK6或7中就是Perm Space
> ```

分两块儿讲，线程共有，线程私有。
堆，方法区，直接内存，程序计数器，虚拟机栈，本地方法栈

1. 程序计数器：字节码解释器可以改变程序计数器来读取指令，控制代码的流程；多线程的话，能记录线程当前运行的位置（唯一一个不出现outofmemoryerror的数据区）
2. 虚拟机栈：为调用java方法服务的，每一个被线程执行的方法，为该栈中的栈帧，即每个方法对应一个栈帧。调用一个方法，就会向栈中压入一个栈帧；一个方法调用完成，就会把该栈帧从栈中弹出；若栈内存大小不允许动态扩容，可能会出现StackOverflowerror；如果栈内存大小可以动态扩容，则可能出现outofmemoryerror；
3. 本地方法栈：native方法通过这个实现
4. 堆：Java堆是Java虚拟机所管理内存中最大的一块，存放实例对象与数组；容易出现outofmemoryerror
5. 方法区：存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码数据。即永久代，在jdk1.8中不存在方法区了，被称为元空间（Metaspace）。原方法区被分成两部分；1：加载的类信息，2：运行时常量池；加载的类信息被保存在元数据区中，运行时常量池保存在堆中

HeiHei 的回答



堆：

- 堆是运行时数据区，所有类的实例和数组都是在堆上分配内存。它在 JVM 启动的时候被创建。对象所占的堆内存是由自动内存管理系统也就是垃圾收集器回收。

虚拟机栈： 每个线程运行时所需要的内存，称为虚拟机栈 每个栈由多个栈帧（Frame）组成，对应着每次方法调用时所占用的内存 每个线程只能有一个活动栈帧，对应着当前正在执行的那个方法

本地方法栈：

- 与虚拟机栈发挥的作用相似，相比于虚拟机栈为Java方法服务，本地方法栈为虚拟机使用的Native方法服务，执行每个本地方法的时候，都会创建一个栈帧用于存储局部变量表，操作数栈，动态链接，方法出口等信息。

程序计数器（PC寄存器）：

- 程序计数器区域是一块较小的区域，它用于存储线程的每个执行指令，每个线程都有自己的程序计数器

方法区（元空间）：

- 方法区是可提供各条线程共享的运行时内存区域。存储了每一个类的结构信息，例如运行时常量池（Runtime Constant Pool）、字段和方法数据、构造函数和普通方法的字节码内容，还包括一些在类、实例、接口初始化时用到的特殊方法。在JDK1.8之前的版本里，代表JVM的一块区域。在1.8版本以后，这块区域的名字改了，叫做“Matespace”

2、什么是双亲委派模式？有什么作用？

官方解析

双亲委派模式（Parent-Delegate Model）是 **Java 类加载器**（ClassLoader）在**加载类**时所采用的一种**设计模式**。这种模式的核心思想是：当一个**类加载器**收到**类加载请求**时，首先不会尝试自己加载这个类，而是将**请求委派给其父类加载器**。依次递归，直到**最顶层的启动类加载器**（Bootstrap ClassLoader）；**如果父类加载器无法加载该类，子类加载器才尝试自己去加载**。

双亲委派模式的作用主要有以下几点：

1. **避免类的重复加载**：通过委派给父类加载器加载类，可以确保同一个类不会被多个类加载器重复加载。这有助于节省内存资源，并确保类之间的互操作性。
2. **保护 Java 核心类库**：由于双亲委派模式的存在，**用户自定义的类加载器无法直接加载 Java 核心类库**（如java.lang.Object等）。这有助于确保 Java 核心类库的安全性，防止恶意代码篡改或破坏Java核心类。
3. **维护类加载器的层次结构**：双亲委派模式使得各级类加载器可以按照一定的层次结构来组织和管理。这有助于降低类加载器的复杂性，简化类加载过程。

双亲委派模式在 Java 类加载器中的应用是一种优秀的设计原则，它有助于确保类加载过程的稳定性、安全性和可维护性。

然而，在某些特殊场景下（如 OSGi、Java 热加载等），双亲委派模式可能无法满足需求，需要采用其他类加载策略。在这些场景下，开发者需要充分了解类加载机制，以避免产生意外的问题。

鱼友的精彩回答

Starry 的回答

双亲委派模式是一种类加载机制，它是基于一个简单的原则：除非有明确的需求，否则类加载器不会尝试去加载一个它找不到的类，而是把这个任务交给父类加载器来完成。

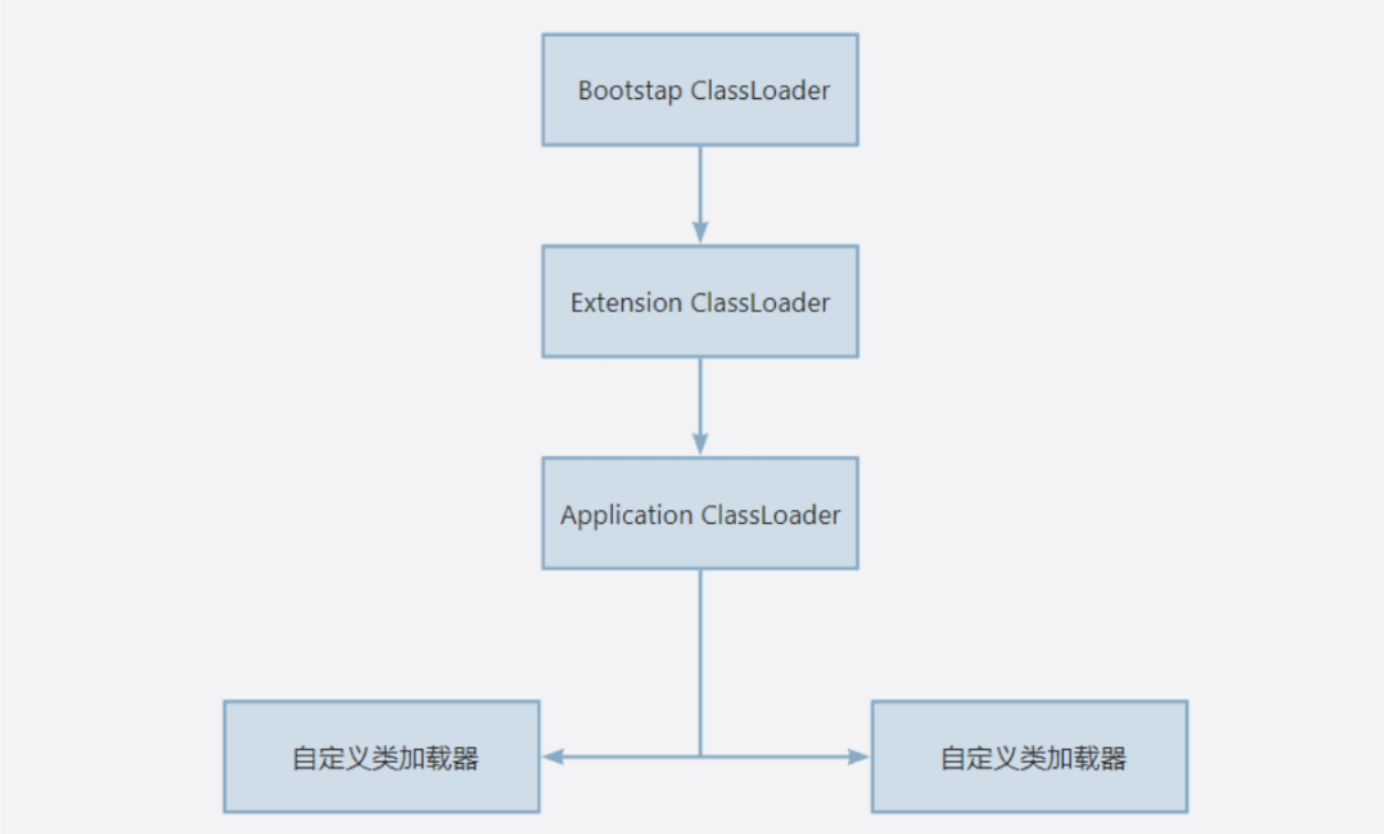
具体来说，当一个类加载器需要加载某个类时，它首先会委派给它的父类加载器去加载该类。如果父类加载器无法找到该类，则再将加载任务委派给它的父类加载器，直到最终委派到顶层的启动类加载器。如果顶层的启动类加载器仍然无法找到该类，则会抛出 ClassNotFoundException 异常。

这样做的好处是：防止内存中出现多份同样的字节码。

双亲委派模式的作用：

- 保证 JVM 中每个类只会被加载一次，避免重复加载。当一个类被加载后，它会被缓存在 JVM 的内存中，以便之后的使用。
- 如果不采用双亲委派模式，可能会导致同一个类被多次加载，从而浪费大量的内存资源。
- 此外，双亲委派模式还可以保证类的安全性，因为如果一个类是由用户自定义的类加载器加载的，那么它并不能访问由系统类加载器或其它更高级别的类加载器加载的类。对于一些核心类，例如 java.lang.Object、java.lang.String 等，它们都是由启动类加载器加载的，因此也可以保证它们的正确性和安全性。

林风的回答



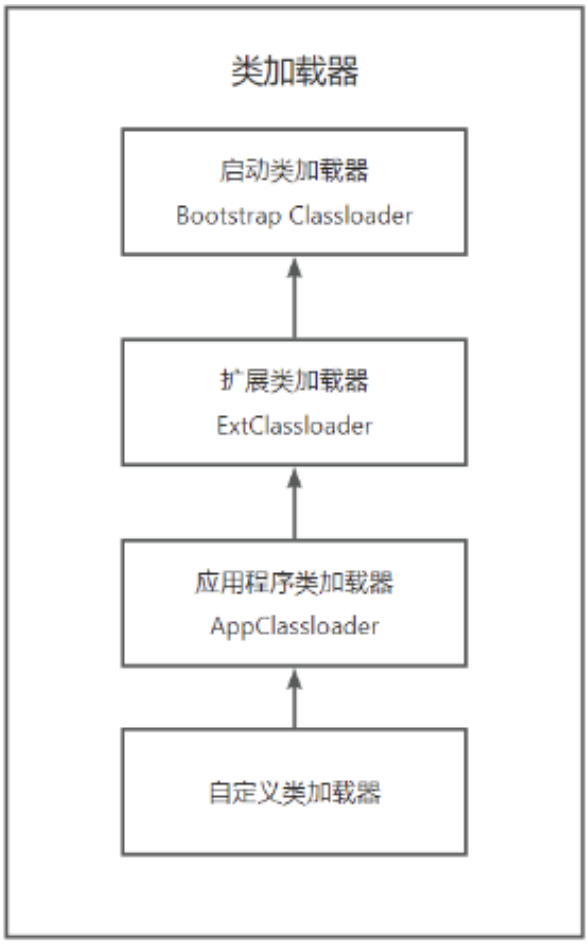
双亲委派机制是指当一个类加载器收到一个类加载请求时，该类加载器首先会把请求委派给父类加载器。每个类加载器都是如此，只有在父类加载器在自己的搜索范围内找不到指定类时，子类加载器才会尝试自己去加载。

双亲委派模型的好处 双亲委派模型保证了 Java 程序的稳定运行，可以避免类的重复加载，因为即使这两个类来源于同一个 Class 文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那么这两个类就必定不相等。

同时保证了 Java 的核心 API 不被篡改。 避免类的重复加载 保证Java核心API不被篡改

HeiHei 的回答

双亲委派：



类加载机制：
双亲委派模型顾名思义就是每一个类加载器（classLoader）收到一个类加载请求，都不会先自己去加载，**而是委托给自己的父类加载，如果父类加载器还存在它的父类加载器，则继续向上委托，直到达到最顶层的父类加载器，从最顶层的父类加载器开始进行类的加载，如果最顶层的类加载器找不到类的路径，又交给子类加载进行加载，依次进行递归，**通俗地说，如下面的例子：

- 假设用户刚刚写的 Tes t类想进行加载，这个时候首先会发送给应用程序类加载器 AppCloassLoader；
- 然后 AppClassLoader 并不会直接去加载 Test 类，而是会委派于父类加载器完成此操作，也就是 ExtClassLoader；
- ExtClassLoader 同样也不会直接去加载 Test 类，而是会继续委派于父类加载器完成，也就是 BootstrapClassLoader；
- BootstrapClassLoader 这个时候已经到顶层了，没有父类加载器了，所以 BootstrapClassLoader 会在 jdk/lib 目录下去搜索是否存在，因为这里是用户自己写的 Test 类，是不会存在于 jdk 下的，所以这个时候会给子类加载器一个反馈。
- ExtClassLoader 收到父类加载器发送的反馈，知道了父类加载器并没有找到对应的类，爸爸靠不住，就只能自己来加载了，结果显而易见，自己也不行，没办法，只能给更下面的子类加载器了。
- AppClassLoader 收到父类加载器的反馈，顿时明白，原来爸爸虽然是爸爸，但是他终究不能管儿子的私事，所以这时候，AppClassLoader 就自己尝试去加载。

作用：

1. 避免重复加载类：如果一个类已经被某个类加载器加载过了，那么其他类加载器就不需要再次加载该类，因为类的定义只需要加载一次就可以了。

2. 确保类的唯一性：通过双亲委派模式，可以保证不同的类加载器加载同一个类时，得到的都是同一个 Class 对象，从而保证了类的唯一性。
3. 安全性：通过双亲委派模式，可以保证核心 Java API 不会被非法替换，从而提高了 Java 应用的安全性。

3、常见的垃圾回收算法有几种类型？他们对应的优缺点是什么？

官方解析

常见的垃圾回收算法有以下几种类型：

1. **标记-清除算法（Mark-Sweep）**：分为标记和清除两个阶段。标记阶段遍历所有活动对象并打上标记，清除阶段将未被标记的对象删除。优点是不需要连续内存空间，缺点是清除后可能会产生内存碎片。
2. **复制算法（Copying）**：将可用内存分为两块，只使用其中一块，当这一块满了后，将存活对象复制到另一块未被使用的空间，然后清除使用的那块。优点是简单高效，没有内存碎片问题，缺点是需要额外的空间来存储复制后的对象。
3. **标记-整理算法（Mark-Compact）**：在标记阶段与标记-清除算法类似，但在清除阶段将存活对象整理到内存的一端，然后清除边界以外的所有对象。优点是不会产生内存碎片，缺点是比较慢。
4. **分代收集算法（Generational）**：根据对象存活的时间将内存分为几个区域，每个区域采用不同的回收策略。一般将新生代分为 Eden 区和两个 Survivor 区，采用复制算法回收；将老年代采用标记-清除或标记-整理算法回收。优点是提高了回收效率，缺点是需要额外的维护成本。

这些算法各有优缺点，适用于不同的场景。标记-清除算法简单，但可能会产生内存碎片；复制算法适用于短时间内产生大量垃圾的场景，但需要额外的空间存储复制后的对象；标记-整理算法不会产生内存碎片，但比较慢；分代收集算法提高了回收效率，但需要额外的维护成本。

对于一个应用程序，选择适合的垃圾回收算法需要综合考虑应用场景、内存需求、性能要求等多个因素，以便达到最佳的效果。

鱼友的精彩回答

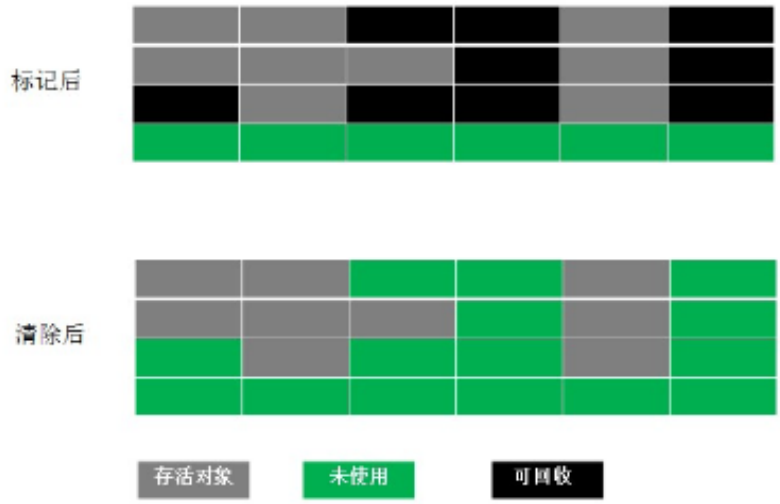
HeiHei 的回答

常见的垃圾回收算法：

- **Mark-Sweep（标记-清除）算法**

  标记-清除算法分为两个阶段：标记阶段和清除阶段。标记阶段的任务是首先通过可达性分析，标记出所有需要回收的对象，清除阶段就是回收被标记的对象所占用的空间。

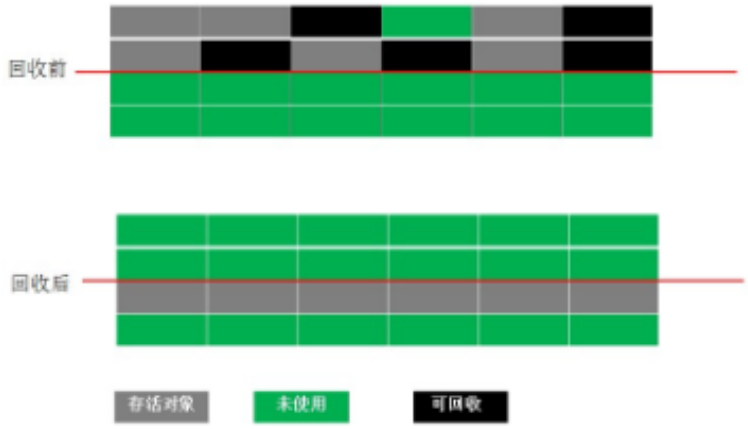
  优点：速度较快；缺点：内存不连续，会造成内存碎片，碎片太多可能会导致后续过程中需要为大对象分配空间时无法找到足够的空间而提前触发新的一次垃圾收集动作。



- **Coping（复制）算法**

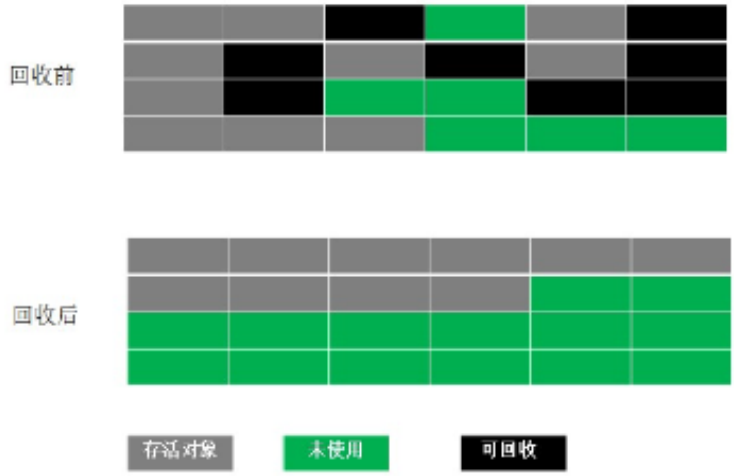
  为了解决Mark-Sweep算法的缺陷，他将可用内存按照容量划分为大小相等的两块，每次只使用一块。当这一块的内存用完了，就将还存活的对象复制到另一块上面，然后再把已使用的内存空间一次清理掉，这样一来就不容易出现内存碎片的问题。

  这种算法虽然实现简单，运行高效且不容易产生内存碎片，但是却对内存空间的使用做出了高昂的代价，因为能够使用的内存缩减到原来的一半。很显然，Copying算法的效率跟存活对象的数目多少有很大的关系，如果存活对象很多，那么Copying算法的效率将会大大降低。



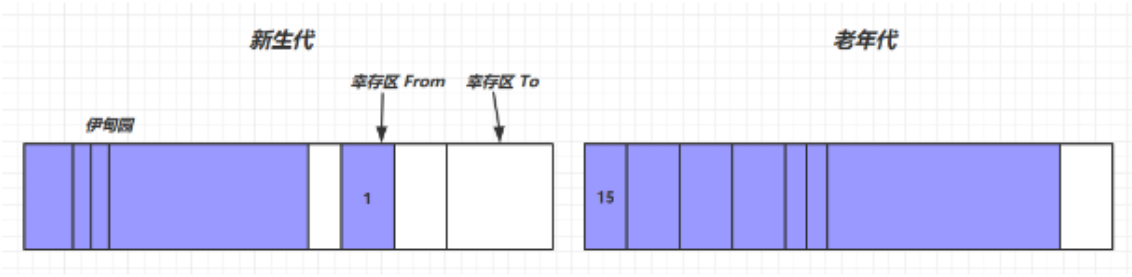
- **Mark-Compact（标记-整理）算法（压缩算法）**

  为了解决Copying算法的缺陷，充分利用内存空间，提出了Mark-Compact算法。该算法标记阶段和Mark-Sweep一样，但是在完成标记之后，他不是直接清理可回收对象，而是将存活对象都向一端移动，然后清理掉端边界以外的内存



- **Generation Collection（分代收集）算法**

  分代收集算法是目前大部分JVM的垃圾回收器采用的算法。他的核心思想是根据对象存活的生命周期将内存划分为若干个不同的区域。一般情况下将堆区**划分为老年代（Tenured Generation）和新生代（Young Generation）**，老年代的特点是每次垃圾回收时，只有少数对象需要被回收，而新生代的特点是每次垃圾回收时都有大量的对象需要被回收
  目前大部分垃圾回收器对于**新生代都采取复制算法**，因为新生代中每次垃圾回收都要回收大部分对象，也就是说需要复制的操作次数较少，但实际上不是按照1：1的比例来划分新生代的空间，一般来说是将新生代划分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden空间和其中的一块Survivor空间。当新生代的Eden Space和From Space空间不足时就会发生一次GC，进行GC后，Eden Space和From Space区的存活对象会被挪到To Space，然后将Eden Space和From Space进行清理。如果To Space无法足够存储某个对象，则将这个对象存储到老生代。在进行GC后，使用的便是Eden Space和To Space了，如此反复循环。当对象在Survivor区躲过一次GC后，其年龄就会+1。默认情况下年龄到达15的对象会被移到老年代中。
  而由于老年代的特点是每次回收都只回收少量对象，一般使用的是**标记-整理算法（压缩法）**，当老年代空间不足，会先尝试触发 minor gc，如果之后空间仍不足，那么触发 full gc，STW的时间更长



Ming 的回答

常见的垃圾回收算法

标记-清除算法、复制算法、标记-整理算法、分代收集算法

标记-**清除算法**

标记—清除算法包括两个阶段：“标记”和“清除”。 标记阶段：确定所有要回收的对象，并做标记。 清除阶段：将标记阶段确定不可用的对象清除。

优点：

算法简单明了，实现容易

缺点：

标记和清除的效率都不高。 会产生大量的碎片，而导致频繁的回收。

复制算法

内存分成大小相等的两块，每次使用其中一块，当垃圾回收的时候， 把存活的对象复制到另一块上，然后把这块内存整个清理掉。

缺点：

需要浪费额外的内存作为复制区。 当存活率较高时，复制算法效率会下降。

标记-整理算法

标记—整理算法不是把存活对象复制到另一块内存，而是把存活对象往内存的一端移动，然后直接回收边界以外的内存。

缺点： 算法复杂度大，执行步骤较多

分代收集算法

目前大部分 JVM 的垃圾收集器采用的算法。根据对象存活的生命周期将内存划分为若干个不同的区域。一般情况下将堆区划分为新生代（ Young Generation 和老年代（ Tenured Generation ），永久代（ Permanet Generation ）。

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

Young：存放新创建的对象，对象生命周期非常短，几乎用完可以立即回收，也叫 Eden 区。

Tenured： young 区多次回收后存活下来的对象将被移到 tenured 区，也叫 old 区。

Perm：永久带，主要存加载的类信息，生命周期长，几乎不会被回收。

缺点： 算法复杂度大，执行步骤较多。

Go 语言的垃圾回收算法

因为我学的是 Go 语言，就说说 Go 语言的垃圾回收算法

Go语言具有自己的垃圾收集器，并在多个版本不断演进优化：

- V1.3及之前：标记清除算法
- V1.5：三色并发标记法
- V1.8：混合写屏障机制

1. **标记清除算法（Mark Sweep Algorithm）**，是常见的垃圾收集算法之一，包括标记（Mark）和清除（Sweep）两个阶段：

- 标记阶段：从根对象（寄存器、执行栈、全局变量）出发，查找并标记堆中所有存活的对象
- 清除阶段：遍历堆中所有的对象，回收未被标记的对象，并将对应的内存加入空闲链表中

2. **三色标记法**

为了解决标记清除算法带来的长时间的STW，多数现代的垃圾收集器都会采用三色标记算法。该算法将程序中的对象分为白色、灰色和黑色三类：

- 白色：潜在的垃圾，未被垃圾收集器访问到的对象，在回收开始阶段，所有对象都被标记为白色；回收结束后，白色对象均不可达，内存会被释放
- 灰色：活跃的对象，已被垃圾收集器访问到，但存在指向白色对象的外部指针，垃圾收集器需要继续扫描其子对象
- 黑色：活跃的对象，已被垃圾收集器访问到，其所有字段都已被扫描

综上，三色标记法的核心过程包括：

- 遍历灰色集合，将灰色对象标记为黑色，放到黑色集合中
- 将黑色对象引用的白色对象标记为灰色，放到灰色集合中
- 重复上述两个步骤，直到不存在灰色对象，并清除所有的白色对象

3. **混合写屏障**

了解这个回收机制，需要先了解两个概念 **插入写屏障**：在对象A引用对象B时，对象B被标记为灰色 **删除写屏障**：被删除的对象，如果自身为灰色或白色，那么被标记为灰色

在V1.8版本，Go语言结合2.1部分的插入写屏障和2.2部分的删除写屏障，构成混合写屏障，其基本思想是：**对正在被覆盖的对象进行着色，且如果当前栈未扫描完成，则同样对指针进行着色。**

Gianhing 的回答

1. 标记清除算法

   标记存活的对象，在标记完成后统一清除没有标记过的对象（将该对象所占用内存的起始结束地址记录到空闲地址列表）。

优点：简单直接，速度较快

缺点：会产生大量不连续的内存碎片，可能无法给大对象分配足够的内存，导致内存溢出

2. 标记整理算法

   标记存活的对象，将存活对象都向一端移动，然后直接清理掉端边界以外的内存。

优点：不会产生内存碎片

缺点：整理时需要进行对象的移动，效率比较低

3. 复制算法

   将内存划分为大小相等的两块，每次只使用其中一块。垃圾回收时，将存活的对象复制到另一块上，然后将原先使用的内存空间一次清理掉。

优点：简单高效，不会产生内存碎片

缺点：只使用了一半内存

4. 分代收集算法

   现在虚拟机的垃圾回收采用的是分代收集算法：根据对象存活周期的不同将内存分成几块，不同块选择各自合适的垃圾回收算法。

一般把堆分成新生代和老年代

- 新生代
  - 存放寿命短的对象
  - 新生代分为一块较大的 Eden 区和两块较小的 Survivor 区（ From 区和 To 区，也称 S0 和 S1 ）
  - 垃圾回收采用的是**复制算法**

首先，把 Eden 区和 From 区中存活的对象复制到 To 区域（如果 To 区空间不足，会通过**分配担保机制**把对象移动到老年代），同时把这些对象年龄加 1；如果对象年龄达到阈值，则会晋升到老年代；然后，清空 Eden 区和 From 区的对象；最后，交换 From 区和 To 区，即原 To 区会成为下次 GC 时的 From 区。

- 老年代
  - 存放长时间存活的对象或大对象
  - 垃圾回收采用的是**标记清除算法**或**标记整理算法**

优点：能针对不同的情况选择最合适的垃圾回收算法

缺点：实现较为复杂

Git

1、什么是 Git 的 fork 命令？它和 clone 命令有什么区别?

官方解析

实际上，Git 本身并没有一个名为 fork 的命令。Fork 是一种在代码托管平台（如GitHub、GitLab等）上进行协作开发的概念。

Fork 操作的本质是复制一个仓库到自己的账户下，这样你就能在自己的仓库中进行修改，而不影响原始仓库。当你对自己仓库中的代码进行了修改，并希望将这些修改合并到原始仓库时，可以发起一个Pull Request。仓库的所有者可以审核你的修改，并选择是否将其合并到原始仓库中。

git clone命令是一个Git命令，用于将远程仓库克隆到本地计算机上。当你克隆一个仓库时，Git会将远程仓库的所有提交历史、分支和标签下载到本地。这样你就可以在本地进行开发、修改和提交操作。

Fork和Clone的区别如下：

1. 操作层面：Fork是在代码托管平台上进行的操作，它会在你的账户下创建一个新的仓库，与原始仓库相互独立。而Clone是在本地计算机上进行的操作，用于将远程仓库的内容复制到本地。
2. 目的：Fork主要用于协作开发，它让你可以在自己的仓库中进行修改，然后通过Pull Request将修改提交给原始仓库。而Clone用于将远程仓库的内容下载到本地，以便你可以在本地进行开发和修改。
3. 权限：Fork操作不需要原始仓库的写权限，只需要读权限。而Clone操作通常需要在本地提交修改后，将修改推送到远程仓库。这可能需要原始仓库的写权限（除非你在自己Fork的仓库中进行开发）。

总之，Fork和Clone是两个不同层面的操作，它们在协作开发和版本管理中扮演不同的角色。在实际应用中，你可能需要同时使用Fork和Clone来进行协作开发。例如，在GitHub上，你可以先Fork一个仓库到自己的账户下，然后使用git clone命令将Fork的仓库克隆到本地进行开发。

鱼友的精彩回答

having 的回答

fork

fork，英语翻译过来就是叉子，动词形式则是分叉，如下图，从左到右，一条直线变成多条直线

转到 git 仓库中，fork 则可以代表分叉、克隆 出一个（仓库的）新拷贝

包含了原来的仓库（即 upstream repository，上游仓库）所有内容，如分支、Tag、提交

如果想将你的修改合并到原项目中时，可以通过的 Pull Request 把你的提交贡献回 原仓库

clone

clone，译为克隆，它的作用是将文件从远程代码仓下载到本地，从而形成一个本地代码仓

执行clone命令后，会在当前目录下创建一个名为xxx的目录，并在这个目录下初始化一个 .git 文件夹，然后从中读取最新版本的文件的拷贝

默认配置下远程 Git 仓库中的每一个文件的每一个版本都将被拉取下来

当你在github发现感兴趣开源项目的时候，可以通过点击github仓库中右上角fork标识的按钮,点击这个操作后会将这个仓库的文件、提交历史、issues和其余东西的仓库复制到自己的github仓库中，而你本地仓库是不会存在任何更改

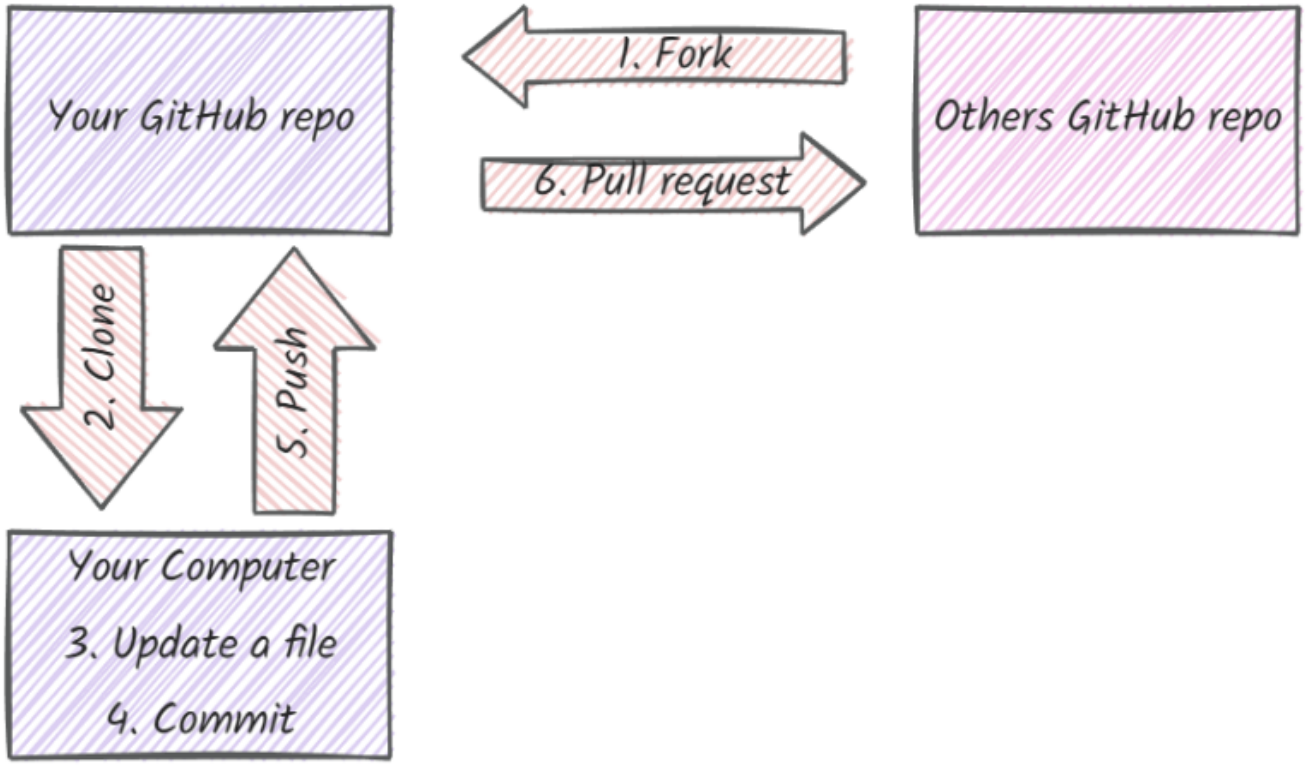
然后你就可以通过git clone对你这个复制的远程仓库进行克隆

后续更改任何东西都可以在本地完成，如git add、git commit一系列的操作，然后通过push命令推到自己的远程仓库

如果希望对方接受你的修改，可以通过发送pull requests给对方，如果对方接受。则会将你的修改内容更新到仓库中

- fork 只能对代码仓进行操作，且 fork 不属于 git 的命令，通常用于代码仓托管平台的一种“操作”
- clone 是 git 的一种命令，它的作用是将文件从远程代码仓下载到本地，从而形成一个本地代码仓
- branch 特征与 fork 很类似，fork 得到的是一个新的、自己的代码仓，而 branch 得到的是一个代码仓的一个新分支

HeiHei 的回答



Fork**是GitHub操作，用于在远程仓库中创建一个副本**，以便其他人可以基于该副本进行修改和贡献。通常，我们使用fork命令在GitHub或其他Git托管服务上创建一个公共仓库的个人分支，以便我们可以在其中进行自己的修改和实验，而不影响公共仓库的稳定性和安全性。

与fork命令相比，**clone命令则是将远程仓库中的代码复制到本地机器上，**以便我们可以在本地对代码进行修改和测试，而不需要连接到远程仓库。clone命令将整个仓库的历史记录、分支和标签都复制到本地，使得我们可以在本地完全重现远程仓库的状态。

因此，fork和clone命令之间的最大区别在于它们的操作目标。fork命令用于在远程仓库中创建个人分支，以便多人协作开发，而clone命令则用于将远程仓库复制到本地，以便在本地进行修改和测试。

2、什么是 Git 的 cherry-pick?

官方解析

Git 的 cherry-pick 是一种将指定的提交（commit）应用到当前分支的操作。它可以帮助我们将**某个分支上的某次提交复制到另一个分支上，而无需将整个分支合并过来。**

通常，我们在使用 Git 进行版本控制时，会在不同的分支上进行不同的开发工作。有时候，我们需要将**某个分支上的某次提交（commit）应用到当前分支上**，这时候就可以使用 cherry-pick 操作。

使用 cherry-pick 操作，我们可以复制指定的提交，然后将其应用到当前分支上，这个提交就成为了当前分支上的一个新的提交。cherry-pick 操作可以方便地将某个分支上的某个功能或修复应用到另一个分支上，而无需将整个分支合并过来。

cherry-pick 操作的使用方法如下：

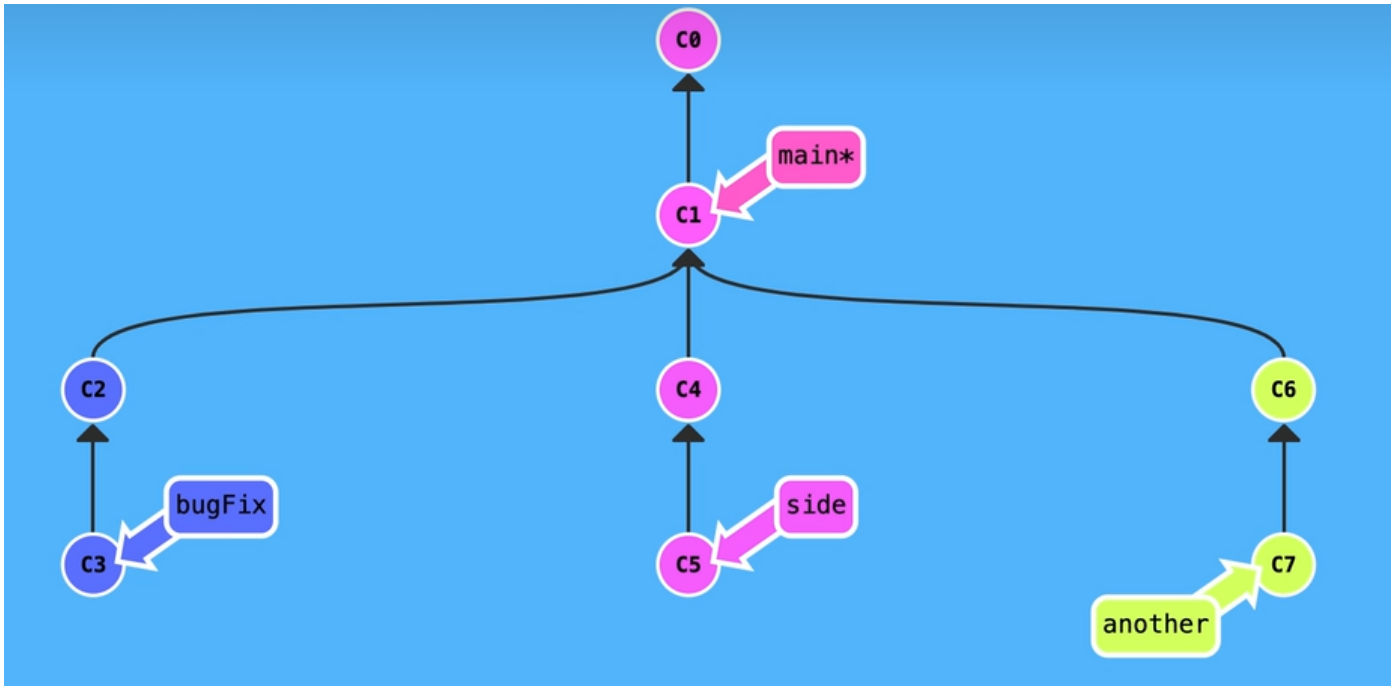
```
git cherry-pick <commit-id>
```

其中， 指定了要复制的提交的 ID。这个命令将复制指定的提交，然后将其应用到当前分支上。

需要注意的是，使用 cherry-pick 操作时，可能会出现冲突，这时候需要**手动解决冲突**，并提交一个新的提交来解决冲突。因此，在使用 cherry-pick 操作之前，我们需要仔细考虑哪些提交需要复制，以及是否会产生冲突等问题。

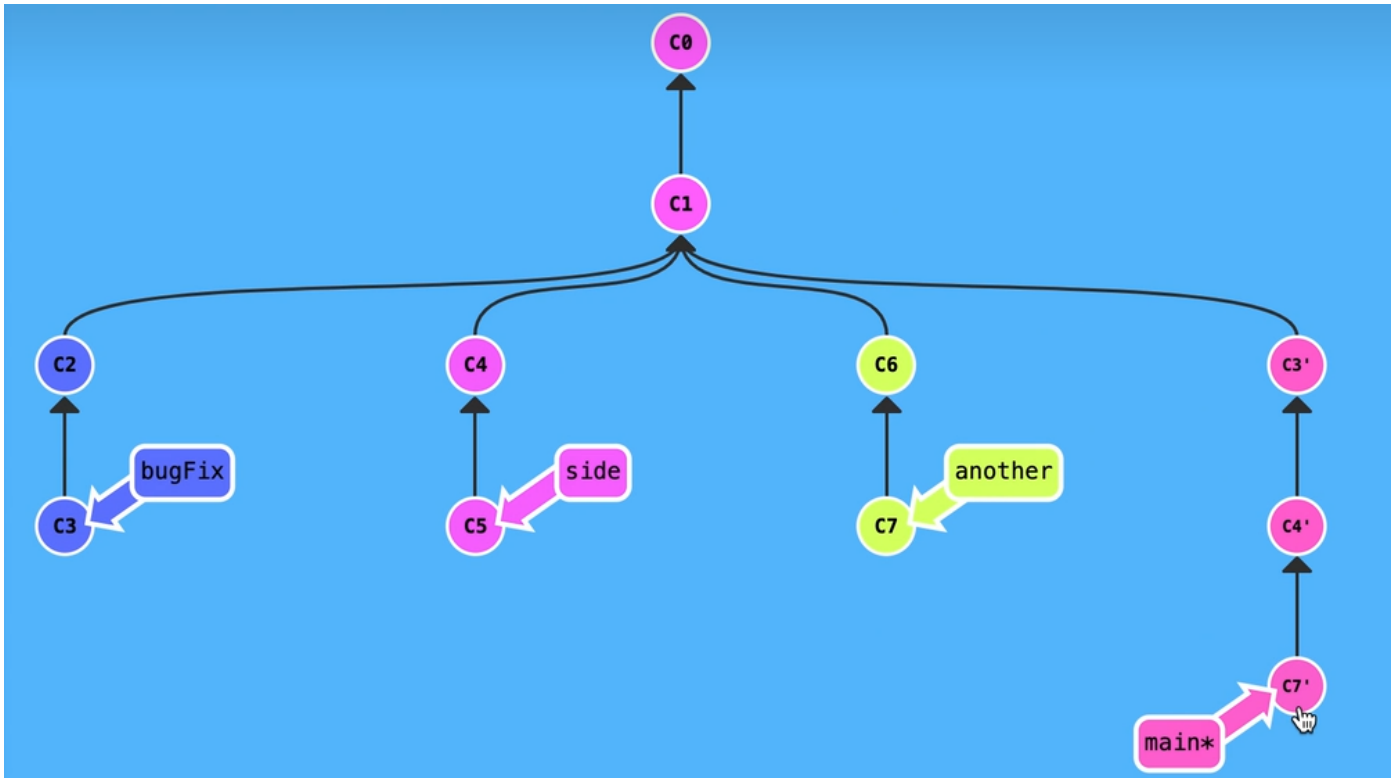
鱼友的精彩回答

维萨斯的回答



当前分支是在main上面!!!

```
git chery-pick C3 C4 C7
```



Starry 的回答

Git 的 cherry-pick 命令，允许将某个分支上的一个或多个提交应用到另一个分支上。

git cherry-pick 命令的作用：是将某个分支的指定提交复制到当前所在的分支上，而不是将整个分支合并过来。这可以方便地将某些特定的修改应用到其他分支上，而无需将整个分支合并。

简单来说，就是将指定的提交（commit）应用于其他分支。

使用 cherry-pick 命令时，需要指定要复制的提交的哈希值或者是可以移植的提交标识符（例如：branchname~2）。这个命令会自动找出这些提交所依赖的其他提交，并把它们也一并复制到目标分支上。

需要注意的是，如果多个提交之间存在冲突，则需要手动解决这些冲突。此外，cherry-pick 命令的使用应该谨慎，因为它可能会导致代码库中出现重复的代码和历史记录。

1. 基本用法：

```
git cherry-pick <commitHash>
```

这个命令会将指定的提交 commitHash，应用到当前分支，会在当前分支产生一个新的提交，当然它们的哈希值是不一样的。

2. 转移多个提交
   Cherry pick 支持一次转移多个提交。

比如：将 A 和 B 两个提交应用到当前分支。这会在当前分支生成两个对应的新提交。

```
git cherry-pick <HashA> <HashB>
```

转移一系列的连续提交：

```
git cherry-pick A..B
```

可以转移从 A 到 B 的所有提交。它们必须按照正确的顺序放置：提交 A 必须早于提交 B，否则命令将失败，但不会报错。

3. 常用配置
   git cherry-pick 命令的常用配置项如下。

（1）-e，--edit

打开外部编辑器，编辑提交信息。

（2）-n，--no-commit

只更新工作区和暂存区，不产生新的提交。

（3）-x

在提交信息的末尾追加一行(cherry picked from commit ...)，方便以后查到这个提交是如何产生的。

（4）-s，--signoff

在提交信息的末尾追加一行操作者的签名，表示是谁进行了这个操作。

（5）-m parent-number，--mainline parent-number

如果原始提交是一个合并节点，来自于两个分支的合并，那么 Cherry pick 默认将失败，因为它不知道应该采用哪个分支的代码变动。

-m 配置项告诉 Git，应该采用哪个分支的变动。它的参数parent-number是一个从1开始的整数，代表原始提交的父分支编号。

JIA 的回答

在 Git 中，cherry-pick 是一种将某个分支上的一个或多个提交应用到另一个分支的操作。通常情况下，我们会使用 git merge 命令将一个分支合并到另一个分支，但是有时候我们并不想将整个分支合并过来，而是只想选择其中的一些提交进行合并，这时候就可以使用 cherry-pick 命令。 具体来说，cherry-pick 命令会将指定的提交复制到当前分支，并创建一个新的提交，这个新的提交与原提交的 SHA 值不同，但是提交的内容是相同的。这样就可以实现将某个分支上的一些提交应用到当前分支的目的。 使用 cherry-pick 命令的步骤如下：

1. 切换到要应用提交的目标分支，例如：git checkout master。
2. 执行 cherry-pick 命令，指定要应用的提交的 SHA 值，例如：git cherry-pick 123456。
3. 解决冲突（如果有的话）。
4. 提交更改，例如：git commit -m "Merge commit message"。 需要注意的是，使用 cherry-pick 命令可能会导致提交历史出现分叉，因为新的提交与原提交的 SHA 值不同。此外，如果要应用的提交中包含一些依赖于其他提交的更改，那么 cherry-pick 命令可能会失败，因为这些依赖的提交并不在当前分支上。因此，在使用 cherry-pick 命令时，需要仔细考虑其带来的影响，并进行必要的测试和验证。

Linux

1、Linux 中的硬链接和软连接是什么，二者有什么区别?

官方解析

在 Linux 文件系统中，硬链接（hard link）和软链接（symbolic link）都是一种文件链接的方式，可以用于将一个文件链接到另一个文件上。它们的主要区别在于创建方式、所占空间和使用限制等方面。

硬链接是通过在文件系统中创建一个新的目录项（directory entry）指向同一文件 inode 的位置来实现的。因为硬链接实际上是指向同一 inode，所以如果原文件被删除，硬链接依然能够访问到原文件的内容。硬链接的使用范围比较受限，因为硬链接只能指向同一个文件系统内的文件，不能跨文件系统创建。

软链接是通过在文件系统中创建一个新的文件来实现的，该文件中包含指向另一个文件的路径。软链接可以跨文件系统创建，并且可以指向任何类型的文件。但是，当原文件被删除时，软链接将会失效。

总的来说，硬链接更加高效，因为它只是添加了一个新的目录项，所以对磁盘空间的消耗比软链接要小。但是，硬链接不能跨文件系统，所以在实际应用中需要根据具体的需求来选择使用哪种链接方式。

鱼友的精彩回答

林风的回答

Linux 系统下提供 ln 指令来进行文件链接

```
ln –s bar.txt foo.txt
```

ls 查看当前目录下的文件

```
➜ ll
total 1.2M
-rw-rw-r--. 1 Nova Nova     0 Aug 11 14:43  bar.txt
lrwxrwxrwx. 1 Nova Nova     7 Aug 11 14:43  foo.txt -> bar.txt
```

这个foo.txt,就是一个文件链接，文件链接主要分为硬链接和软链接，通过查看ln --help

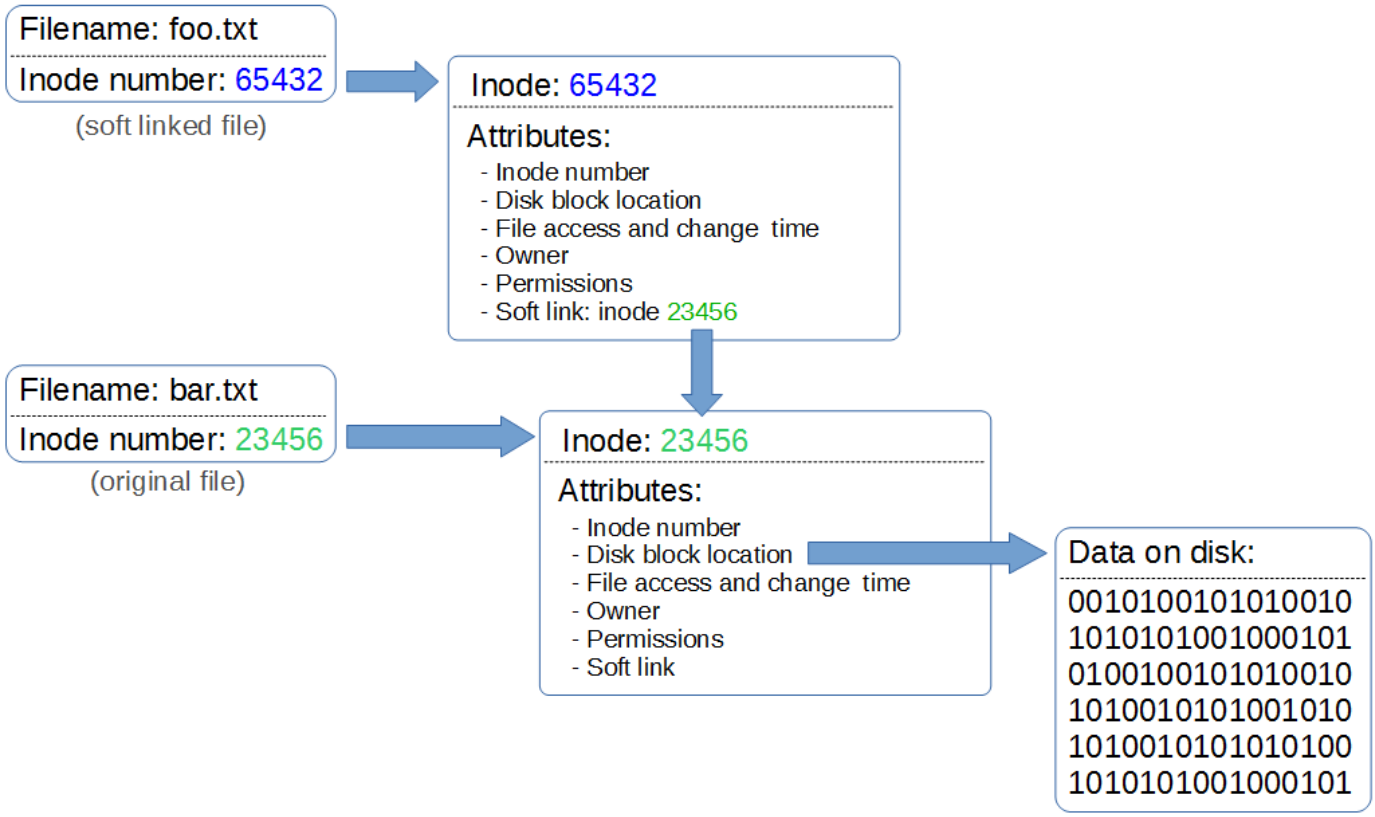
```
➜  Desktop ln --help
Usage: ln [OPTION]... [-T] TARGET LINK_NAME
Create hard links by default, symbolic links with --symbolic.

Mandatory arguments to long options are mandatory for short options too.
  -s, --symbolic              make symbolic links instead of hard links
```

ln 指令默认创建的是硬链接，如果加入了-s参数，则会生成一个软链接。

硬链接

Linux 下的文件是通过索引节点-Inode来识别文件。 硬链接当中，多个文件名可以指向同一个索引节点。 当创建一个硬链接的时候，文件系统会维护一个引用计数，只要有文件指向这个区块，它就不会从硬盘上消失。 两个文件拥有相同的 inode，通过查看文件内容也会发现是一个文件。只删除一个连接并不影响索引节点本身和其它的连接，只有当最后一个链接被删除后，文件的数据块及目录的连接才会被释放，也就是说，文件才会被真正删除

软链接



软链接又叫符号链接，这个文件包含了另一个文件的路径名，例如在上图中，foo.txt 就是 bar.txt 的软连接，bar.txt 是实际的文件，foo.txt包含的是对于 bar.txt 的 inode 的记录。

软连接可以是任意文件或目录，可以链接不同文件系统的文件，在对符号文件进行读或写操作的时候，系统会自动把该操作转换为对源文件的操作，但删除链接文件时，系统仅仅删除链接文件，而不删除源文件本身，这一点类似于 Windows 操作系统下的快捷方式。

区别

	软连接	硬链接
inode	软链接原文件＆链接文件拥有不同的 inode 号，表明他们是两个不同的文件	硬链接原文件和链接文件共用一个 inode 号，就像一个文件有多个文件名，说明他们是同一个文件
文件属性	软链接明确写出了是链接文件	硬链接没有写出来，因为在本质上硬链接文件和原文件是完全平等关系
跨越文件系统建立	支持	不支持
链接数目	软链接的链接数目不会增加，文件大小是不一样的（可以理解为快捷方式和对应源文件之间的关系）	硬链接文件显示的大小是跟原文件是一样的

另外一点，在删除文件时： 硬链接不会直接删除掉文件，除非这个链接是最后一个指向这个文件的链接； 软链接也不会直接删除掉文件，相当于删除一个快捷方式。

软链接应用 - 灵活切换不同版本的目标程序 - 动态库版本管理 - 快捷方式 硬链接应用 - 从不同角度对文件进行分类 - 文件多人共享 - 文件备份

Starry 的回答

linux中软链接和硬链接区别为：

1、软链接以路径的形式存在，硬链接以文件副本的形式存在；

2、软链接可以跨文件系统，硬链接不可以；

3、软链接可以对目录进行链接，硬链接不可以。

更详细的区别对比如图所示：

#	软链接	硬链接
1	软链接类似于 Windows 系统中的快捷方式	硬链接是原始文件的一个镜像副本
2	软链接又称符号链接	硬链接没有别名
3	链接中任何一个文件发生改变，都会同步到连接中的其它文件	和软链接一样
4	软链接可以跨文件系统进行创建	硬链接不可以跨文件系统进行创建
5	软链接可以指向文件或目录	硬链接只能指向文件
6	链接文件和原始文件之间的 inode 和文件权限不完全一致	链接文件和原始文件的 inode 和文件权限完全一致
7	链接文件只记录原始文件的路径，不记录原始文件的内容	链接文件记录了原始文件的内容
8	如果原始文件被移除，软链接就会因为指向不存在的文件而失效。这被称为“挂起链接”(hanging link)	即使原始文件被移除，链接文件也不受影响。
9	通过 `ln -s <原始文件> <链接文件>` 命令创建软链接	通过 `ln <原始文件> <链接文件>` 命令创建硬链接
10	软链接文件的文件权限中有一个特殊标记 `l`	硬链接文件没有特殊标记
11	通过 `find / -type l` 命令可以查找软链接文件	通过 `find / -samefile <原始文件>` 命令可以查找硬链接文件
12	通过 `symlinks <目录>` 命令可以查找失效的软链接	硬链接不存在失效链接

2、CC 攻击是什么？什么叫 DDOS 攻击？什么是网站数据库注入？

官方解析

1. CC 攻击（CC Attack）是一种网络攻击方式。它通常是指对服务器进行大量并发请求的攻击，从而导致服务器的瘫痪。攻击者通过使用大量的机器或网络中的代理服务器，向目标服务器发送大量的请求，以消耗服务器的带宽和资源，从而使其无法正常提供服务。CC 攻击可以是攻击者自己编写的脚本，也可以是专门的 CC 攻击工具。
2. DDOS 攻击（Distributed Denial of Service Attack）是一种网络攻击方式，它通常是指利用大量的计算机或网络中的代理服务器，同时向目标服务器发送大量的请求，从而导致目标服务器的瘫痪。DDOS 攻击可以采用多种方式实现，如 SYN 攻击、UDP 攻击、ICMP 攻击等。
3. 网站数据库注入（SQL Injection）是一种利用 Web 应用程序漏洞的攻击方式，攻击者通过将恶意 SQL 代码插入到 Web 应用程序的输入字段中，从而获取对数据库的未授权访问。网站数据库注入攻击可以导致许多问题，如破坏数据库的完整性、泄漏敏感数据、执行未经授权的操作等。常见的防御措施包括输入验证、参数化查询、使用安全的 API 等。

鱼友的精彩回答

Gundam 的回答

1. CC 攻击是什么？

CC 攻击（CC Attack，即“HTTP Flood”攻击）是一种常见的 DDoS 攻击，它针对的是 Web 服务器。CC 攻击通过模拟大量的用户请求来消耗 Web 服务器的资源，从而使其无法响应合法用户的请求。攻击者通常使用大量的代理服务器或僵尸网络来发动 CC 攻击。CC 攻击通常使用假冒的请求头和 IP 地址来混淆受害者的防御系统，从而让攻击更加难以被检测和防御。

2. 什么叫 DDOS 攻击?

DDoS 攻击（Distributed Denial of Service Attack）是一种网络攻击，旨在通过使用大量的计算机和网络设备同时向受害者的服务器发送流量，来使其无法提供正常的服务。攻击者可以使用多个计算机和网络设备来形成一个庞大的网络，称为“僵尸网络”或“肉鸡网络”，从而将攻击流量分散在多个来源。这种攻击可以让受害者的服务器处理大量无效流量，从而耗尽其计算资源和带宽，并最终导致其无法正常工作。

3. 什么是网站数据库注入?

网站数据库注入是一种针对 Web 应用程序的攻击，攻击者利用应用程序漏洞，向应用程序中的数据库中插入恶意代码。攻击者通常通过输入恶意代码来绕过应用程序的安全机制，从而获取访问受害者数据库的权限，并盗取敏感数据或者对数据库进行破坏。网站数据库注入攻击可以通过对 Web 应用程序的输入和输出进行过滤和验证来预防。

having 的回答

什么是 DDOS 攻击?

DDoS 是英文 Distributed Denial of Service 的缩写，意即“分布式拒绝服务”。分布式拒绝服务攻击发起后，攻击网络包就会从很多 DOS 攻击源(俗称肉鸡)犹如洪水般涌向受害主机，从而把合法用户的网络包淹没，导致合法用户无法正常访问服务器的网络资源，因此拒绝服务攻击又被称之为“洪水攻击”

什么是 CC 攻击?

HTTP Flood：俗称 cc 攻击。

CC 攻击（Challenge Collapsar）是 DDOS（分布式拒绝服务）的一种，前身名为 Fatboy 攻击，也是一种常见的网站攻击方法。攻击者通过代理服务器或者肉鸡向向受害主机不停地发大量数据包，造成对方服务器资源耗尽，一直到宕机崩溃。相比其它的 DDOS 攻击 CC 似乎更有技术含量一些。这种攻击你见不到真实源 IP，见不到特别大的异常流量，但造成服务器无法进行正常连接。最让站长们忧虑的是这种攻击技术含量低，利用更换 IP 代理工具和一些IP代理一个初、中级的电脑水平的用户就能够实施攻击。

针对系统的每个 Web 页面，或者资源，或者 Rest API，用大量肉鸡，发送大量 http request。这种攻击主要是针对存在 ASP、JSP、PHP、CGI 等脚本程序，并调用 MSSQLServer、MySQLServer、Oracle 等数据库的网站系统而设计的，特征是和服务器建立正常的 TCP 连接，并不断的向脚本程序提交查询、列表等大量耗费数据库资源的调用，典型的以小博大的攻击方法。缺点是对付只有静态页面的网站效果会大打折扣。

什么是 SQL 注入?

攻击者把 SQL 命令插入到 Web 表单的输入域或页面请求的查询字符串，欺骗服务器执行恶意的 SQL 命令。在某些表单中，用户输入的内容直接用来构造（或者影响）动态 SQL 命令，或作为存储过程的输入参数，这类表单特别容易受到 SQL 注入式攻击它通过将任意 SQL 代码插入数据库查询，使攻击者能够完全控制 Web 应用程序后面的数据库服务器。攻击者可以使用 SQL 注入漏洞绕过应用程序安全措施；可以绕过网页或 Web 应用程序的身份验证和授权，并检索整个 SQL 数据库的内容；还可以使用 SQL 注入来添加，修改和删除数据库中的记录。

2、如何在 Linux 中查看系统资源使用情况？比如内存、CPU、网络端口。

官方解析

在 Linux 中，可以使用一些命令和工具来查看系统资源使用情况。下面介绍一些常用的命令和工具：

1. top：top 命令可以实时地显示系统中各个进程的资源使用情况，包括 CPU 使用率、内存使用率、进程 ID 等信息。可以通过 top 命令来查看当前系统的 CPU 和内存使用情况。
2. free：free 命令可以查看系统中的内存使用情况，包括总内存量、已使用内存量和可用内存量等。可以通过 free 命令来查看当前系统的内存使用情况。
3. df：df 命令可以查看系统中磁盘的使用情况，包括磁盘总容量、已使用空间和可用空间等。可以通过 df 命令来查看当前系统的磁盘使用情况。
4. iostat：iostat 命令可以查看系统的 CPU 和磁盘 I/O 使用情况。可以通过 iostat 命令来查看当前系统的 CPU 和磁盘 I/O 使用情况。
5. netstat：netstat 命令可以查看系统中的网络连接情况，包括本地地址、远程地址、连接状态等信息。可以通过 netstat 命令来查看当前系统的网络连接情况。
6. lsof：lsof 命令可以查看系统中打开的文件和网络连接情况，包括文件名、文件描述符、进程 ID、进程名等信息。可以通过 lsof 命令来查看当前系统的文件和网络连接情况。

这些命令和工具都可以帮助我们了解当前系统资源的使用情况，方便我们对系统进行优化和管理。同时，也可以通过一些其他的工具，如 htop、iotop 等，来查看系统资源使用情况。

鱼友的精彩回答

爱吃鱼蛋的回答

在 Linux 中，有多种命令可以查看系统资源使用情况。以下是常用的几个命令：

1. top：top 命令可以实时显示系统进程和资源使用情况，包括 CPU 占用率、内存占用率、交换分区占用率等。按下键盘上的 M 键可以按照内存占用率排序，按下键盘上的 P 键可以按照 CPU 占用率排序;

```
$ top
```

2. free：free 命令可以显示系统内存使用情况，包括总内存、已用内存、空闲内存、缓存和交换分区等信息。运行 free 命令时，可以添加参数 -h 以更友好的方式显示输出结果;

```
$ free –h
```

3. vmstat：vmstat 命令可以显示系统虚拟内存使用情况，包括内存、交换分区、CPU 和 I/O 等信息。运行 vmstat 命令时，可以添加参数 -a 以显示所有的活动和非活动内存块;

```
$ vmstat –a
```

4. ps：ps 命令可以显示系统进程列表和相关信息，包括进程 ID、进程名、用户、内存占用等。可以通过结合不同的参数来获取更详细的信息，还可结合管道符进行筛选，比如 ps aux 可以显示所有进程的详细信息;

```
$ ps aux
```

5. netstat：netstat 命令可以显示当前网络连接状态和相关信息，包括本地 IP 地址、远程 IP 地址、端口号等。可以通过结合不同的参数来获取更详细的信息，比如 netstat -tulpn 可以显示所有 TCP 和 UDP 连接的详细信息，包括 PID 和进程名称;

```
$ netstat -tulpn
```

猫十二懿的回答

以下是Linux中一些常用的命令来查看系统资源使用情况：

- top：实时动态地显示系统的 CPU 使用情况、进程信息、内存占用情况等。可以使用 q 键退出。top命令可以实时显示各个进程的 CPU 占用率、内存占用率等信息。
- htop：类似于 top，但是在交互性和功能上更加强大。可以使用 q 键退出。
- free：查看系统内存的使用情况，包括总共的内存量、已使用的内存量、空闲的内存量等。可以使用 -h 参数让输出结果以易读的方式显示。
    - 内存使用情况： free -h，其中 -h 参数可以让输出结果以易读的方式显示，该命令可以显示系统总共的内存量、已使用的内存量、空闲的内存量等信息。
- vmstat：以文本形式显示系统的 CPU 使用情况、内存使用情况、虚拟内存使用情况等。可以使用 -s 参数来查看更详细的信息。
- sar：收集系统历史性能数据，并以报告形式输出。可以通过安装 sysstat 包来使用该命令。
- iostat：显示磁盘 I/O 活动情况，包括读写速度、等待时间等信息。可以使用 -x 参数来显示更详细的信息。
    - iostat -c 1 1 命令可以显示 CPU 的负载情况。
- mpstat：显示每个 CPU 核心的使用情况，包括用户模式下和内核模式下的占用情况、CPU 间的切换次数等信息。
- nmon：显示 Linux 系统的各种性能指标，包括 CPU 使用率、内存使用率、磁盘 I/O 使用率、网络吞吐量等。可以使用交互式界面查看。
- netstat：可以查看正在使用的网络连接、监听的端口等信息。如果只想查看特定端口的连接情况，可以使用 netstat -an | grep PORT 命令来进行过滤。
    - 该命令可以显示当前正在使用的网络连接、监听的端口等信息。如果只想查看特定端口的连接情况，可以使用 netstat -an | grep PORT 命令来进行过滤，其中 PORT 指代特定的端口号。

> 作者：编程导航知识星球 + 星球鱼友们

Nginx

1、Nginx 是什么？它有哪些应用场景?

官方解析

Nginx（发音为“engine-x”）是一个**高性能的开源 Web 服务器**和**反向代理服务器**，可以处理大量的并发连接和请求。它使用**事件驱动的异步架构和多线程设计**，可以高效地处理并发请求，同时也支持**反向代理、负载均衡、动态HTTP缓存、SSL/TLS终止、基于模块的扩展**等功能。

Nginx 的应用场景非常广泛，以下是其中的几个：

- **Web 服务器**：Nginx 可以作为 HTTP 服务器，处理并发的静态请求和动态请求，并且可以支持**负载均衡和缓存**，为网站提供高性能和高可用性。
- **反向代理服务器**：Nginx 可以作为反向代理服务器，接收客户端请求并将其转发到后端服务器，同时也可以支持负载均衡和缓存。
- **邮件代理服务器**：Nginx 可以作为邮件代理服务器，支持 POP3、IMAP 和 SMTP 协议，并且可以支持 SSL/TLS 加密和反垃圾邮件功能。
- **流媒体服务器**：Nginx 可以作为流媒体服务器，支持 RTMP、HLS 和 DASH 协议，可以用于实现直播、点播和视频-on-demand（VoD）等场景。

总之，Nginx 具有高性能、可扩展性和可靠性等特点，被广泛应用于大型网站、互联网公司、云计算、视频流媒体、游戏等领域。

> 鱼皮补充：这题建议大家说一下自己是怎么使用 Nginx 的，这是一个很重要、但是缺容易被忽略的后端技术

鱼友的精彩回答

一万小时的回答

nginx 是一个高性能的 http 和反向代理服务器 和 tomcat 一样也是 web 服务器。

tomcat 能够应对的并发 一般在 400 左右， 也就是支持 400 个请求 并发访问，比如学校选课的时候 尤其一些选修课 。你在进入选课系统的时候会很卡，再如比如说 经历一次春运，票放出来的一瞬间，就没了，这一瞬间的请求买票可能高达上亿，会瞬间冲垮服务器 导致宕机(死机)，所以 我们为了应对这种高并发 能够经受高负载这种场景 我们选择使用 nginx 服务器

nginx 服务器可以应对 50000 个并发连接

nginx 作为一个 web 服务器，它是不支持 java 程序的，tomcat 是支持 java 程序的，那么如果我想要让 nginx 能够部署我的 java 项目 那么需要借助 tomcat 来帮助，所以我们需要将 nginx 和 tomcat 进行合作

反向代理，反向代理起始就是代理服务器端，也就是使用 nginx 去代理 tomcat

2、什么是正向代理和反向代理，如何使用 Nginx 做反向代理?

官方解析

正向代理和**反向代理**都是代理服务器的两种应用场景，它们在网络请求的处理过程中扮演不同的角色：

正向代理（Forward Proxy）：正向代理**位于客户端和目标服务器之间**，客户端通过正向代理来访问目标服务器。正向代理代表客户端发起请求，**隐藏客户端的真实身份**。常见的应用场景包括**使用魔法、访问内网资源、缓存和过滤**等。

反向代理（Reverse Proxy）：反向代理位于客户端和目标服务器之间，客户端直接访问反向代理服务器，反向代理将请求转发给目标服务器。反向代理代表目标服务器接收请求，隐藏目标服务器的真实身份。常见的应用场景包括**负载均衡、安全防护、缓存和SSL终端**等。

使用 Nginx 作为反向代理的方法如下：

安装Nginx：根据操作系统和需求选择合适的Nginx版本进行安装。安装完成后，启动Nginx。

配置反向代理：编辑 Nginx 的配置文件（通常位于/etc/nginx/nginx.conf或/etc/nginx/sites-available/default），在 http 或 server 块中添加反向代理配置。

示例配置：

```
http {
    ...
    server {
        listen 80; # 监听的端口号
        server_name example.com; # 反向代理的域名

        location / {
            proxy_pass http://backend_server; # 将请求转发给目标服务器
            proxy_set_header Host $host; # 设置请求头部信息
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }
    }

}
```

在这个示例中，将客户端的请求转发给名为 backend_server 的目标服务器。同时设置了一些请求头部信息，以便目标服务器获取客户端的真实 IP 地址。

重启 Nginx：保存配置文件并重启 Nginx，使配置生效。通常可以使用 nginx -s reload 或 systemctl restart nginx 命令重启 Nginx。

完成以上步骤后，Nginx 就被配置为反向代理服务器，可以将客户端的请求转发给目标服务器。

鱼友的精彩回答

林寻的回答

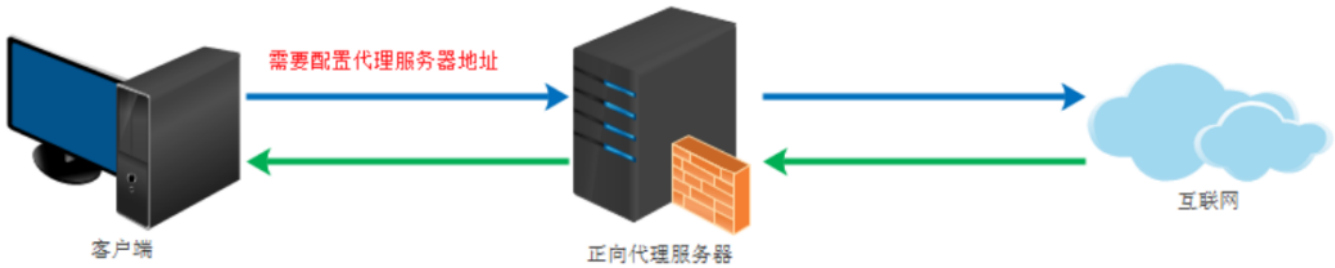
什么是正向代理和反向代理

1. 正向代理
2. 反向代理
3. 两者的区别

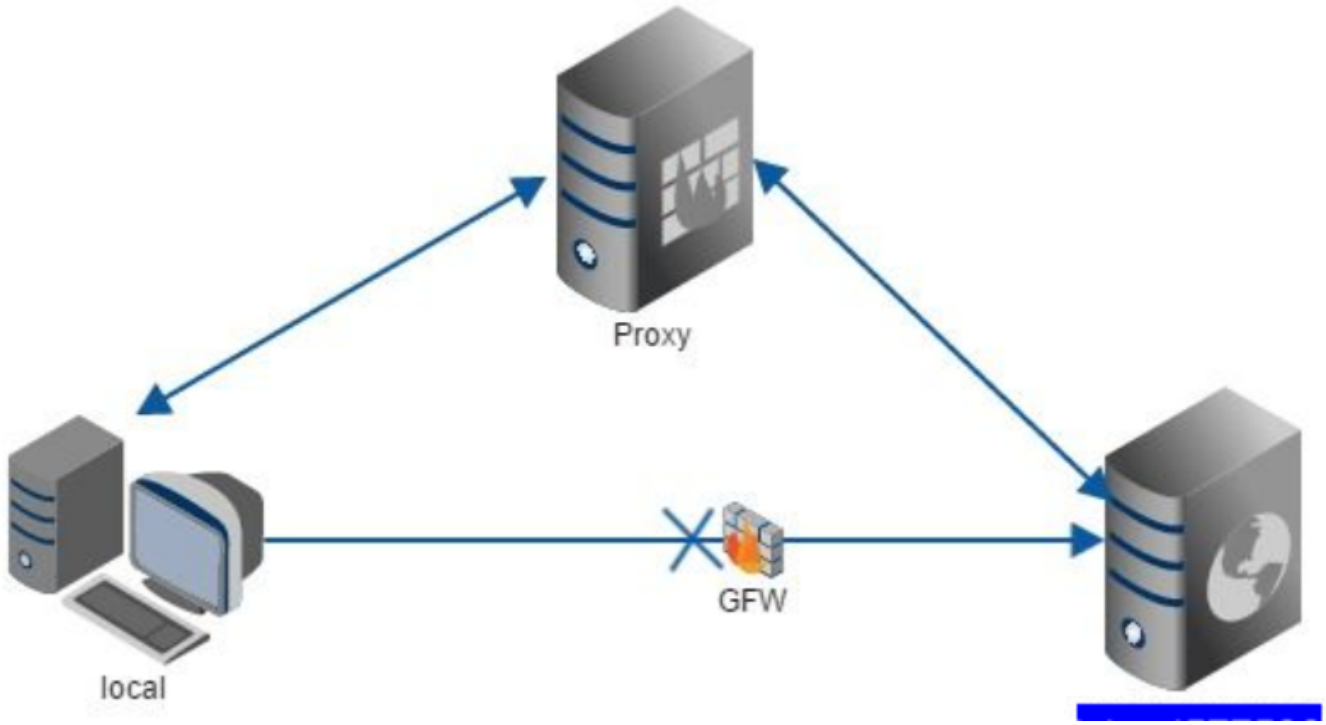
1.正向代理

正向代理隐藏真实客户端

正向代理，就是一个位于客户端和原始服务器之前的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并且指定目标（原始服务器），然后代理向原始服务器转交请求并将获得的内容返回给客户端，客户端才能使用正向代理。



比如我们要去访问某个网站，我们直接访问不通，那么我们就可以找一个代理服务器为我们服务，我们通过代理服务器请求到这个网站。对于这个网站而言他只知道有一个服务器访问了自己，并不知道你访问了他。



再举一个简单的例子告诉你什么是正向代理：

假设 A 和 B 是同学，但平时并不是很熟，A 向 B 借钱，被 B 拒绝了。

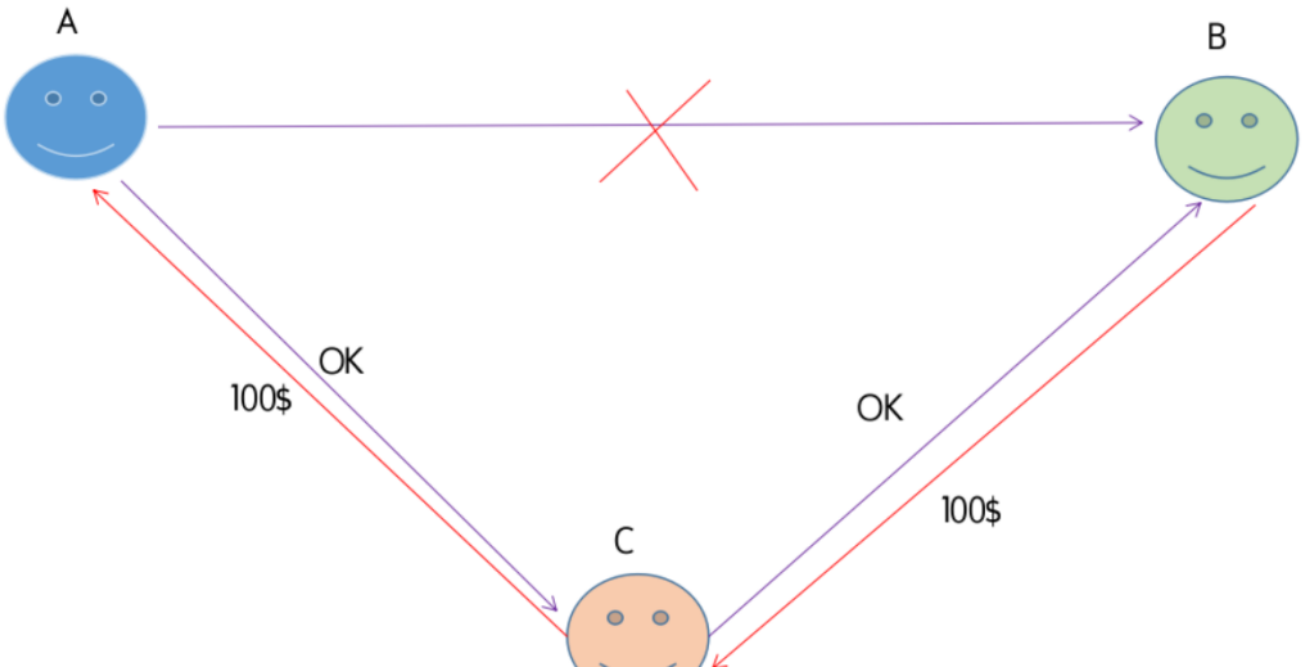
此时 A 联系了 C，C 是 A 的好朋友，C 和 B 也很熟。

C 向 B 借了钱并且给了 A。

此时 A 拿到了 B 的钱，但B并不知道他的钱借给了 A。

这时B同学扮演了一个非常关键的角色，就是代理，也可以说是正向代理

即就是隐藏了真实的客户端



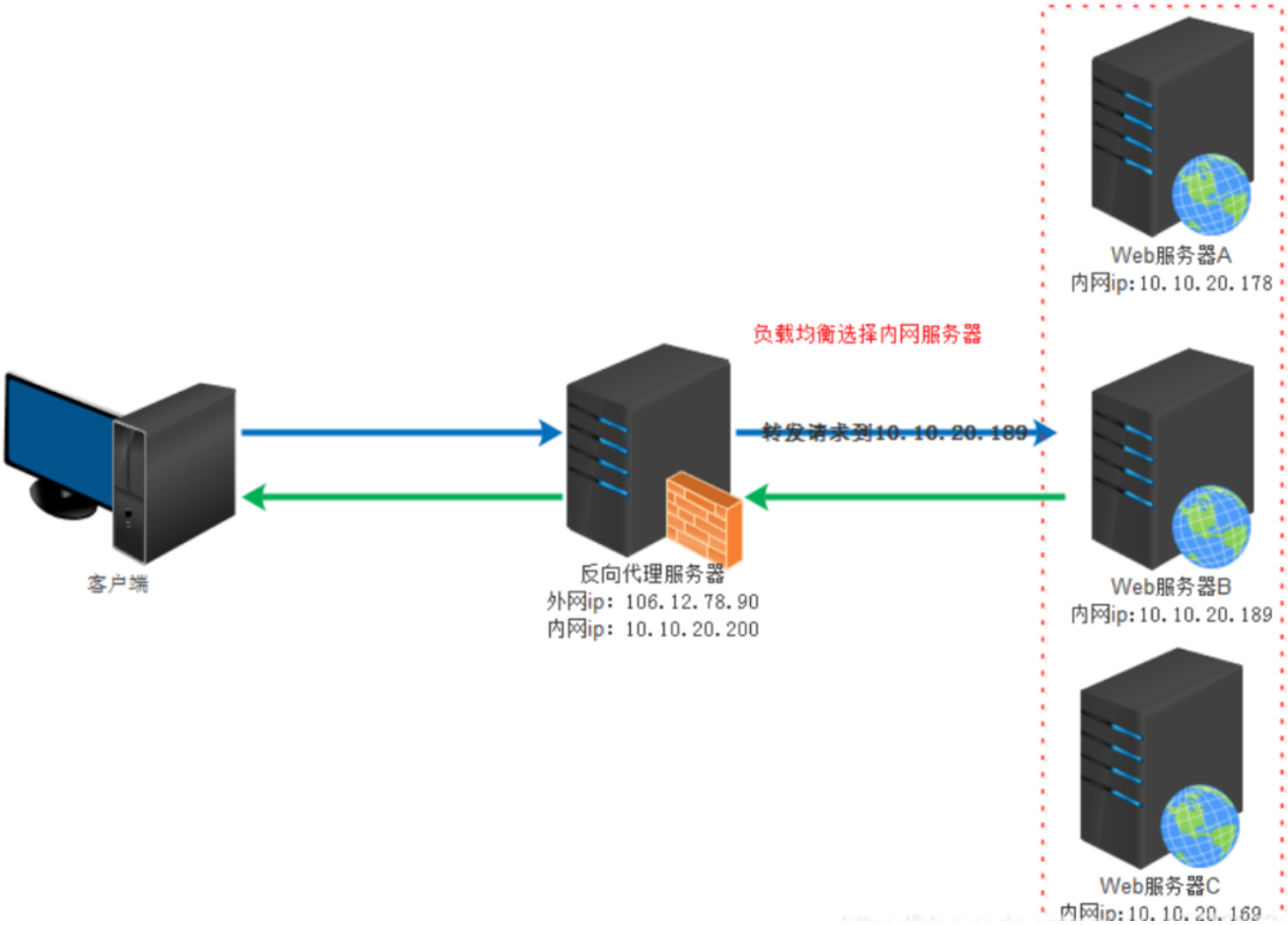
正向代理的过程，隐藏了真实的客户端。客户端请求的服务都被代理服务器代替来请求

2.**反向代理**

反向代理隐藏真实服务端

反向代理服务器位于用户与目标服务器之间，但是对于用户而言，反向代理服务器就相当于目标服务器，即用户直接访问反向代理服务器就可以获得目标服务器的资源。同时，用户不需要知道目标服务器的地址，也无须在用户端作任何设定。反向代理服务器通常可用来作为 Web 加速，即使用反向代理作为 Web 服务器的前置机来降低网络和服务器的负载，提高访问效率。

反向代理方式是指以代理服务器来接收 internet 网上的连接请求，然后将请求转发给内部网络上的服务器，并从服务器上得到的结果返回给 internet 上请求连接的客户端，此时代理服务器对外就表现为一个节点服务器。



简单举例什么是反向代理：

我们平时访问百度时，直接访问网址。

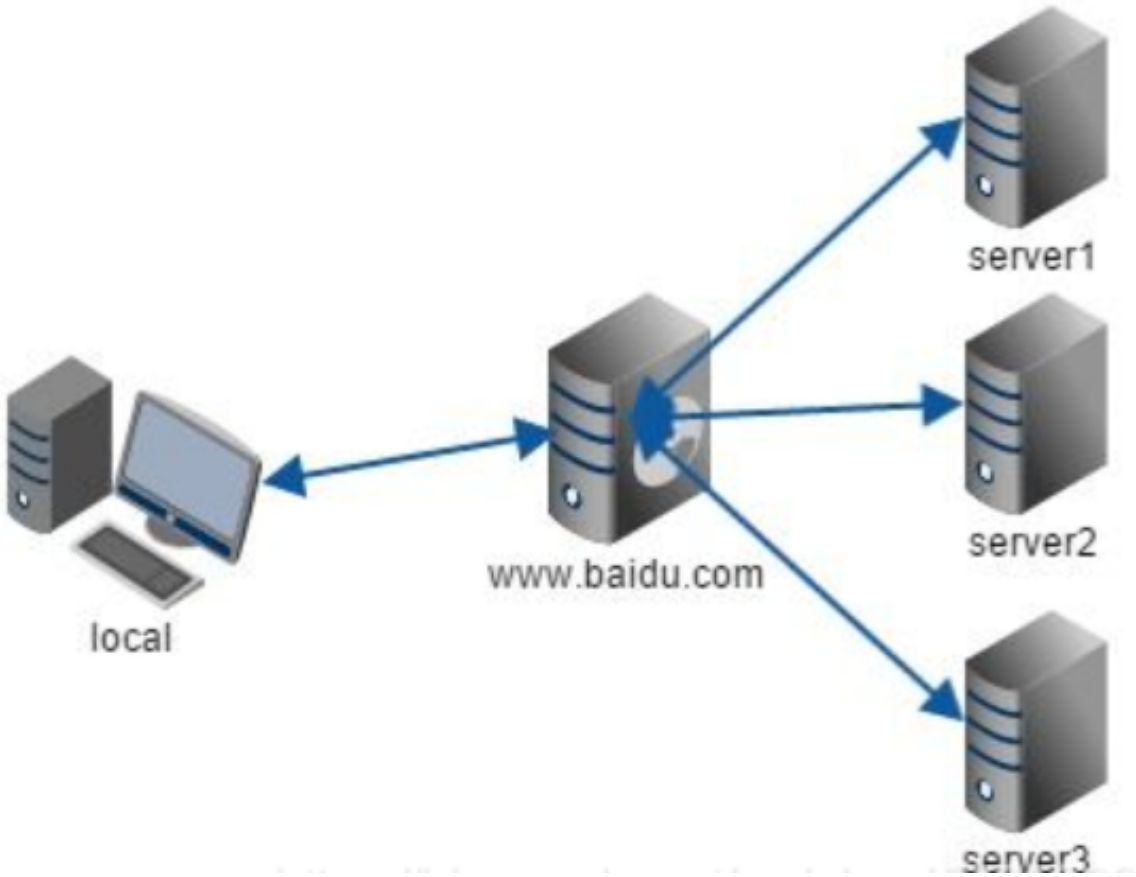
它背后可能有成千上万的服务器为我们服务，但具体是哪一台为我们服务，我们并不知道，也没必要知道。

我们只需要知道反向代理服务器是谁就可以（只要达到目的就可以了）。

网址就是我们的反向代理服务器，它会把我们的请求发送到真实的服务器那里去。

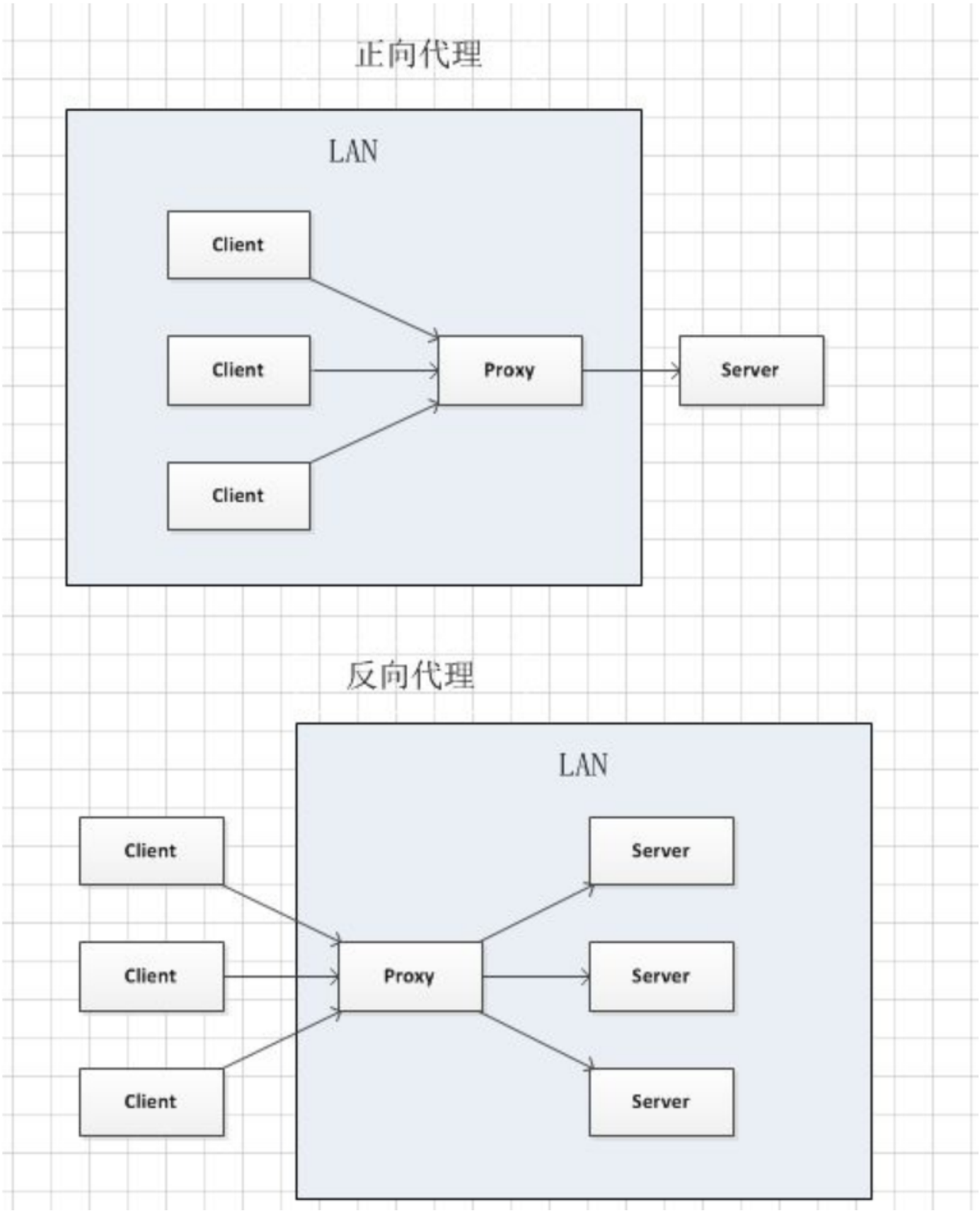
即隐藏真实的服务端

Nginx 就是性能非常好的反向代理服务器，用来做负载均衡。



3. **两者的区别**

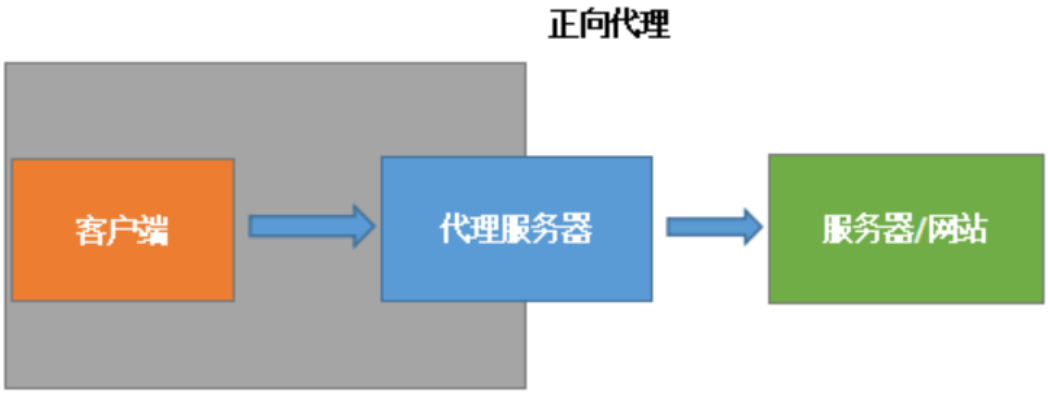
两者的区别在于代理的对象不一样：正向代理代理的对象是客户端，反向代理代理的对象是服务端



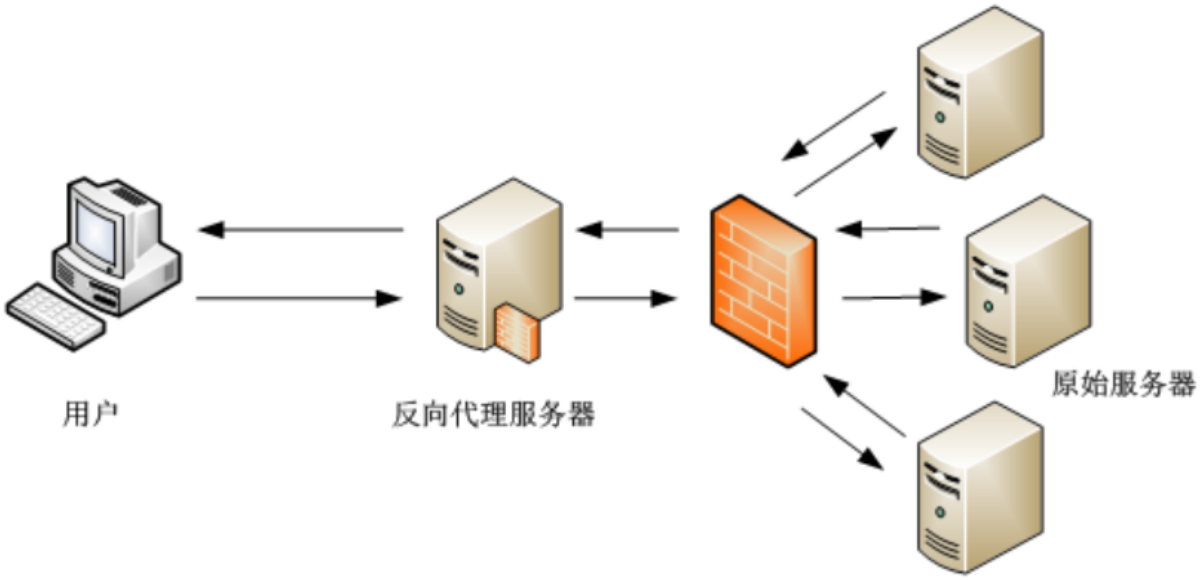
正向代理中，proxy 和 client 同属一个 LAN，对 server 透明；
反向代理中，proxy 和 server 同属一个 LAN，对 client 透明。
实际上 proxy 在两种代理中做的事都是代为收发请求和响应，不过从结构上来看正好左右互换了下，所以把后出现的那种代理方式叫成了反向代理。

猫十二懿的回答

正向代理是指代理服务器代表客户端向目标服务器发送请求，代理服务器与目标服务器通信，并将响应返回给客户端。在这种情况下，目标服务器不知道请求的来源是代理服务器还是真正的客户端，因此正向代理可以用于隐藏客户端的真实 IP 地址和身份信息。客户端必须要进行一些特别的设置才能使用正向代理。就像要访问 google 用 vpn 代理翻墙去访问（用户知道要访问真正的服务器）



反向代理是指==代理服务器代表原始服务器==向客户端发送响应，代理服务器接收客户端请求并将其转发到原始服务器，然后将原始服务器的响应返回给客户端。在这种情况下，客户端不知道响应的来源是原始服务器还是代理服务器，因此反向代理可以用于负载均衡、缓存静态资源、增加安全性等方面。



Nginx 是一个功能强大的开源 Web 服务器，也是一个反向代理服务器。使用 Nginx 做反向代理可以将客户端请求转发到后端多台真实服务器上，从而实现负载均衡、缓存静态资源、防止攻击等功能。下面是使用 Nginx 做反向代理的示例代码：

```
http {
    upstream backend { # upstream 模块指定了后端服务器的列表，可以设置不同的权重，以实现负载均衡；
        # 设置后端服务器列表
        server backend1.com weight=5;
        server backend2.com;
        server unix:/tmp/backend3;
```

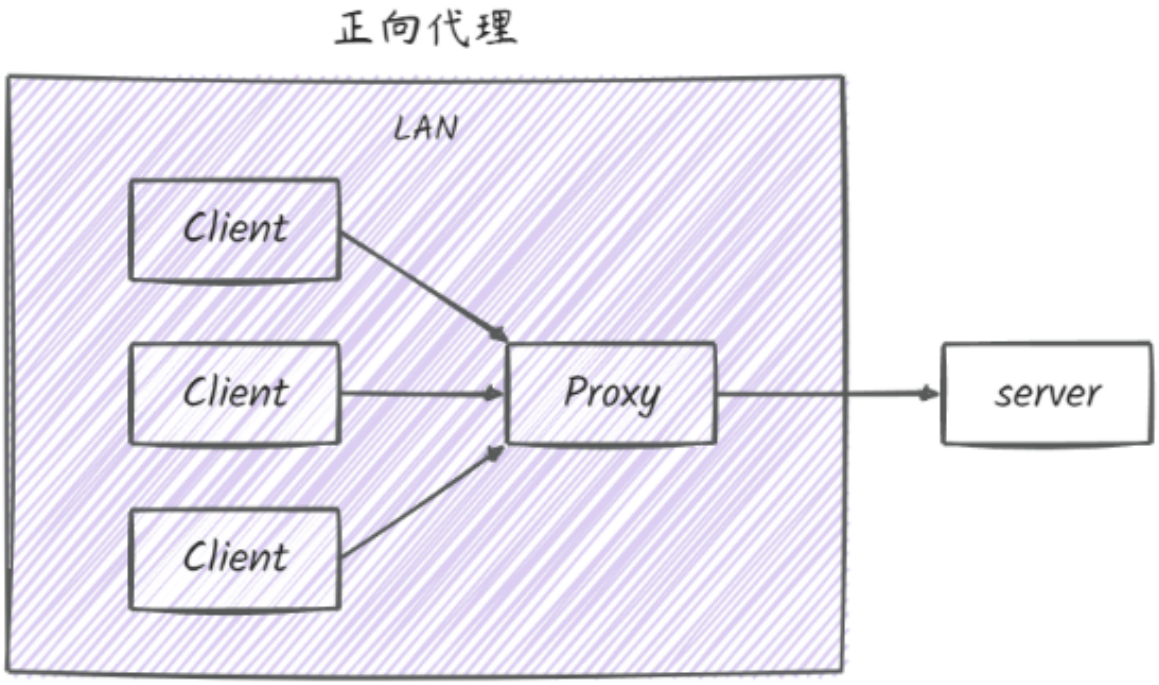
```
    }

    server {
        listen 80 ; # 监听端口
        server_name shier.com;

        location / { # location 模块指定了反向代理的规则，以及缓存静态资源和防止攻击等功能
            # 设置反向代理规则
            proxy_pass http://backend/; # 将请求转发给原始服务器
            proxy_set_header Host $host; #设置请求头部信息
            proxy_set_header X-Real-IP $remote_addr;

            # 缓存静态资源
            proxy_cache_path /var/cache/nginx levels=1:2 keys_zone=my_cache:10m
inactive=60m;
            proxy_cache_key "$scheme$request_method$host$request_uri";
            proxy_cache_valid any 20m;
            proxy_cache_bypass $http_pragma;
            proxy_cache_revalidate on;

            # 防止攻击
            limit_conn_zone $binary_remote_addr zone=addr:10m;
            limit_conn addr 5;
            limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
            limit_req zone=one burst=50 nodelay;
        }
    }
}
```

HeiHei 的回答

正向代理：

正向代理是一个位于客户端和目标服务器之间的代理服务器（中间服务器）。为了从目标服务器取得内容，客户端向代理服务器发送一个请求，并且指定目标服务器，之后代理向目标服务器转发请求，将获得的内容返回给客户端。正向代理的情况下，客户端必须要进行一些特殊的设置才能使用。

- 正向代理需要主动设置代理服务器 ip 或者域名进行访问，由设置的服务器 ip 或者域名去访问内容并返回
- 正向代理是代理客户端，为客户端收发请求，使真实客户端对服务器不可见。

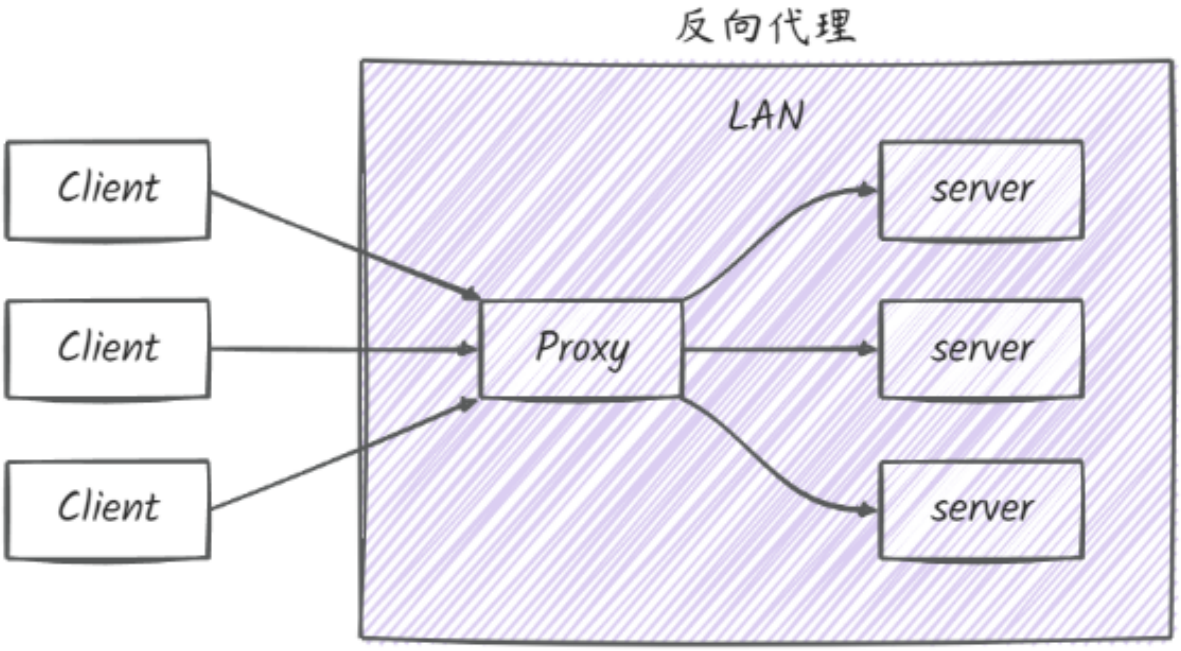


使用场景：比如我们国内访问谷歌，直接访问访问不到，我们可以通过一个正向代理服务器，请求发到代理服，代理服务器能够访问谷歌，这样由代理去谷歌取到返回数据，再返回给我们，这样我们就能访问谷歌了

反向代理：

反向代理是指以代理服务器来接收客户端的请求，然后将请求转发给内部网络上的服务器，将从服务器上得到的结果返回给客户端，此时代理服务器对外表现为一个反向代理服务器。
对于客户端来说，反向代理就相当于目标服务器，只需要将反向代理当作目标服务器一样发送请求就可以了，并且客户端不需要进行任何设置。

- 正向代理需要配置代理服务器，而反向代理不需要做任何设置。
- 反向代理是代理服务器，为服务器收发请求，使真实服务器对客户端不可见。



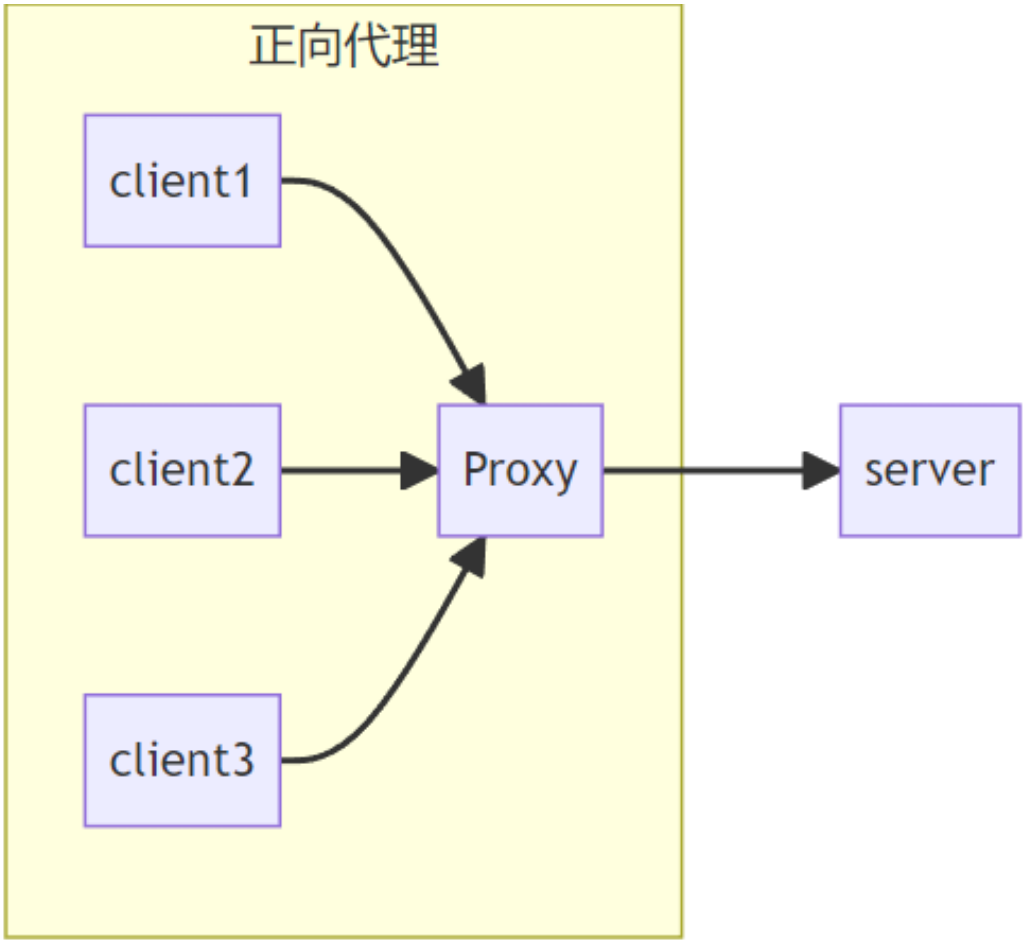
使用 Nginx 实现反向代理：

示例：使用 Nginx 反向代理，根据访问的路径跳转到不同端口的服务中，Nginx 监听端口为 9001
访问 http://192.168.17.129/edu/直接跳转到 127.0.0.1:8080
访问 http://192.168.17.129/vod/直接跳转到 127.0.0.1:8081
第一步，需要准备两个 tomcat，一个 8080 端口，一个 8081 端口，并准备好测试的页面
第二步，修改 nginx 的配置文件，在 http 块中配置 server

```
server {
  listen       9001;
  server_name  192.168.17.129;

  location ~ /edu/ {
    proxy_pass  http://127.0.0.1:8080
  }

  location ~ /vod/ {
    proxy_pass  http://127.0.0.1:8081
  }
}
```

Gianhing 的回答

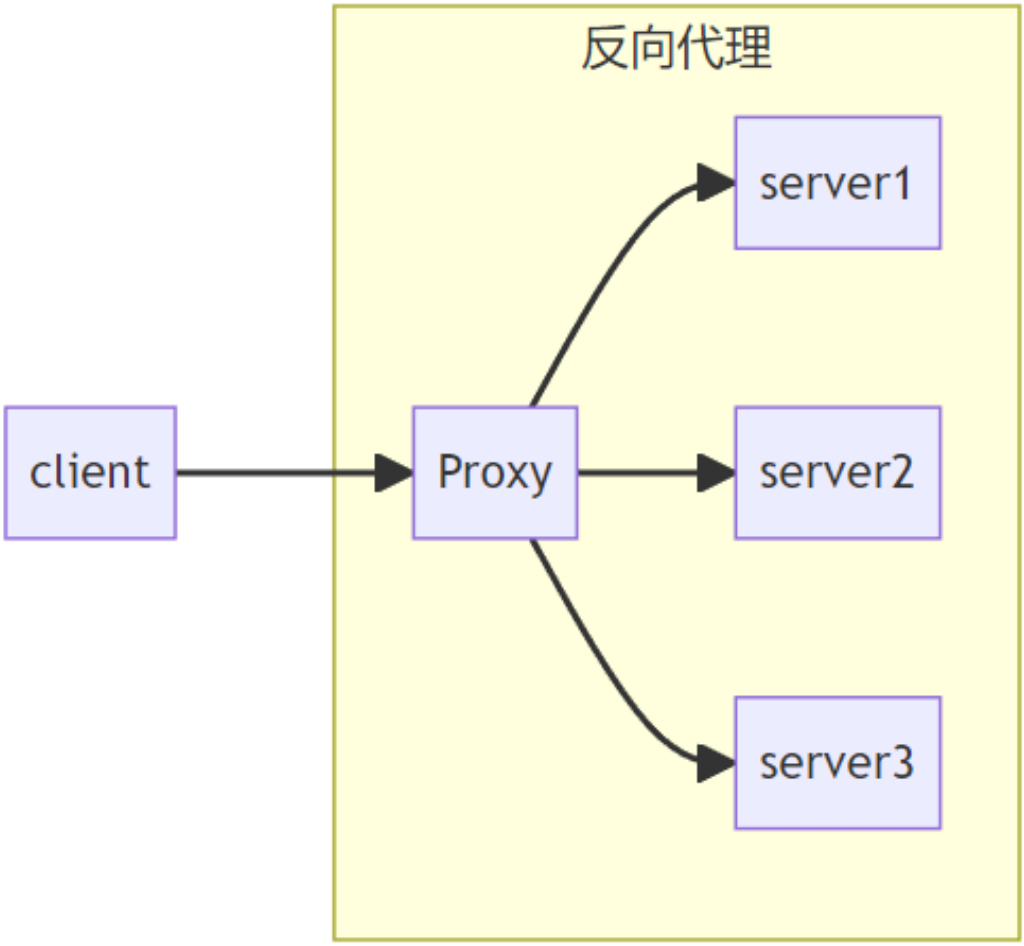
正向代理：客户端向代理服务器发送一个请求并指定目标，由代理服务器向目标服务器发起请求，并将响应结果返回给客户端。**代理的对象是客户端**



使用场景：

- 突破访问限制：正向代理可以访问受限制的资源，例如被墙的网站和内网资源等
- 提高访问速度：正向代理能够缓存数据，当客户端请求相同资源时，能直接从缓冲区获取并返回，提高访问速度
- 隐藏客户端信息：正向代理能够隐藏客户端的真实 IP 和其他信息，保护客户端的隐私
- 网络爬虫：通过设置代理服务器，爬取目标网站的数据

反向代理：客户端发起请求到代理服务器，代理服务器将请求转发到内部服务器，获取到服务器的响应并返回给客户端。**代理的对象是服务器**



使用场景：

- 负载均衡：反向代理可以将客户端的请求分发到多台服务器上，实现负载均衡
- 缓存加速：反向代理通过缓存服务器上的静态资源，例如页面、图片和文件等，加速网站的访问速度
- 安全防护：反向代理可以在客户端和服务器之间进行安全过滤，如过滤恶意攻击、限制外部访问等，保护服务器的安全

使用 Nginx 做反向代理：

1. 安装 Nginx 服务器
2. 修改 nginx.conf 配置文件

```
http {
    server {
        listen 80;
        server_name  localhost;
        location / {
            proxy_pass http://backend;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }
    }
```

```
    # 实现负载均衡
    upstream backend {
        server 127.0.0.1:8081;
        server 127.0.0.1:8082;
    }
}
```

- listen： 指定监听端口
- server_name：指定监听地址
- location：指定请求的路径
- proxy_pass： 指定目标服务器
- proxy_set_header： 指定传递给目标服务器的请求头信息，比如 Host，X-Real-IP，X-Forwarded-For 等，用来获取客户端的真实 IP

3. 重启 Nginx 服务

在上面的例子中，当客户端访问 http://localhost/（没加端口时默认是 80 端口），会采用轮询的机制将请求转发到 127.0.0.1:8081 和 127.0.0.1:8082 上，然后将结果返回给客户端

3、如何用 Nginx 做限流，有几种限流算法，分别如何实现?

官方解析

Nginx 可以通过配置限制每个客户端请求的速率来实现限流。具体来说，Nginx 有两种限流方式：**基于请求速率限制和基于连接速率限制。**

下面分别介绍这两种方式以及常用的限流算法。

1. 基于请求速率限制：基于请求速率限制是指限制每个客户端的请求速率，常用的限流算法有以下几种：
    - 漏桶算法：在单位时间内处理一定数量的请求，多余的请求则会放入一个“漏桶”中，随后以固定速率处理。
    - 令牌桶算法：在每个单位时间内，将一定数量的“令牌”放入桶中，每次请求需要获取一个令牌才能被处理，当桶中没有令牌时，请求将被拒绝。
    - 计数器算法：简单地对请求计数，并限制每个客户端在单位时间内最多可以处理的请求数量。

在 Nginx 中实现基于请求速率的限流通常需要使用模块，如 ngx_http_limit_req_module、ngx_http_limit_conn_module 等。

2. 基于连接速率限制：基于连接速率限制是指限制每个客户端的连接速率，常用的限流算法有以下几种：
    - 并发连接数：限制每个客户端同时能够建立的连接数，以此来限制连接速率。
    - 队列长度：将每个连接加入到一个队列中，限制队列中同时存在的连接数量，以此来限制连接速率。

在 Nginx 中实现基于连接速率的限流通常需要使用模块，如 ngx_http_limit_conn_module 等。

要实现基于请求速率或连接速率的限流，我们需要在 Nginx 配置文件中设置相应的限流规则。以下是一个基于漏桶算法的 Nginx 配置示例：

```
http {
    limit_req_zone $binary_remote_addr zone=one:10m rate=10r/s;

    server {
        location / {
            limit_req zone=one burst=20;
            proxy_pass http://backend;
        }
    }
}
```

其中，limit_req_zone 指定了一个名为 one 的共享内存区域，用于存储请求的统计信息，同时指定了速率为 10r/s。在 location 中，limit_req 指定了限流规则，并将请求转发到后端服务器。

需要注意的是，不同的限流算法适用于不同的场景，我们需要根据实际情况选择合适的算法和配置参数。同时，在实际应用中，为了防止恶意攻击和 DDOS 攻击，通常需要将多种限流算法组合起来使用。

鱼友的精彩回答

猿二哈的回答

Nginx的限流

1. Nginx的限流策略有两种：限制访问频率，限制并发连接数
2. 限制访问频率：limit_req_zone
3. 其实现为：limit_req_zone key zone rate
4. key表示定义限流的对象
5. zone表示定义共享内存区来存储访问信息
6. rate表示设置最大的访问速率
7. 限制并发连接数：limit_conn_zone
8. 其实现为：limit_conn_zone key zone
9. key表示定义限流的对象
10. zone表示定义共享区内存来存储访问信息

Nginx的限流算法有三种：计数器，漏桶算法，桶令牌算法

1. 计数器：计数器在接收一个请求时，会进行累加，累加有上限，不能超过这个值。超过这个值之后的请求会被拦截。计数器会周期性的置零，这就导致计数器算法存在临界问题：在计数器置零前后有大量的请求过来，此时可能会超过允许的最大请求数。
2. 漏桶算法：流进同的水滴的速率是随机的，流出桶的水滴的速率是固定的。当桶满时，水会溢出。这样就实现了限流的目的。应用场景有：流量整型，流量控制
3. 通令牌算法：放进桶中的令牌的速率是固定的，请求取出令牌的速率时随机的。当请求达到时，会先去桶中获取令牌。若桶中没有令牌，则请求等待。当桶中的令牌容量满时，新的令牌会被丢弃。

关键词：限制访问频率，限制并发连接数，key，zone，rate，共享内存区，计数器，临界问题，漏桶算法，桶令牌算法

前沿技术探索

1、什么是云原生？它有哪些优缺点?

官方解析

云原生是一种开发和运行应用程序的方法，**旨在利用云计算的弹性、可扩展性、可靠性和高可用性**等优势。它通过将应用程序打包到**容器**中，使用**容器编排工具**进行管理，实现了应用程序在不同环境中的**快速部署、弹性伸缩和高可用性**。

云原生的优点包括：

1. **灵活性和可扩展性**：容器可以快速部署和扩展，以满足不同的负载要求。
2. **高可用性**：容器编排工具可以自动检测和恢复容器故障，提供高可用性。
3. **效率和成本优化**：使用容器可以节省资源和成本，提高应用程序的运行效率。
4. **安全性**：容器隔离应用程序的运行环境，减少了安全漏洞的风险。

云原生的缺点包括：

1. **学习曲线较陡峭**：云原生技术较为复杂，需要学习一些新的技术和工具。
2. **可能存在依赖问题**：应用程序可能依赖于某些特定的云原生技术或工具，这可能导致一些限制或局限性。
3. **管理和维护难度**：容器编排工具可能需要额外的管理和维护，需要更多的操作和维护成本。

云原生的应用场景包括：

1. **微服务架构**：云原生技术非常适合构建微服务架构，将应用程序拆分为小型、自治的服务。
2. **弹性伸缩**：云原生技术可以根据应用程序的负载自动扩展或缩小容器的数量，以适应不同的负载要求。
3. **快速部署**：使用云原生技术，可以快速地部署和更新应用程序，提高开发和部署效率。
4. **多云部署**：云原生技术可以在多个云环境中运行，方便应用程序在不同云环境中的部署和迁移。
5. **数据处理和分析**：云原生技术可以处理大规模的数据处理和分析任务，提高数据处理效率。

> 鱼皮评论：这题可以在开头补充：什么是 “云”、什么是 “原生”，再去解释云原生的概念和好处

鱼友的精彩回答

HeiHei 的回答

云原生：

云原生是一种面向云计算架构的软件开发和部署方法论，旨在优化应用程序在云环境中的性能、弹性、可靠性和可扩展性。它强调使用容器、微服务、自动化管理和云原生编排工具来构建和管理应用程序，以适应云环境的动态性和复杂性。

云原生的优点：

1. **高可扩展性**：云原生技术可以帮助应用程序轻松扩展，使其能够在需要时快速适应业务增长或减少。
2. **高可用性和弹性**：云原生技术使用容器和微服务，可以使应用程序更加稳定和可靠。容器可以在单个节点或整个集群之间自由移动，从而在节点或集群发生故障时保持高可用性和弹性。
3. **更快的部署**：云原生技术可以通过自动化流程来加速软件部署，从而缩短开发到生产的时间。
4. **更好的资源利用率**：云原生技术可以帮助企业更好地利用其计算、存储和网络资源，从而提高效率并降低成本。

云原生的缺点：

1. **技术门槛高**：云原生技术需要一定的技术水平才能理解和应用，因此对于那些缺乏技术知识的企业而言可能较为困难。
2. **管理复杂**：使用云原生技术的应用程序可能需要更多的管理和监控，以确保其正常运行。这可能增加了管理的复杂性。
3. **安全性挑战**：容器和微服务可能增加了安全威胁，因此需要采取额外的安全措施来保护应用程序和数据。

2、你是否了解过新版本的 Java 特性？对 Java 未来的发展有什么看法?

官方解析

以下是一些值得关注的特性：

Java 8（2014年）：

- Lambda 表达式：简化函数式编程。
- Stream API：用于处理集合，支持函数式操作，如过滤、映射和聚合。
- 默认方法：在接口中提供默认实现，提高接口的灵活性。
- Optional 类：减少空指针异常，提高代码可读性。

Java 9（2017年）：

- 模块系统（Project Jigsaw）：将 Java 的庞大代码库划分为可重用的模块，简化大型应用的构建和维护。
- JShell：Java 的交互式命令行工具，用于快速尝试和测试 Java 代码片段。
- 新的集合工厂方法：方便地创建不可变集合，如 List.of()、Set.of() 和 Map.of()。

Java 10（2018年）：

- 局部变量类型推断：使用 var 关键字自动推断局部变量的类型，简化代码。
- 垃圾收集器接口改进：提高了垃圾收集器的可插拔性和灵活性。

Java 11（2018年，长期支持版本）：

- 新的 HTTP 客户端 API：支持 HTTP/2 和 WebSocket，提供了更现代化的编程方式。
- 改进的垃圾收集：引入了 ZGC 和 Epsilon 垃圾收集器。
- String 类的新方法：如 lines()、isBlank()、strip() 等。

Java 12 - 17 的部分新特性：

- Switch 表达式：简化 switch 语句的编写，支持使用箭头语法。
- 文本块：简化多行字符串字面量的表示。
- Records：简化数据类的定义，自动为其生成构造函数、getter、hashCode()、equals() 和 toString() 方法。
- Pattern Matching for instanceof（预览功能）：简化 instanceof 操作符的使用，避免显式类型转换。
- 密封类
- 删除实验性 AOT 和 JIT 编译器
- 特定于上下文的反序列化过滤器

Java 未来的发展趋势将是更加注重性能、安全和可靠性。以下是一些可能会影响 Java 未来发展的趋势：

1. 云计算和容器化：Java 非常适合在云环境中使用，因为它具有高度的可移植性和跨平台性。Java 的未来将会更加注重云计算和容器化的支持。
2. 数据科学和人工智能：Java 正在逐渐成为数据科学和人工智能领域的重要语言之一，未来 Java 可能会提供更多的支持和功能。
3. 静态类型检查：静态类型检查是提高代码质量和可靠性的重要手段，Java 未来可能会提供更加严格和强大的类型检查机制。
4. 简化语法：比如 Java 16 中的 Records 和 Sealed Classes 是一种更简洁的语法，未来 Java 可能会继续简化语法以提高代码的可读性和可维护性。

总之，Java 未来将继续发展和改进，以适应不断变化的编程环境和需求。

其实大家只需关注 LTS（长期支持版本）的 Java 8、11、17 就好了。

鱼友的精彩回答

Gundam 的回答

Java 8：

1. Lambda 表达式：允许以更简洁的语法编写函数式接口的实例，使代码更加简洁。
2. 流 API：提供了一种简单而高效的处理数据集合的方式，提高了代码的可读性和简洁性。
3. 方法引用：允许直接引用现有方法或构造函数，避免了重复编写类似的代码。
4. 接口默认方法：允许向现有接口添加默认方法实现，提供了更好的接口设计灵活性。
5. 时间 API：提供了一组强大的时间操作类，简化了日期和时间的操作。
6. 重复注解：允许在同一个地方多次声明同一个注解，提高了代码的可读性。
7. CompletableFuture 类：简化异步编程，提供更好的错误处理和异常处理机制。
8. Nashorn 引擎：提供了一种基于 JavaScript 的解决方案，允许将 JavaScript 代码嵌入到 Java 应用程序中。

Java 9：

1. 模块系统：引入了一种新的模块系统，提供更好的封装性和安全性。
2. 改进的 JShell 工具：提供了一种交互式编程的方式，可直接在控制台中执行 Java 代码。
3. 改进的 Stream API：提供了更多的流操作方法，使流操作更加强大和灵活。
4. Reactive Streams：提供了一种标准化的异步流处理框架。

5. 改进的 Optional 类：提供了更多的 API 方法和更好的 null 值处理机制。
6. HTTP 2 客户端：支持 HTTP/2 协议的客户端 API。
7. 集合工厂方法：提供了一种简化集合创建和初始化的方法。

Java 10：

1. 局部变量类型推断：允许在不显式声明变量类型的情况下，让编译器自动推断变量类型。
2. 线程局部变量回收：提供了一种新的垃圾回收器，可用于回收线程局部变量。
3. 应用类数据共享：允许多个 Java 应用程序共享类数据。
4. G1 垃圾回收器改进：提高了垃圾回收的效率和稳定性。
5. 基于时间的版本控制系统：允许使用时间作为版本号来管理代码版本。
6. 针对低内存设备的堆分配器：为低内存设备提供更好的内存管理机制。

Java 11：

1. HTTP 客户端 API：提供了一种标准化的 HTTP 客户端 API，支持异步和同步请求。
2. 嵌套访问控制：允许在一个类中定义另一个类，并限制访问权限，提高了代码的封装性和安全性。
3. 改进的字符串类：提供了一些新的方法，使得字符串的操作更加方便和高效。
4. Epsilon 垃圾回收器：提供了一种完全不执行垃圾回收的垃圾回收器，用于测试和性能分析。
5. ZGC 垃圾回收器：提供了一种低延迟的垃圾回收器，适用于大型堆内存的应用程序。
6. 应用程序类数据共享：允许不同的 Java 应用程序共享类元数据，提高了内存利用率和启动时间。
7. Unicode 10 支持：支持 Unicode 10 字符集和语言特性。

Java 12：

1. switch 表达式：允许在 switch 语句中使用表达式，提高了代码的可读性和简洁性。
2. 改进的字符串类：提供了一些新的方法，使得字符串的操作更加方便和高效。
3. Shenandoah 垃圾回收器：提供了一种低停顿时间的垃圾回收器，适用于大型堆内存的应用程序。
4. 微基准测试套件：提供了一种用于快速测试性能的微基准测试框架。
5. JDK 源代码重构：对 JDK 源代码进行了重构，提高了代码的可读性和维护性。

Java 13：

1. 文本块：允许以更简洁的语法创建多行字符串，提高了代码的可读性和简洁性。
2. 改进的 switch 表达式：允许在 switch 语句中使用表达式，提供更好的类型推断和更灵活的写法。
3. ZGC 垃圾回收器改进：提高了 ZGC 垃圾回收器的性能和可靠性。
4. 应用程序类数据共享改进：提高了应用程序类数据共享的性能和效率。

Java 14：

1. instanceof 模式匹配：允许在 instanceof 操作符中使用模式匹配，提高了代码的简洁性和可读性。
2. Records 类：提供了一种更简单和安全的数据类的定义方式。
3. Switch 表达式增强：允许使用箭头操作符(->)作为 lambda 表达式的简写语法。
4. 文本块增强：允许在文本块中使用嵌入式表达式，使得文本块更加灵活和强大。

5. 改进的 NullPointerException 信息：提供更详细的 NullPointerException 信息。

Java 15：

1. 隐式的类文件：允许在 Java 源代码中定义多个类，而不需要单独的类文件。
2. 改进的文本块：允许在文本块中使用转义字符和 Unicode 转义，提高了文本块的灵活性和可读性。
3. 改进的 switch 表达式：允许在 switch 语句中使用多个匹配项，提供更灵活的写法。
4. Sealed 类和接口：允许控制哪些类或接口可以继承或实现该类或接口，提高了代码的安全性和可维护性。
5. 其他改进：包括增强的 ZGC 垃圾回收器、改进的内存管理、新增的 Unix 域套接字 API 等。

Java 16：

1. 增强的文本块：允许在文本块中使用转义字符和嵌入式表达式。
2. 移除了废弃的 ParallelScavenge 垃圾回收器。
3. 改进的 ZGC 垃圾回收器：提高了性能和可靠性，增加了可配置参数。
4. Records 类的增强：允许在 records 类中添加静态方法和私有构造函数。
5. Vector API：提供了一种新的 API，用于高效地执行矢量化操作。

Java 17：

1. 嵌套枚举：允许在类和接口中定义嵌套枚举，提高了代码的可读性和简洁性。
2. 改进的 switch 语句：允许在 switch 语句中使用 case 标签作为表达式，提供更灵活的写法。
3. 预览性功能：包括模式匹配、嵌套枚举、记录类的序列化等新特性。
4. 增强的垃圾回收器：提高了性能和可靠性，增加了可配置参数。
5. 其他改进：包括新的内存管理和性能优化，增强的 JIT 编译器等。

发展趋势：

1. 云计算和大数据：随着云计算和大数据技术的迅猛发展，Java 在这些领域将继续扮演重要的角色。Java 在分布式计算和并行编程方面拥有丰富的经验和工具，能够帮助开发人员更好地利用云计算和大数据技术，实现高效的数据处理和分析。
2. 人工智能和机器学习：人工智能和机器学习是目前最热门的技术领域之一，Java 在这些领域也有不少的应用。未来，随着机器学习和深度学习技术的普及，Java 将继续为开发人员提供各种工具和框架，帮助他们构建高效和可扩展的机器学习应用。
3. 物联网：随着物联网技术的发展，Java 在这个领域也有很大的潜力。Java 具有跨平台性和高度可移植性，这些特性对于构建物联网应用至关重要。未来，Java 将继续提供各种工具和框架，帮助开发人员构建更加智能和高效的物联网应用。
4. 性能优化：Java 一直以来都是一门高性能的编程语言，未来 Java 将继续致力于性能优化。这包括优化 Java 虚拟机、垃圾回收器和其他核心组件，以及提供更好的工具和框架，帮助开发人员更好地优化他们的 Java 应用程序。
5. 安全性：随着网络攻击和数据泄露事件的不断增多，安全性已经成为了 Java 开发人员最关注的问题之一。未来，Java 将继续提供各种安全性功能和工具，帮助开发人员构建更加安全和可靠的 Java 应用程序。

> 作者：编程导航知识星球 + 星球鱼友们