

第一部分：Java语言

一、Java基础

其实过了萌新阶段，面试问基础就问的不多，但是保不齐突然问一下。想一下，总不能张口高并发、闭口分布式，结果什么是面向对象，说不清，那多少有点魔幻。所以赶紧来看看，这些基础有没有你不会的！

Java概述

1.什么是Java？



Java是世界上最好的语言！



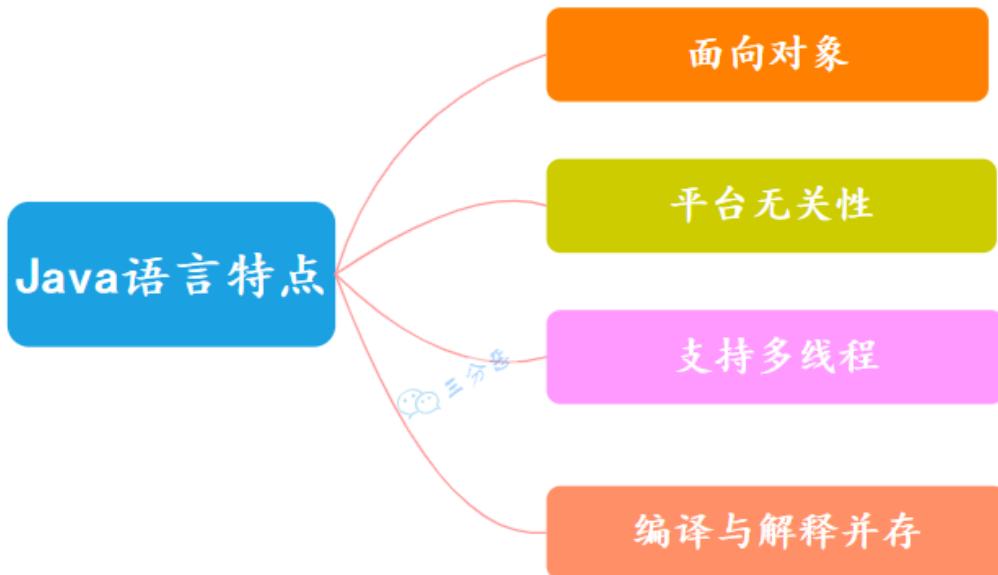
下辈子，还学Java!

PS：碎怂Java，有啥好介绍的。哦，面试啊。

Java是一门面向对象的编程语言，不仅吸收了C++语言的各种优点，还摒弃了C++里难以理解的多继承、指针等概念，因此Java语言具有功能强大和简单易用两个特征。Java语言作为静态面向对象编程语言的优秀代表，极好地实现了面向对象理论，允许程序员以优雅的思维方式进行复杂的编程。

2. Java语言有哪些特点？

Java语言有很多优秀（可吹）的特点，以下几个是比较突出的：



- 面向对象（封装，继承，多态）；
- 平台无关性，平台无关性的具体表现在于，Java 是“一次编写，到处运行（Write Once, Run Anywhere）”的语言，因此采用 Java 语言编写的程序具有很好的可移植性，而保证这一点的正是 Java 的虚拟机机制。在引入虚拟机之后，Java 语言在不同的平台上运行不需要重新编译。
- 支持多线程。C++ 语言没有内置的多线程机制，因此必须调用操作系统的多线程功能来进行多线程程序设计，而 Java 语言却提供了多线程支持；
- 编译与解释并存；

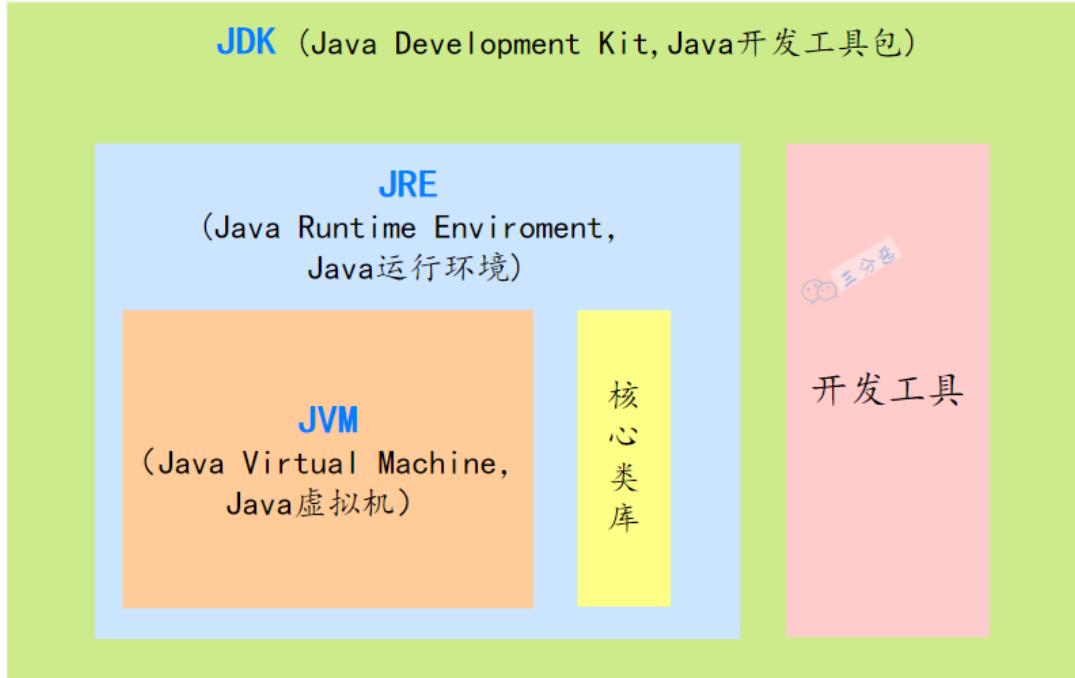
3. JVM、JDK 和 JRE 有什么区别？

JVM: Java Virtual Machine, Java虚拟机，Java程序运行在Java虚拟机上。针对不同系统的实现（Windows, Linux, macOS）不同的JVM，因此Java语言可以实现跨平台。

JRE: 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。

JDK: Java Development Kit，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。

简单来说，JDK包含JRE，JRE包含JVM。



4.说说什么是跨平台性？原理是什么？

所谓跨平台性，是指Java语言编写的程序，一次编译后，可以在多个系统平台上运行。

实现原理：Java程序是通过Java虚拟机在系统平台上运行的，只要该系统可以安装相应的Java虚拟机，该系统就可以运行java程序。

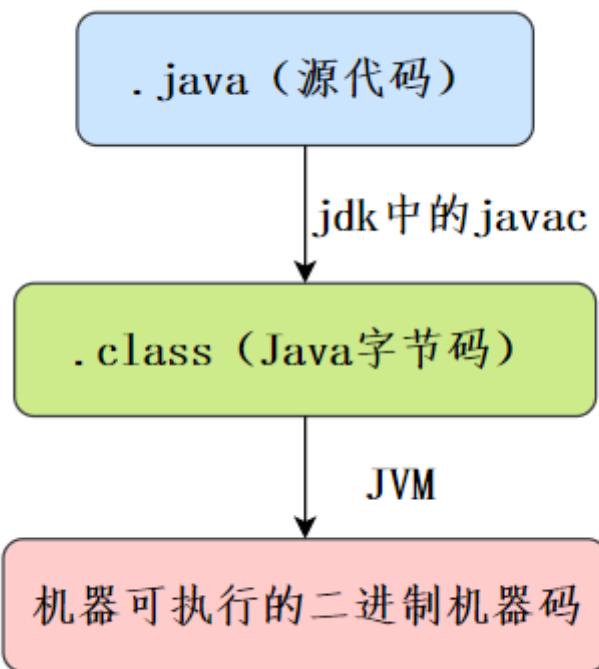
5.什么是字节码？采用字节码的好处是什么？

所谓的字节码，就是Java程序经过编译之类产生的.class文件，字节码能够被虚拟机识别，从而实现Java程序的跨平台性。

Java 程序从源代码到运行主要有三步：

- 编译：将我们的代码（.java）编译成虚拟机可以识别理解的字节码(.class)

- 解释：虚拟机执行Java字节码，将字节码翻译成机器能识别的机器码
- 执行：对应的机器执行二进制机器码



只需要把Java程序编译成Java虚拟机能识别的Java字节码，不同的平台安装对应的Java虚拟机，这样就可以实现Java语言的平台无关性。

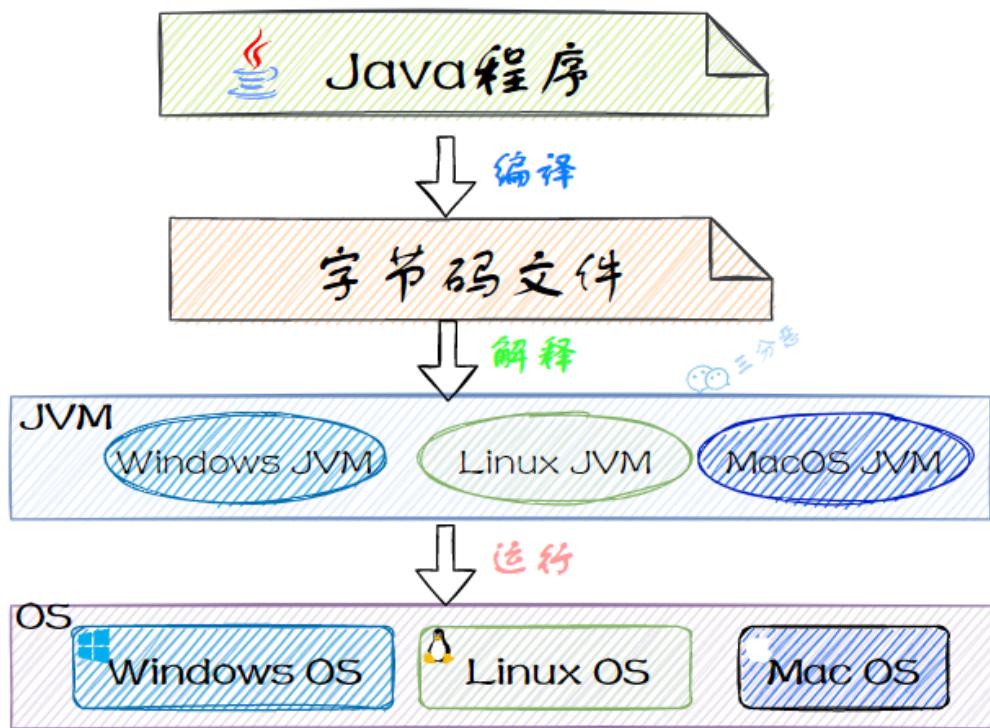
6.为什么说 Java 语言“编译与解释并存”？

高级编程语言按照程序的执行方式分为**编译型**和**解释型**两种。

简单来说，编译型语言是指编译器针对特定的操作系统将源代码一次性翻译成可被该平台执行的机器码；解释型语言是指解释器对源程序逐行解释成特定平台的机器码并立即执行。

比如，你想读一本外国的小说，你可以找一个翻译人员帮助你翻译，有两种选择方式，你可以先等翻译人员将全本的小说（也就是源码）都翻译成汉语，再去阅读，也可以让翻译人员翻译一段，你在旁边阅读一段，慢慢把书读完。

Java 语言既具有编译型语言的特征，也具有解释型语言的特征，因为 Java 程序要经过先编译，后解释两个步骤，由 Java 编写的程序需要先经过编译步骤，生成字节码（`*.class` 文件），这种字节码必须再经过JVM，解释成操作系统能识别的机器码，在由操作系统执行。因此，我们可以认为 Java 语言**编译与解释并存**。

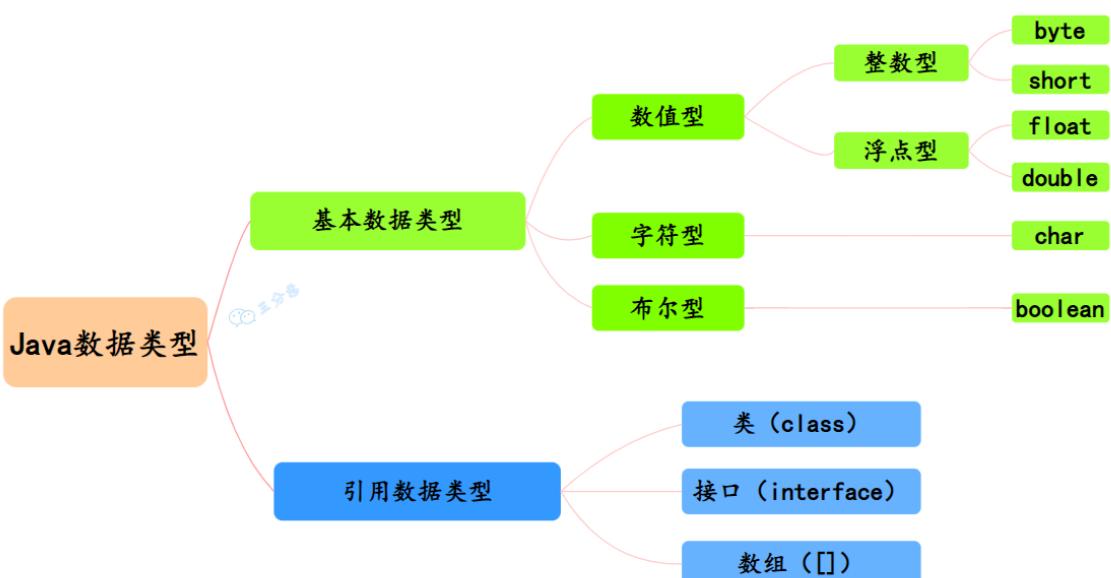


基础语法

7. Java有哪些数据类型？

定义：Java语言是强类型语言，对于每一种数据都定义了明确的具体的数据类型，在内存中分配了不同大小的内存空间。

Java语言数据类型分为两种：**基本数据类型**和**引用数据类型**。



基本数据类型：

- 数值型
 - 整数类型（byte、short、long）
 - 浮点类型（float、double）
- 字符型（char）
- 布尔型（boolean）

Java基本数据类型范围和默认值：

基本类型	位数	字节	默认值
int	32	4	0
short	16	2	0
long	64	8	0L
byte	8	1	0
char	16	2	'u0000'
float	32	4	0f
double	64	8	0d
boolean	1		false

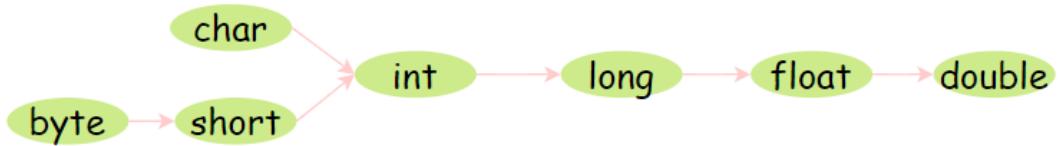
引用数据类型：

- 类（class）
- 接口（interface）
- 数组([])

8.自动类型转换、强制类型转换？看看这几行代码？

Java 所有的数值型变量可以相互转换，当把一个表数范围小的数值或变量直接赋给另一个表数范围大的变量时，可以进行自动类型转换；反之，需要强制转换。

自动类型转换方向



这就好像，小杯里的水倒进大杯没问题，但大杯的水倒进小杯就不行了，可能会溢出。

`float f=3.4`，对吗？

不正确。3.4 是单精度数，将双精度型（double）赋值给浮点型（float）属于下转型（down-casting，也称为窄化）会造成精度损失，因此需要强制类型转换 `float f = (float)3.4;` 或者写成 `float f = 3.4F`

`short s1 = 1; s1 = s1 + 1;` 对吗？ `short s1 = 1; s1 += 1;` 对吗？

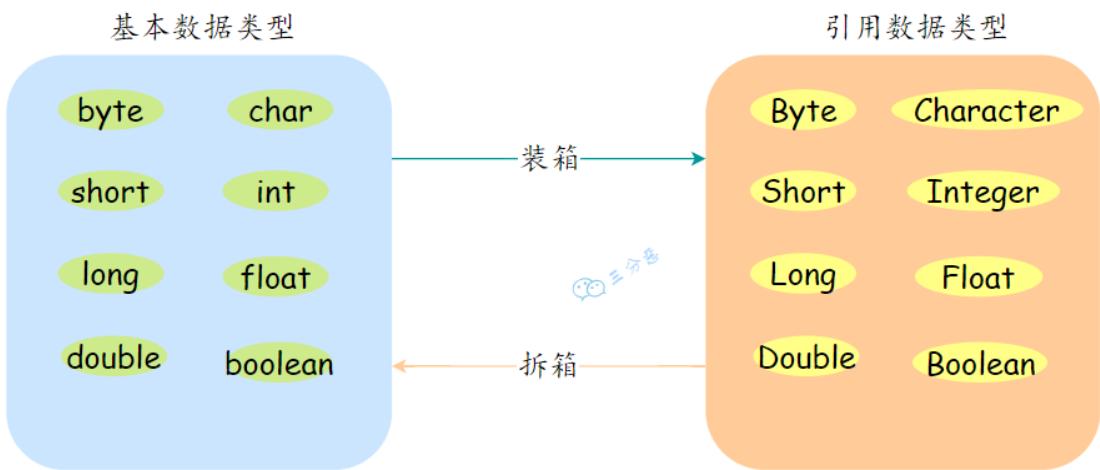
对于 `short s1 = 1; s1 = s1 + 1;` 编译出错，由于 1 是 int 类型，因此 `s1+1` 运算结果也是 int 型，需要强制转换类型才能赋值给 short 型。

而 `short s1 = 1; s1 += 1;` 可以正确编译，因为 `s1+=1;` 相当于 `s1 = (short(s1 + 1));` 其中有隐含的强制类型转换。

9.什么是自动拆箱/封箱？

- 装箱：将基本类型用它们对应的引用类型包装起来；
- 拆箱：将包装类型转换为基本数据类型；

Java 可以自动对基本数据类型和它们的包装类进行装箱和拆箱。



举例：

```

1 | Integer i = 10; //装箱
2 | int n = i;    //拆箱

```

10.&和&&有什么区别？

&运算符有两种用法：短路与、逻辑与。

&&运算符是短路与运算。逻辑与跟短路与的差别是非常巨大的，虽然二者都要求运算符左右两端的布尔值都是true 整个表达式的值才是 true。

&&之所以称为短路运算是因为，如果&&左边的表达式的值是 false，右边的表达式会被直接短路掉，不会进行运算。很多时候我们可能都需要用&&而不是&。

例如在验证用户登录时判定用户名不是 null 而且不是空字符串，应当写为

`username != null && !username.equals("")`，二者的顺序不能交换，更不能用&运算符，因为第一个条件如果不成立，根本不能进行字符串的 equals 比较，否则会产生 NullPointerException 异常。

注意：逻辑或运算符（|）和短路或运算符（||）的差别也是如此。

11.switch 是否能作用在 byte/long/String 上？

Java5 以前 switch(expr)中，expr 只能是 byte、short、char、int。

从 Java 5 开始，Java 中引入了枚举类型，expr 也可以是 enum 类型。

从 Java 7 开始，expr还可以是字符串(String)，但是长整型(long)在目前所有的版本中都是不可以的。

12.break ,continue ,return 的区别及作用？

- break 跳出整个循环，不再执行循环(结束当前的循环体)
- continue 跳出本次循环，继续执行下次循环(结束正在执行的循环 进入下一个循环条件)
- return 程序返回，不再执行下面的代码(结束当前的方法 直接返回)

```
public int circle() {  
    while (*) {  
        if (**){  
            return 100;  
        }  
        if (**){  
            continue;  
        }  
        if (**){  
            break;  
        }  
        int.....  
    }  
    return 0;  
}
```

13.用最有效率的方法计算2乘以8？

$2 \ll 3$ 。位运算，数字的二进制位左移三位相当于乘以2的三次方。

14.说说自增自减运算？看下这几个代码运行结果？

在写代码的过程中，常见的一种情况是需要某个整数类型变量增加 1 或减少 1，Java 提供了一种特殊的运算符，用于这种表达式，叫做自增运算符（`++`）和自减运算符（`--`）。

`++` 和 `--` 运算符可以放在变量之前，也可以放在变量之后。

当运算符放在变量之前时（前缀），先自增/减，再赋值；当运算符放在变量之后时（后缀），先赋值，再自增/减。

例如，当 `b = ++a` 时，先自增（自己增加 1），再赋值（赋值给 `b`）；当 `b = a++` 时，先赋值（赋值给 `b`），再自增（自己增加 1）。也就是，`++a` 输出的是 `a+1` 的值，`a++` 输出的是 `a` 值。

用一句口诀就是：“符号在前就先加/减，符号在后就后加/减”。

看一下这段代码运行结果？

```
1 | int i = 1;
2 | i = i++;
3 | System.out.println(i);
```

答案是 1。有点离谱对不对。

对于 JVM 而言，它对自增运算的处理，是会先定义一个临时变量来接收 `i` 的值，然后进行自增运算，最后又将临时变量赋给了值为 2 的 `i`，所以最后的结果为 1。

相当于这样的代码：

```
1 | int i = 1;
2 | int temp = i;
3 | i++;
4 | i = temp;
5 | System.out.println(i);
```

这段代码会输出什么？

```
1 int count = 0;
2 for(int i = 0;i < 100;i++)
3 {
4     count = count++;
5 }
6 System.out.println("count = "+count);
```

答案是0。

和上面的题目一样的道理，同样是用了临时变量，count实际是等于临时变量的值。

```
1 int autoAdd(int count)
2 {
3     int temp = count;
4     count = coutn + 1;
5     return temp;
6 }
```

PS：笔试面试可能会碰到的奇葩题，开发这么写，见一次吊一次。

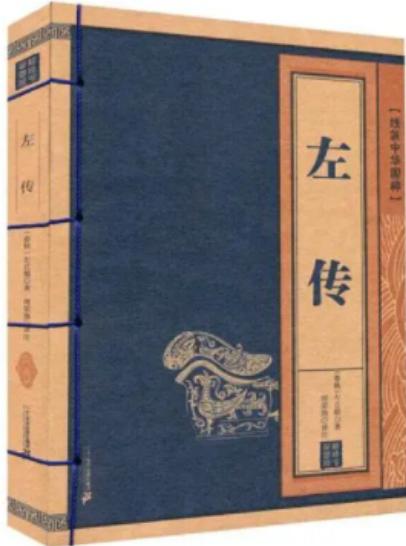
面向对象

15.面向对象和面向过程的区别?

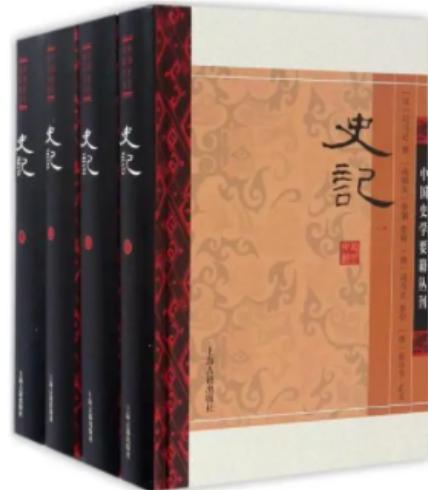
- 面向过程：面向过程就是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候再一个一个的一次调用就可以。
- 面向对象：面向对象，把构成问题的事务分解成各个对象，而建立对象的目的也不是为了完成一个个步骤，而是为了描述某个事件在解决整个问题的过程所发生的行为。目的是为了写出通用的代码，加强代码的重用，屏蔽差异性。

用一个比喻：面向过程是编年体；面向对象是纪传体。

面向过程：
编年体

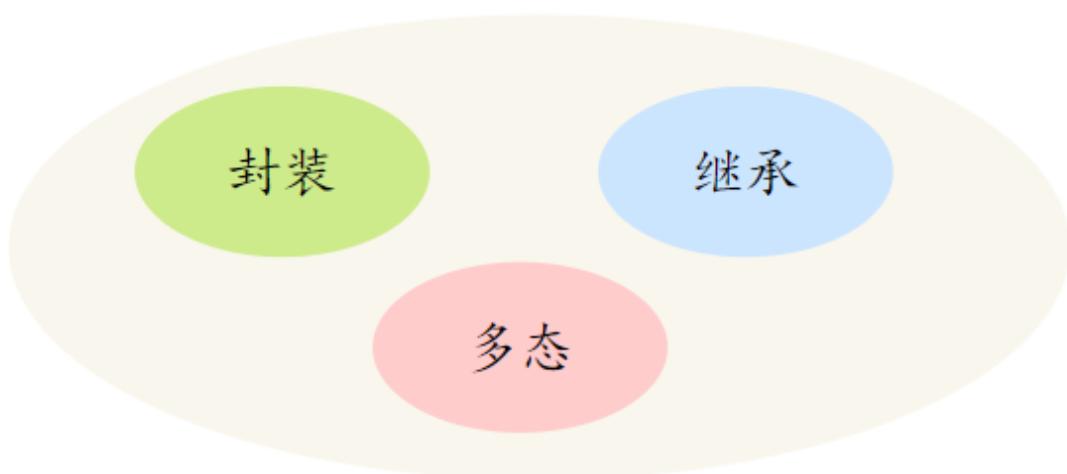


面向对象：
纪传体



16. 面向对象有哪些特性？

面向对象三大特征



- 封装

封装把一个对象的属性私有化，同时提供一些可以被外界访问的属性的方法。

- 继承

继承是使用已存在的类的定义作为基础创建新的类，新类的定义可以增加新的属性或新的方法，也可以继承父类的属性和方法。通过继承可以很方便地进行代码复用。

关于继承有以下三个要点：

1. 子类拥有父类对象所有的属性和方法（包括私有属性和私有方法），但是父类中的私有属性和方法子类是无法访问，只是拥有。
2. 子类可以拥有自己属性和方法，即子类可以对父类进行扩展。
3. 子类可以用自己的方式实现父类的方法。

- 多态

所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

在 Java 中有两种形式可以实现多态：继承（多个子类对同一方法的重写）和接口（实现接口并覆盖接口中同一方法）。

17. 重载（overload）和重写（override）的区别？

方法的重载和重写都是实现多态的方式，区别在于前者实现的是编译时的多态性，而后者实现的是运行时的多态性。

- 重载发生在一个类中，同名的方法如果有不同的参数列表（参数类型不同、参数个数不同或者二者都不同）则视为重载；
- 重写发生在子类与父类之间，重写要求子类被重写方法与父类被重写方法有相同的返回类型，比父类被重写方法更好访问，不能比父类被重写方法声明更多的异常（里氏代换原则）。

方法重载的规则：

1. 方法名一致，参数列表中参数的顺序，类型，个数不同。
2. 重载与方法的返回值无关，存在于父类和子类，同类中。
3. 可以抛出不同的异常，可以有不同修饰符。

18. 访问修饰符public、private、protected、以及不写（默认）时的区别？

Java中，可以使用访问控制符来保护对类、变量、方法和构造方法的访问。Java 支持 4 种不同的访问权限。

- default (即默认，什么也不写) : 在同一包内可见，不使用任何修饰符。可以修饰在类、接口、变量、方法。

- **private** : 在同一类内可见。可以修饰变量、方法。注意：不能修饰类（外部类）
- **public** : 对所有类可见。可以修饰类、接口、变量、方法
- **protected** : 对同一包内的类和所有子类可见。可以修饰变量、方法。注意：不能修饰类（外部类）。

可见性	private	default	protected	public
同一个类中	✓	✓	✓	✓
同一个包中	✗	✓	✓	✓
子类中	✗	✗	✓	✓
全局范围	✗	✗	✗	✓

19.this关键字有什么作用？

this是自身的一个对象，代表对象本身，可以理解为：指向对象本身的一个指针。

this的用法在Java中大体可以分为3种：

1. 普通的直接引用，this相当于是指向当前对象本身
2. 形参与成员变量名字重名，用this来区分：

```

1 | public Person(String name, int age){
2 |     this.name=name;
3 |     this.age=age;
4 |

```

3. 引用本类的构造函数

20.抽象类(abstract class)和接口(interface)有什么区别？

1. 接口的方法默认是public，所有方法在接口中不能有实现(Java 8开始接口方法可以有默认实现)，而抽象类可以有非抽象的方法。
2. 接口中除了static、final变量，不能有其他变量，而抽象类中则不一定。
3. 一个类可以实现多个接口，但只能实现一个抽象类。接口自己本身可以通过extends关键字扩展多个接口。

4. 接口方法默认修饰符是 public , 抽象方法可以有 public 、 protected 和 default 这些修饰符（抽象方法就是为了被重写所以不能使用 private 关键字修饰！）。
5. 从设计层面来说，抽象是对类的抽象，是一种模板设计，而接口是对行为的抽象，是一种行为的规范。

1. 在 JDK8 中，接口也可以定义静态方法，可以直接用接口名调用。实现类和实现是不可以调用的。如果同时实现两个接口，接口中定义了一样的默认方法，则必须重写，不然会报错。
2. jdk9 的接口被允许定义私有方法。

总结一下 jdk7~jdk9 Java 中接口的变化：

1. 在 jdk 7 或更早版本中，接口里面只能有常量变量和抽象方法。这些接口方法必须由选择实现接口的类实现。
2. jdk 8 的时候接口可以有默认方法和静态方法功能。
3. jdk 9 在接口中引入了私有方法和私有静态方法。

21. 成员变量与局部变量的区别有哪些？

1. 从语法形式上看：成员变量是属于类的，而局部变量是在方法中定义的变量或是方法的参数；成员变量可以被 public , private , static 等修饰符所修饰，而局部变量不能被访问控制修饰符及 static 所修饰；但是，成员变量和局部变量都能被 final 所修饰。
2. 从变量在内存中的存储方式来看：如果成员变量是使用 static 修饰的，那么这个成员变量是属于类的，如果没有使用 static 修饰，这个成员变量是属于实例的。对象存于堆内存，如果局部变量类型为基本数据类型，那么存储在栈内存，如果为引用数据类型，那存放的是指向堆内存对象的引用或者是指向常量池中的地址。
3. 从变量在内存中的生存时间上看：成员变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而自动消失。
4. 成员变量如果没有被赋初值：则会自动以类型的默认值而赋值（一种情况例外：被 final 修饰的成员变量也必须显式地赋值），而局部变量则不会自动赋值。

22. 静态变量和实例变量的区别？静态方法、实例方法呢？

静态变量和实例变量的区别？

静态变量: 是被 static 修饰符修饰的变量，也称为类变量，它属于类，不属于类的任何一个对象，一个类不管创建多少个对象，静态变量在内存中有且仅有一个副本。

实例变量: 必须依存于某一实例，需要先创建对象然后通过对象才能访问到它。静态变量可以实现让多个对象共享内存。

静态方法和实例方法有何不同?

类似地。

静态方法: static修饰的方法，也被称为类方法。在外部调用静态方法时，可以使用"类名.方法名"的方式，也可以使用"对象名.方法名"的方式。静态方法里不能访问类的非静态成员变量和方法。

实例方法: 依存于类的实例，需要使用"对象名.方法名"的方式调用；可以访问类的所有成员变量和方法。

24.final关键字有什么作用？

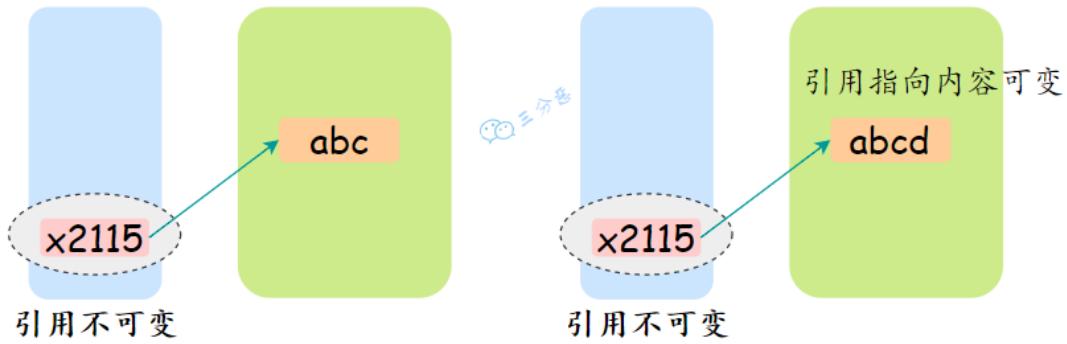
final表示不可变的意思，可用于修饰类、属性和方法：

- 被final修饰的类不可以被继承
- 被final修饰的方法不可以被重写
- 被final修饰的变量不可变，被final修饰的变量必须被显式第指定初始值，还得注意的是，这里的不可变指的是变量的引用不可变，不是引用指向的内容的不可变。

例如：

```
1 final StringBuilder sb = new StringBuilder("abc");
2 sb.append("d");
3 System.out.println(sb); //abcd
```

一张图说明：



25.final、finally、finalize的区别？

- final 用于修饰变量、方法和类：final修饰的类不可被继承；修饰的方法不可被重写；修饰的变量不可变。
- finally 作为异常处理的一部分，它只能在 `try/catch` 语句中，并且附带一个语句块表示这段语句最终一定被执行（无论是否抛出异常），经常被用在需要释放资源的情况下，`System.exit(0)` 可以阻断 finally 执行。
- finalize 是在 `java.lang.Object` 里定义的方法，也就是说每一个对象都有这么个方法，这个方法在 `gc` 启动，该对象被回收的时候被调用。

一个对象的 finalize 方法只会被调用一次， finalize 被调用不一定会立即回收该对象，所以有可能调用 finalize 后，该对象又不需要被回收了，然后到了真正要被回收的时候，因为前面调用过一次，所以不会再次调用 finalize 了，进而产生问题，因此不推荐使用 finalize 方法。

26.==和 equals 的区别？

`==`：它的作用是判断两个对象的地址是不是相等。即，判断两个对象是不是同一个对象(基本数据类型`==`比较的是值，引用数据类型`==`比较的是内存地址)。

`equals()`：它的作用也是判断两个对象是否相等。但是这个“相等”一般也分两种情况：

- 默认情况：类没有覆盖 `equals()` 方法。则通过 `equals()` 比较该类的两个对象时，等价于通过“`==`”比较这两个对象，还是相当于比较内存地址。
- 自定义情况：类覆盖了 `equals()` 方法。我们平时覆盖的 `equals()` 方法一般是比较两个对象的内容是否相同，自定义了一个相等的标准，也就是两个对象的值是否相等。

举个例子，Person，我们认为两个人的编号和姓名相同，就是一个人：

```
1 public class Person {  
2     private String no;  
3     private String name;  
4  
5     @Override  
6     public boolean equals(Object o) {  
7         if (this == o) return true;  
8         if (!(o instanceof Person)) return false;  
9         Person person = (Person) o;  
10        return Objects.equals(no, person.no) &&  
11                Objects.equals(name, person.name);  
12    }  
13  
14    @Override  
15    public int hashCode() {  
16        return Objects.hash(no, name);  
17    }  
18 }
```

27.hashCode与equals?

这个也是面试常问——“你重写过 hashCode 和 equals 么，为什么重写 equals 时必须重写 hashCode 方法？”

什么是HashCode?

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个 int 整数，定义在 Object 类中，是一个本地方法，这个方法通常用来将对象的内存地址转换为整数之后返回。

```
1 public native int hashCode();
```

哈希码主要在哈希表这类集合映射的时候用到，哈希表存储的是键值对(key-value)，它的特点是：能根据“键”快速的映射到对应的“值”。这其中就利用到了哈希码！

为什么要有 hashCode?

上面已经讲了，主要是在哈希表这种结构中用的到。

例如HashMap怎么把key映射到对应的value上呢？用的就是哈希取余法，也就是拿哈希码和存储元素的数组的长度取余，获取key对应的value所在的下标位置。详细可见：[面渣逆袭：Java集合连环三十问](#)

为什么重写 equals 时必须重写 hashCode 方法？

如果两个对象相等，则 hashCode 一定也是相同的。两个对象相等，对两个对象分别调用 equals 方法都返回 true。反之，两个对象有相同的 hashCode 值，它们也不一定是相等的。因此，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖。

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

为什么两个对象有相同的 hashCode 值，它们也不一定是相等的？

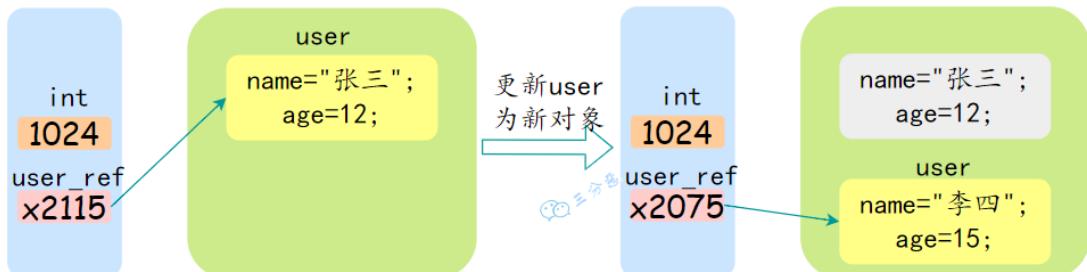
因为可能会碰撞， hashCode() 所使用的散列算法也许刚好会让多个对象传回相同的散列值。越糟糕的散列算法越容易碰撞，但这也与数据值域分布的特性有关（所谓碰撞也就是指的是不同的对象得到相同的 hashCode）。

28.Java是值传递，还是引用传递？

Java语言是值传递。Java 语言的方法调用只支持参数的值传递。当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。对象的属性可以在被调用过程中被改变，但对对象引用的改变是不会影响到调用者的。

JVM 的内存分为堆和栈，其中栈中存储了基本数据类型和引用数据类型实例的地址，也就是对象地址。

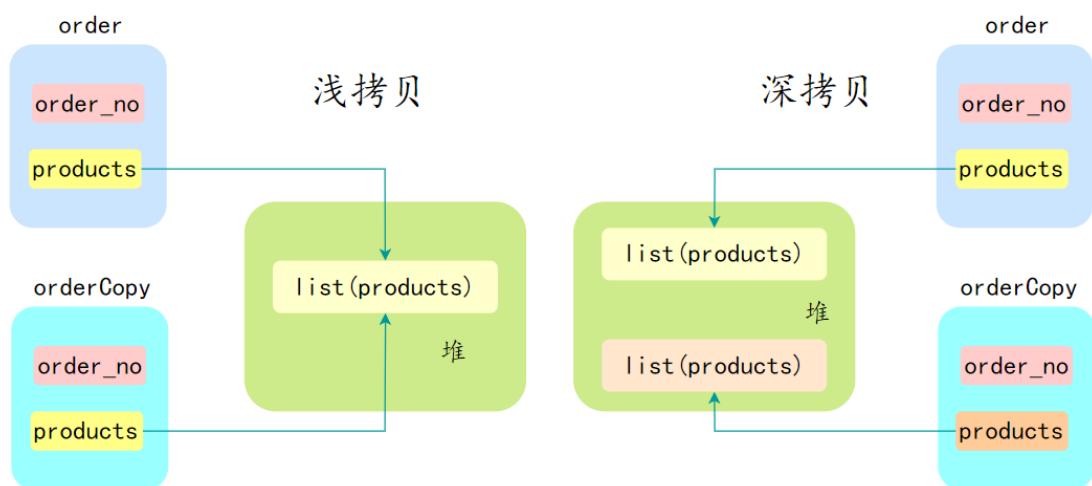
而对象所占的空间是在堆中开辟的，所以传递的时候可以理解为把变量存储的对象地址给传递过去，因此引用类型也是值传递。



29. 深拷贝和浅拷贝？

- 浅拷贝：仅拷贝被拷贝对象的成员变量的值，也就是基本数据类型变量的值，和引用数据类型变量的地址值，而对于引用类型变量指向的堆中的对象不会拷贝。
- 深拷贝：完全拷贝一个对象，拷贝被拷贝对象的成员变量的值，堆中的对象也会拷贝一份。

例如现在有一个order对象，里面有一个products列表，它的浅拷贝和深拷贝的示意图：



因此深拷贝是安全的，浅拷贝的话如果有引用类型，那么拷贝后对象，引用类型变量修改，会影响原对象。

浅拷贝如何实现呢？

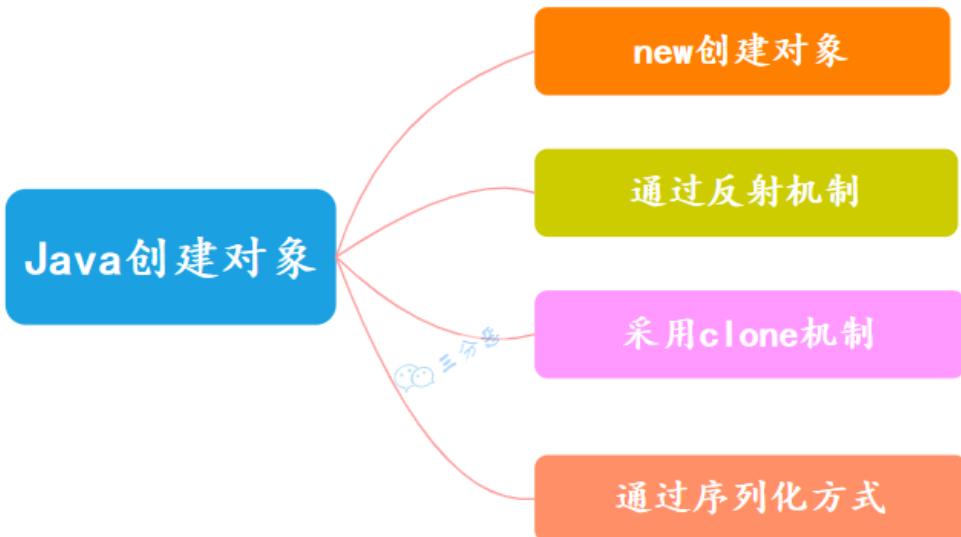
Object类提供的`clone()`方法可以非常简单地实现对象的浅拷贝。

深拷贝如何实现呢？

- 重写克隆方法：重写克隆方法，引用类型变量单独克隆，这里可能会涉及多层递归。
- 序列化：可以先讲原对象序列化，再反序列化成拷贝对象。

30. Java 创建对象有哪几种方式？

Java中有以下四种创建对象的方式：



- new创建新对象
- 通过反射机制
- 采用clone机制
- 通过序列化机制

前两者都需要显式地调用构造方法。对于clone机制,需要注意浅拷贝和深拷贝的区别,对于序列化机制需要明确其实现原理,在Java中序列化可以通过实现Externalizable或者Serializable来实现。

常用类

31.String 是 Java 基本数据类型吗? 可以被继承吗?

String是Java基本数据类型吗?

不是。Java 中的基本数据类型只有8个: byte、short、int、long、float、double、char、boolean; 除了基本类型 (primitive type), 剩下的都是引用类型 (reference type)。

String是一个比较特殊的引用数据类型。

String 类可以继承吗?

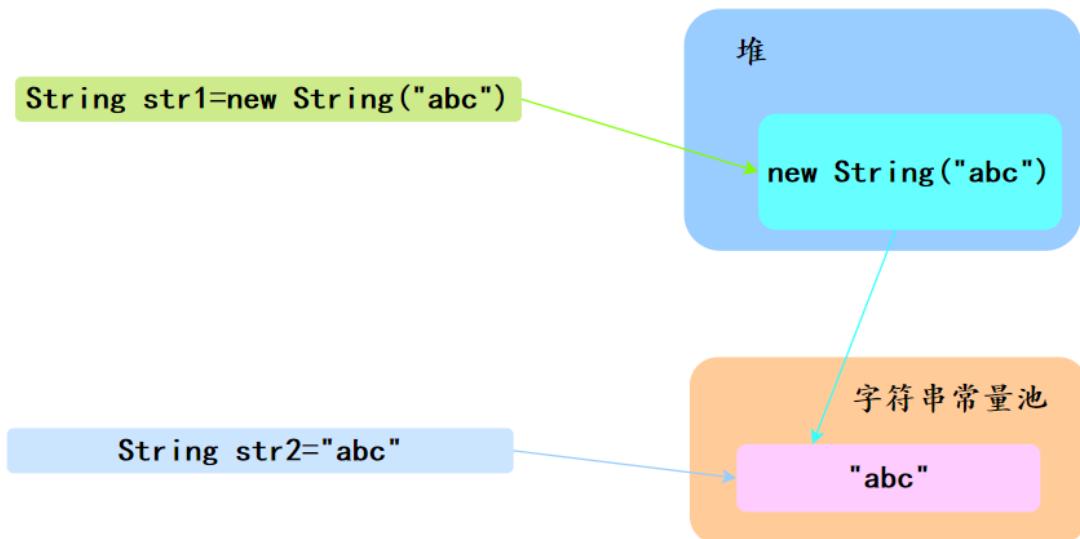
不行。String 类使用 final 修饰, 是所谓的不可变类, 无法被继承。

32.String和StringBuilder、 StringBuffer的区别？

- String: String 的值被创建后不能修改，任何对 String 的修改都会引发新的 String 对象的生成。
- StringBuffer: 跟 String 类似，但是值可以被修改，使用 synchronized 来保证线程安全。
- StringBuilder: StringBuffer 的非线程安全版本，性能上更高一些。

33.String str1 = new String("abc")和String str2 = "abc" 和 区别？

两个语句都会去字符串常量池中检查是否已经存在 “abc”，如果有则直接使用，如果没有则会在常量池中创建 “abc” 对象。



但是不同的是，`String str1 = new String("abc")` 还会通过 `new String()` 在堆里创建一个 “abc” 字符串对象实例。所以后者可以理解为被前者包含。

String s = new String("abc")创建了几个对象？

很明显，一个或两个。如果字符串常量池已经有“abc”，则是一个；否则，两个。

当字符创常量池没有 “abc”，此时会创建如下两个对象：

- 一个是字符串字面量 "abc" 所对应的、字符串常量池中的实例
- 另一个是通过 `new String()` 创建并初始化的，内容与"abc"相同的实例，在堆中。

34.String不是不可变类吗？字符串拼接是如何实现的？

String的确是不可变的，“+”的拼接操作，其实是会生成新的对象。

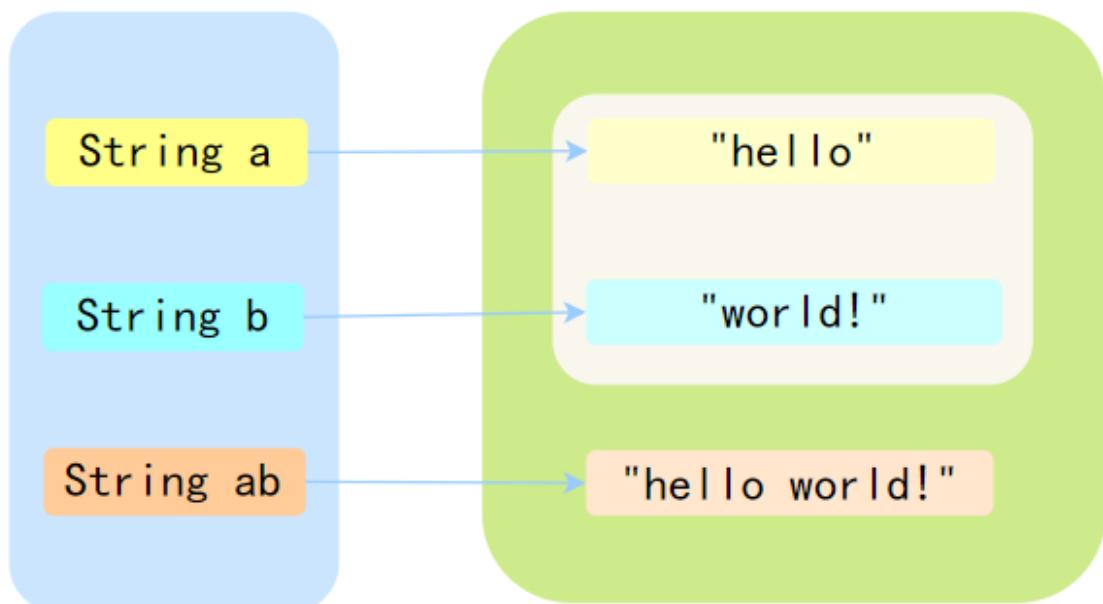
例如：

```
1 | String a = "hello ";
2 | String b = "world!";
3 | String ab = a + b;
```

在**jdk1.8之前**，a和b初始化时位于字符串常量池，ab拼接后的对象位于堆中。经过拼接新生成了String对象。如果拼接多次，那么会生成多个中间对象。

内存如下：

jdk1.8之前的字符串拼接



在**Java8时**JDK对“+”号拼接进行了优化，上面所写的拼接方式会被优化为基于StringBuilder的append方法进行处理。Java会在编译期对“+”号进行处理。

下面是通过javap -verbose命令反编译字节码的结果，很显然可以看到StringBuilder的创建和append方法的调用。

```
1 | stack=2, locals=4, args_size=1
2 |     0: ldc           #2                  // String hello
```

```
3   2: astore_1
4   3: ldc           #3                      // String world!
5   5: astore_2
6   6: new          #4                      // class
7   java/lang/StringBuilder
8   9: dup
9   10: invokespecial #5                   // Method
10  java/lang/StringBuilder."<init>":()V
11  13: aload_1
12  14: invokevirtual #6                  // Method
13  java/lang/StringBuilder.append:
14  (Ljava/lang/String;)Ljava/lang/StringBuilder;
15  17: aload_2
16  18: invokevirtual #6                  // Method
17  java/lang/StringBuilder.append:
18  (Ljava/lang/String;)Ljava/lang/StringBuilder;
19  21: invokevirtual #7                  // Method
20  java/lang/StringBuilder.toString:()Ljava/lang/String;
21  24: astore_3
22  25: return
```

也就是说其实上面的代码其实相当于：

```
1 String a = "hello ";
2 String b = "world!";
3 StringBuilder sb = new StringBuilder();
4 sb.append(a);
5 sb.append(b);
6 String ab = sb.toString();
```

此时，如果再笼统的回答：通过加号拼接字符串会创建多个String对象，因此性能比StringBuilder差，就是错误的了。因为本质上加号拼接的效果最终经过编译器处理之后和StringBuilder是一致的。

当然，循环里拼接还是建议用StringBuilder，为什么，因为循环一次就会创建一个新的StringBuilder对象，大家可以自行实验。

35.intern方法有什么作用？

JDK源码里已经对这个方法进行了说明：

```
1     * <p>
2         * When the intern method is invoked, if the pool already
3             contains a
4                 * string equal to this {@code String} object as
5                     determined by
6                         * the {@link #equals(Object)} method, then the string
7                             from the pool is
8                                 * returned. Otherwise, this {@code String} object is
9                                     added to the
10                                         * pool and a reference to this {@code String} object is
11                                             returned.
12
13     * <p>
```

意思也很好懂：

- 如果当前字符串内容存在于字符串常量池（即equals()方法为true，也就是内容一样），直接返回字符串常量池中的字符串
- 否则，将此String对象添加到池中，并返回String对象的引用

36.Integer a= 127, Integer b = 127; Integer c= 128, Integer d = 128; , 相等吗？

答案是a和b相等，c和d不相等。

- 对于基本数据类型==比较的值
- 对于引用数据类型==比较的是地址

Integer a= 127这种赋值，是用到了Integer自动装箱的机制。自动装箱的时候会去缓存池里取Integer对象，没有取到才会创建新的对象。

如果整型字面量的值在-128到127之间，那么自动装箱时不会new新的Integer对象，而是直接引用缓存池中的Integer对象，超过范围 a1==b1的结果是false

```
1  public static void main(String[] args) {
2      Integer a = 127;
3      Integer b = 127;
4      Integer b1 = new Integer(127);
5      System.out.println(a == b); //true
6      System.out.println(b==b1); //false
7
8      Integer c = 128;
9      Integer d = 128;
10     System.out.println(c == d); //false
11 }
```

什么是Integer缓存？

因为根据实践发现大部分的数据操作都集中在值比较小的范围，因此 Integer 搞了个缓存池，默认范围是 -128 到 127，可以根据通过设置 `JVM-XX:AutoBoxCacheMax=` 来修改缓存的最大值，最小值改不了。

实现的原理是int 在自动装箱的时候会调用Integer.valueOf，进而用到了 IntegerCache。

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

很简单，就是判断下值是否在缓存范围之内，如果是的话去 IntegerCache 中取，不是的话就创建一个新的Integer对象。

IntegerCache是一个静态内部类，在静态块中会初始化好缓存值。

```
1 private static class IntegerCache {
2     ....
3     static {
4         //创建Integer对象存储
5         for(int k = 0; k < cache.length; k++)
6             cache[k] = new Integer(j++);
7         ....
8     }
9 }
```

37.String怎么转成Integer的？原理？

PS:这道题印象中在一些面经中出场过几次。

String转成Integer，主要有两个方法：

- Integer.parseInt(String s)
- Integer.valueOf(String s)

不管哪一种，最终还是会调用Integer类内中的 `parseInt(String s, int radix)` 方法。

抛去一些边界之类的看看核心代码：

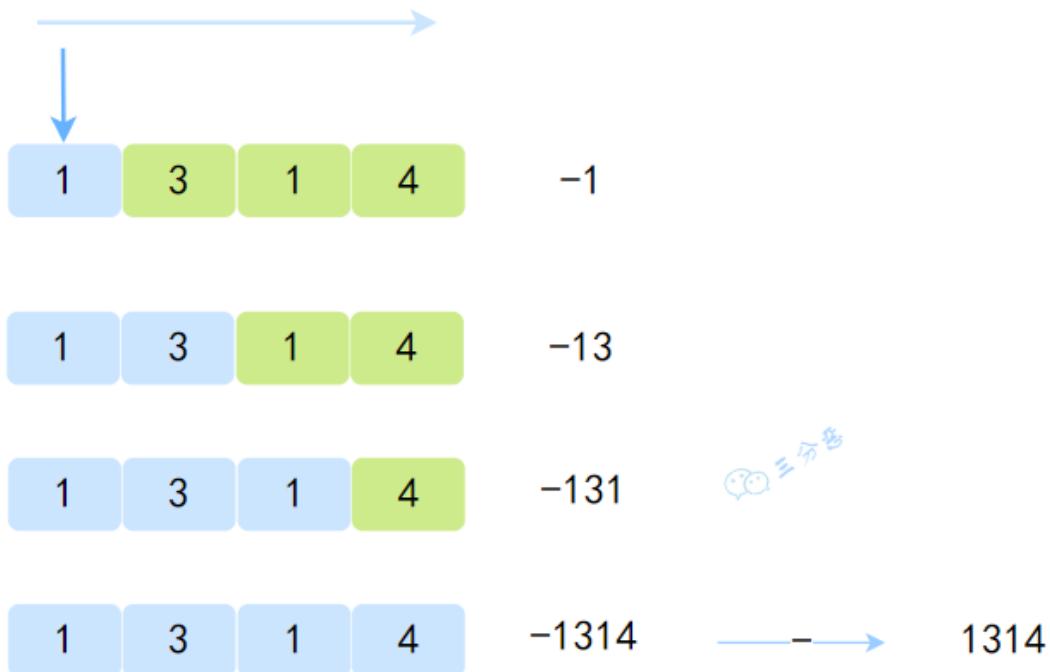
```
1 public static int parseInt(String s, int radix)
2                     throws NumberFormatException
3 {
4
5     int result = 0;
6     //是否是负数
7     boolean negative = false;
8     //char字符数组下标和长度
9     int i = 0, len = s.length();
10    ....
11    int digit;
12    //判断字符长度是否大于0, 否则抛出异常
13    if (len > 0) {
14        ....
15        while (i < len) {
16            // Accumulating negatively avoids surprises
17            near MAX_VALUE
```

```

17     //返回指定基数中字符表示的数值。 (此处是十进制数值)
18     digit = Character.digit(s.charAt(i++), radix);
19     //进制位乘以数值
20     result *= radix;
21     result -= digit;
22 }
23 }
24 //根据上面得到的是否负数, 返回相应的值
25 return negative ? result : -result;
26 }
27

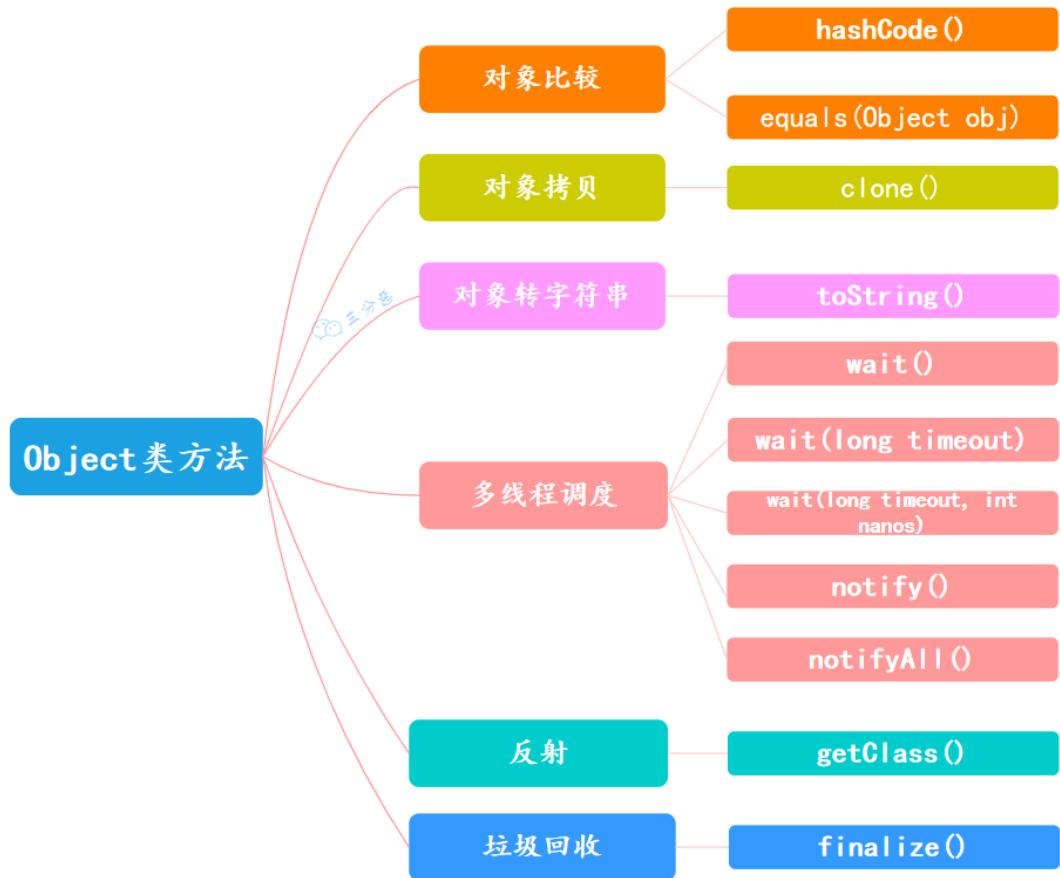
```

去掉枝枝蔓蔓（当然这些枝枝蔓蔓可以去看看，源码cover了很多情况），其实剩下的就是一个简单的字符串遍历计算，不过计算方式有点反常规，是用负的值累减。



38. Object 类的常见方法?

Object 类是一个特殊的类，是所有类的父类，也就是说所有类都可以调用它的方法。它主要提供了以下 11 个方法，大概可以分为六类：



对象比较：

- public native int hashCode(): native方法，用于返回对象的哈希码，主要使用在哈希表中，比如JDK中的HashMap。
- public boolean equals(Object obj): 用于比较2个对象的内存地址是否相等，String类对该方法进行了重写用户比较字符串的值是否相等。

对象拷贝：

- protected native Object clone() throws CloneNotSupportedException: native方法，用于创建并返回当前对象的一份拷贝。一般情况下，对于任何对象 x，表达式 x.clone() != x 为true，x.clone().getClass() == x.getClass() 为true。Object本身没有实现Cloneable接口，所以不重写clone方法并且进行调用的话会发生 CloneNotSupportedException 异常。

对象转字符串：

- public String toString(): 返回类的名字@实例的哈希码的16进制的字符串。建议 Object所有的子类都重写这个方法。

多线程调度：

- `public final native void notify()`: native方法，并且不能重写。唤醒一个在此对象监视器上等待的线程(监视器相当于就是锁的概念)。如果有多个线程在等待只会任意唤醒一个。
- `public final native void notifyAll()`: native方法，并且不能重写。跟notify一样，唯一的区别就是会唤醒在此对象监视器上等待的所有线程，而不是一个线程。
- `public final native void wait(long timeout) throws InterruptedException`: native方法，并且不能重写。暂停线程的执行。注意：sleep方法没有释放锁，而wait方法释放了锁。`timeout`是等待时间。
- `public final void wait(long timeout, int nanos) throws InterruptedException`: 多了`nanos`参数，这个参数表示额外时间（以毫微秒为单位，范围是 0-999999）。所以超时的时间还需要加上`nanos`毫秒。
- `public final void wait() throws InterruptedException`: 跟之前的2个wait方法一样，只不过该方法一直等待，没有超时时间这个概念

反射：

- `public final native Class<?> getClass()`: native方法，用于返回当前运行时对象的Class对象，使用了final关键字修饰，故不允许子类重写。

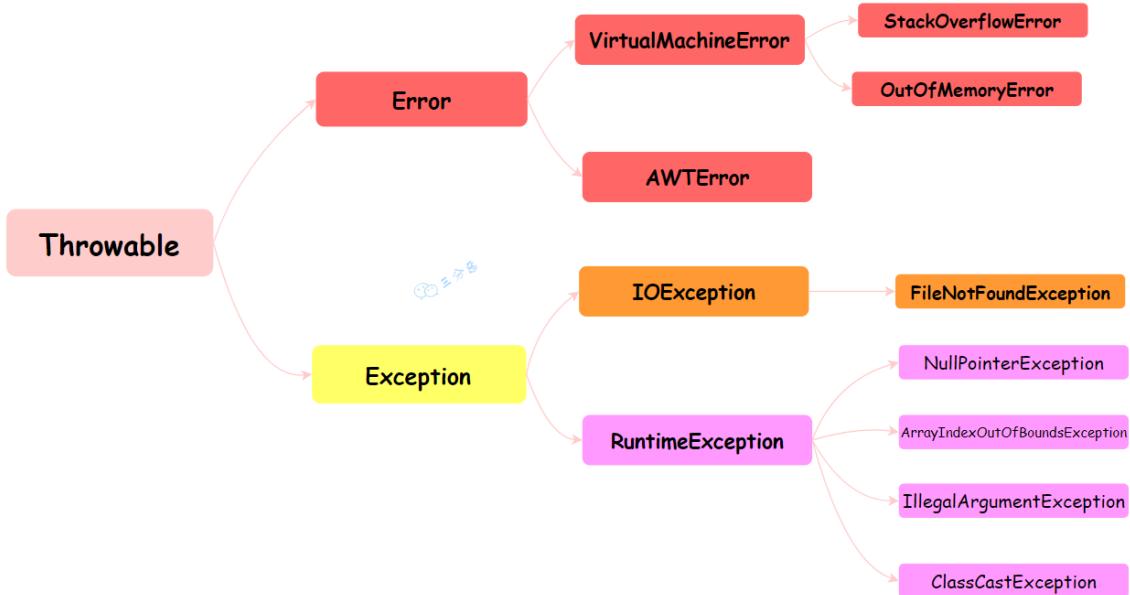
垃圾回收：

- `protected void finalize() throws Throwable` : 通知垃圾收集器回收对象。

异常处理

39. Java 中异常处理体系？

Java的异常体系是分为多层的。

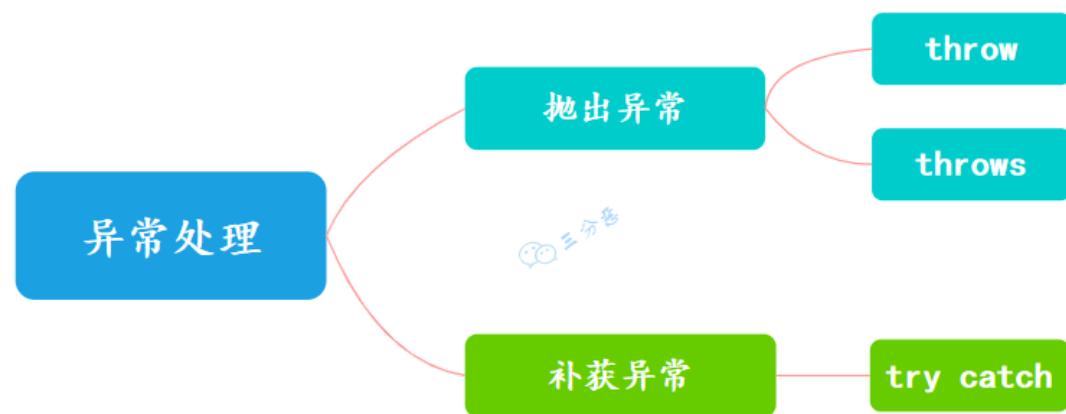


Throwable 是 Java 语言中所有错误或异常的基类。 **Throwable** 又分为 **Error** 和 **Exception**，其中 **Error** 是系统内部错误，比如虚拟机异常，是程序无法处理的。**Exception** 是程序问题导致的异常，又分为两种：

- **CheckedException** 受检异常：编译器会强制检查并要求处理的异常。
- **RuntimeException** 运行时异常：程序运行中出现异常，比如我们熟悉的空指针、数组下标越界等等

40. 异常的处理方式？

针对异常的处理主要有两种方式：



- 遇到异常不进行具体处理，而是继续抛给调用者（**throw**, **throws**）

抛出异常有三种形式，一是 **throw**, 一个 **throws**，还有一种系统自动抛异常。

throws 用在方法上，后面跟的是异常类，可以跟多个；而 **throw** 用在方法内，后面跟的是异常对象。

- try catch 捕获异常

在catch语句块中捕获发生的异常，并进行处理。

```
1   try {
2       //包含可能会出现异常的代码以及声明异常的方法
3   }catch(Exception e) {
4       //捕获异常并进行处理
5   }finally {
6       //可选，必执行的代码
7 }
```

try-catch捕获异常的时候还可以选择加上finally语句块，finally语句块不管程序是否正常执行，最终它都会必然执行。

41.三道经典异常处理代码题

题目1

```
1 public class TryDemo {
2     public static void main(String[] args) {
3         System.out.println(test());
4     }
5     public static int test() {
6         try {
7             return 1;
8         } catch (Exception e) {
9             return 2;
10        } finally {
11            System.out.print("3");
12        }
13    }
14 }
```

执行结果：31。

try、catch、finally 的基础用法，在 return 前会先执行 finally 语句块，所以是先输出 finally 里的 3，再输出 return 的 1。

题目2

```
1 public class TryDemo {  
2     public static void main(String[] args) {  
3         System.out.println(test1());  
4     }  
5     public static int test1() {  
6         try {  
7             return 2;  
8         } finally {  
9             return 3;  
10        }  
11    }  
12 }
```

执行结果：3。

try 返回前先执行 finally，结果 finally 里不按套路出牌，直接 return 了，自然也就走不到 try 里面的 return 了。

finally 里面使用 return 仅存在于面试题中，实际开发这么写要挨吊的。

题目3

```
1 public class TryDemo {  
2     public static void main(String[] args) {  
3         System.out.println(test1());  
4     }  
5     public static int test1() {  
6         int i = 0;  
7         try {  
8             i = 2;  
9             return i;  
10        } finally {  
11            i = 3;  
12        }  
13    }  
14 }
```

执行结果：2。

大家可能会以为结果应该是 3，因为在 return 前会执行 finally，而 i 在 finally 中被修改为 3 了，那最终返回 i 不是应该为 3 吗？

但其实，在执行 finally 之前，JVM 会先将 i 的结果暂存起来，然后 finally 执行完毕后，会返回之前暂存的结果，而不是返回 i，所以即使 i 已经被修改为 3，最终返回的还是之前暂存起来的结果 2。

I/O

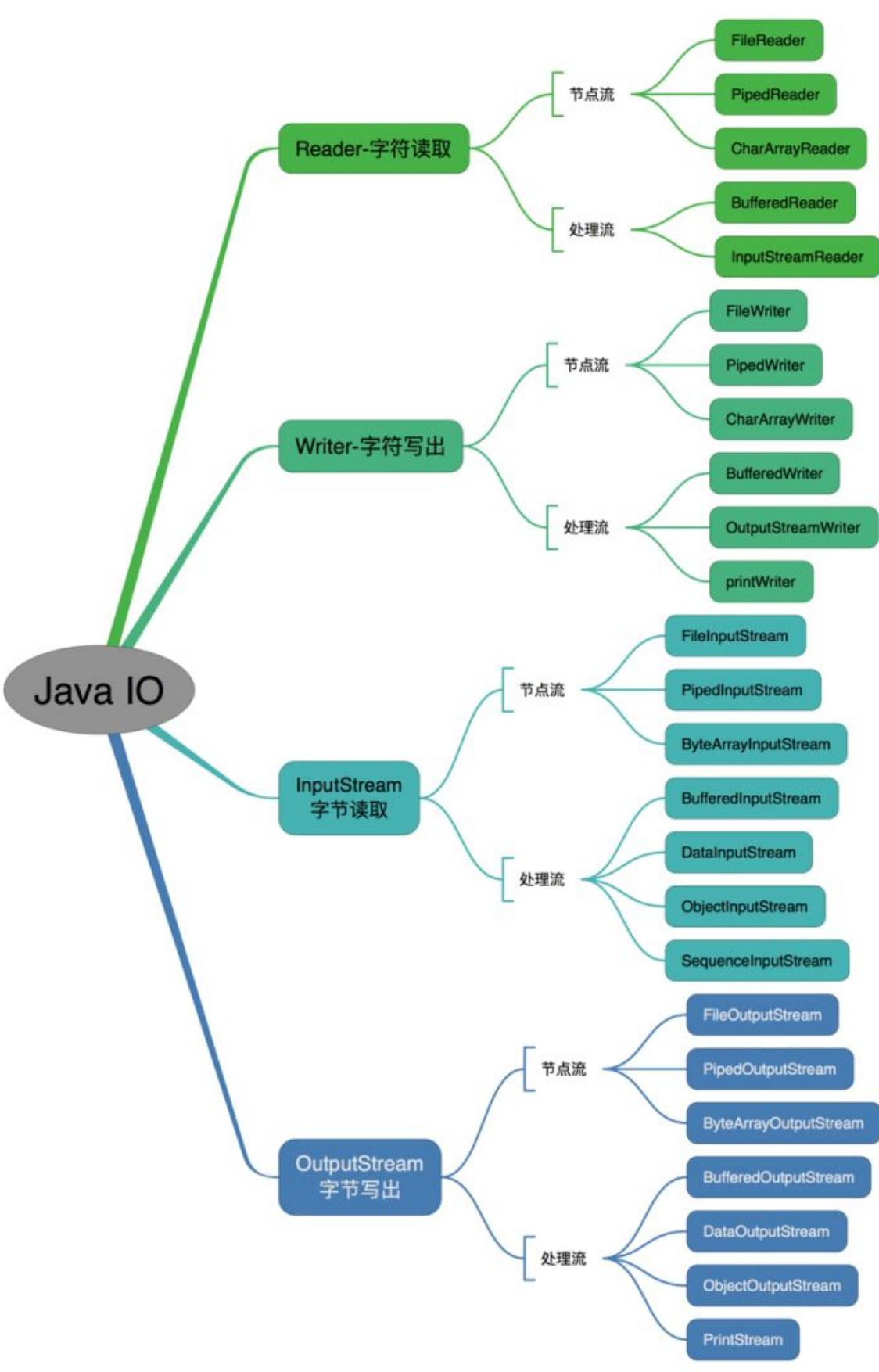
42. Java 中 IO 流分为几种？

流按照不同的特点，有很多种划分方式。

- 按照流的流向分，可以分为 输入流 和 输出流；
- 按照操作单元划分，可以划分为 字节流 和 字符流；
- 按照流的角色划分为 节点流 和 处理流

Java Io流共涉及40多个类，看上去杂乱，其实都存在一定的关联，Java I0流的40多个类都是从如下4个抽象类基类中派生出来的。

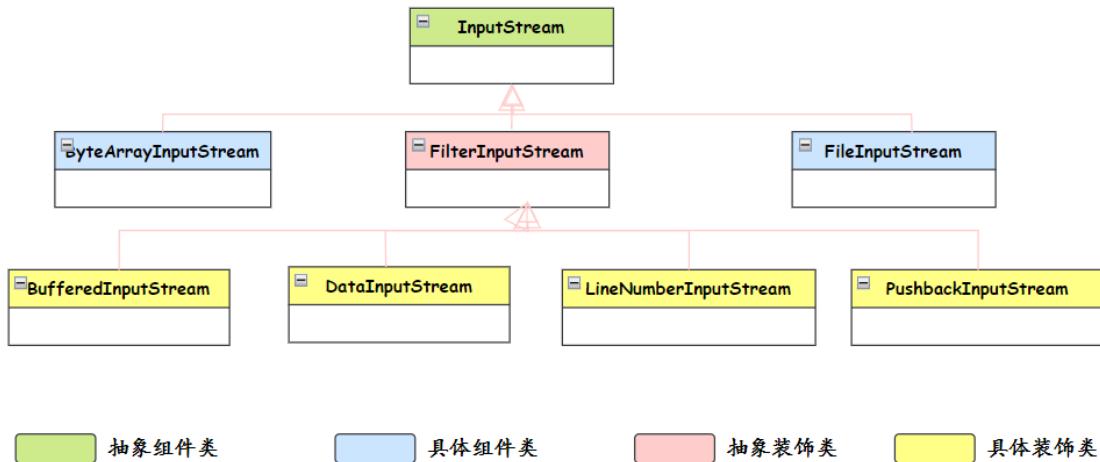
- InputStream / Reader : 所有的输入流的基类，前者是字节输入流，后者是字符输入流。
- OutputStream / Writer : 所有输出流的基类，前者是字节输出流，后者是字符输出流。



IO流用到了什么设计模式？

其实，Java的IO流体系还用到了一个设计模式——**装饰器模式**。

InputStream相关的部分类图如下，篇幅有限，装饰器模式就不展开说了。

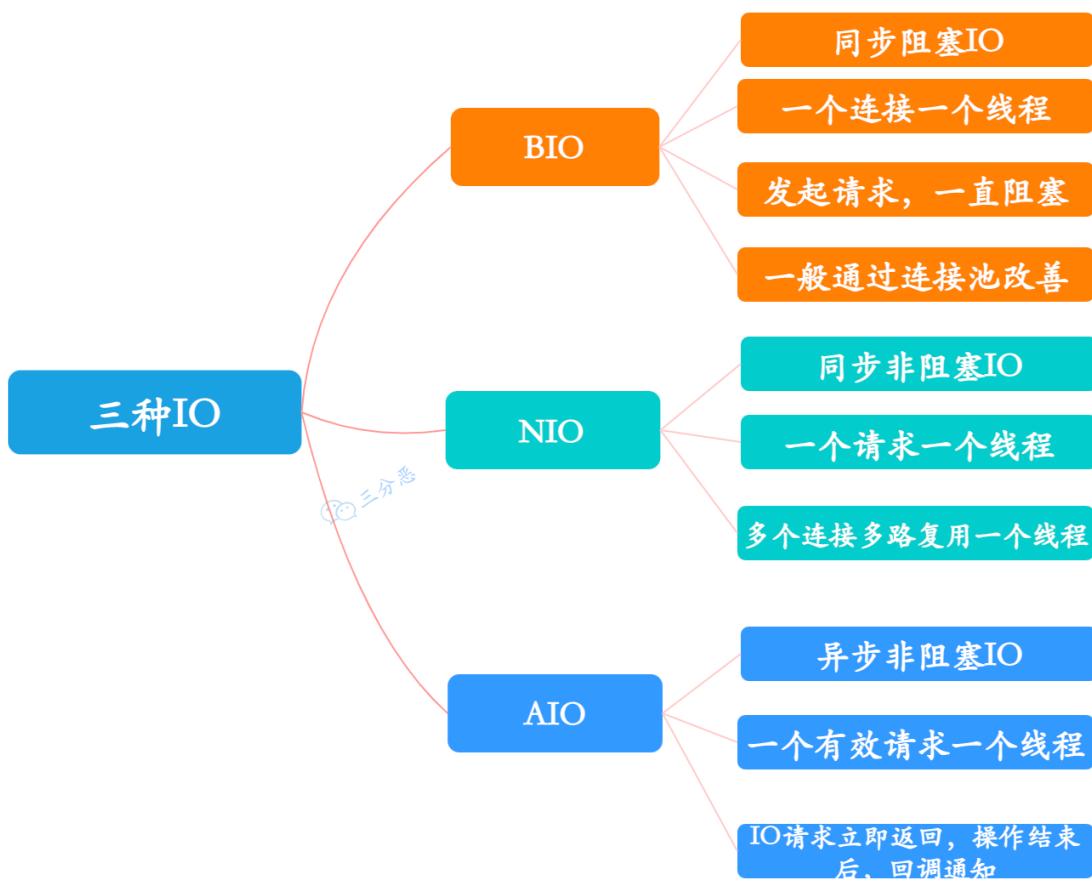


43.既然有了字节流,为什么还要有字符流?

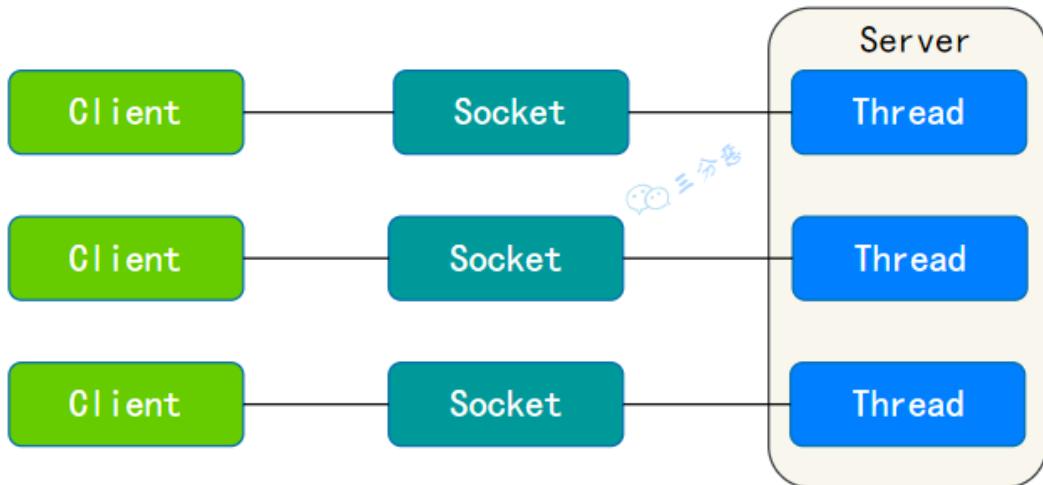
其实字符流是由 Java 虚拟机将字节转换得到的，问题就出在这个过程还比较耗时，并且，如果我们不知道编码类型就很容易出现乱码问题。

所以，I/O 流就干脆提供了一个直接操作字符的接口，方便我们平时对字符进行流操作。如果音频文件、图片等媒体文件用字节流比较好，如果涉及到字符的话使用字符流比较好。

44.BIO、NIO、AIO?



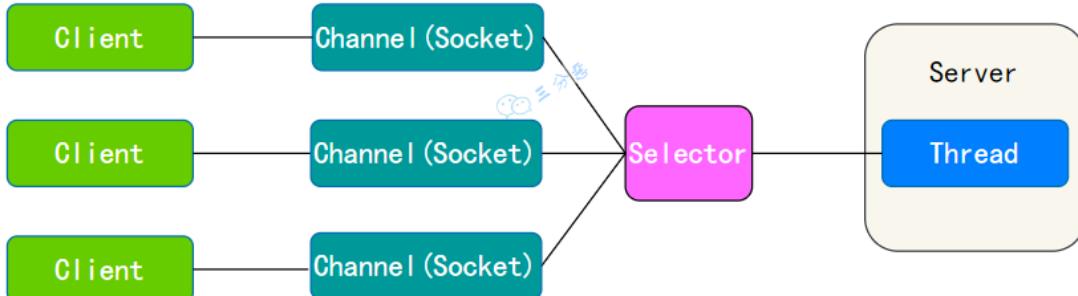
BIO(blocking I/O)：就是传统的IO，同步阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，可以通过连接池机制改善(实现多个客户连接服务器)。



BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4 以前的唯一选择，程序简单易理解。

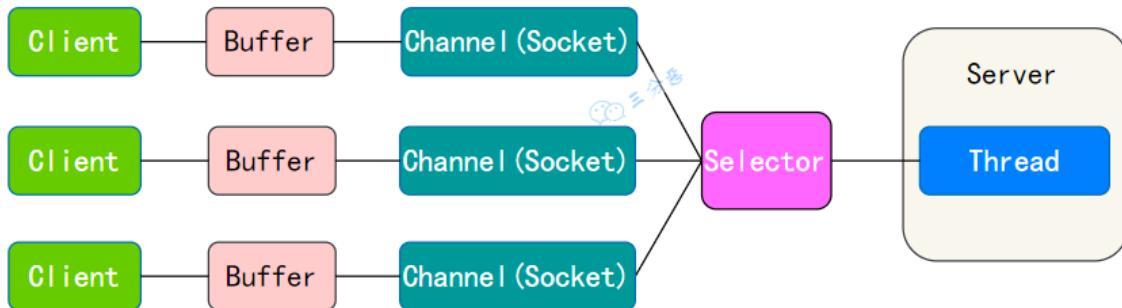
NIO：全称 java non-blocking IO，是指 JDK 提供的新 API。从JDK1.4开始，Java 提供了一系列改进的输入/输出的新特性，被统称为NIO(即New IO)。

NIO是**同步非阻塞**的，服务器端用一个线程处理多个连接，客户端发送的连接请求会注册到多路复用器上，多路复用器轮询到连接有IO请求就进行处理：



NIO的数据是面向**缓冲区Buffer**的，必须从Buffer中读取或写入。

所以完整的NIO示意图：



可以看出，NIO的运行机制：

- 每个Channel对应一个Buffer。
- Selector对应一个线程，一个线程对应多个Channel。
- Selector会根据不同的事件，在各个通道上切换。
- Buffer是内存块，底层是数据。

AOI：JDK 7 引入了 Asynchronous I/O，是**异步不阻塞**的 IO。在进行 I/O 编程中，常用到两种模式：Reactor 和 Proactor。Java 的 NIO 就是 Reactor，当有事件触发时，服务器端得到通知，进行相应的处理，完成后才通知服务端程序启动线程去处理，一般适用于连接数较多且连接时间较长的应用。

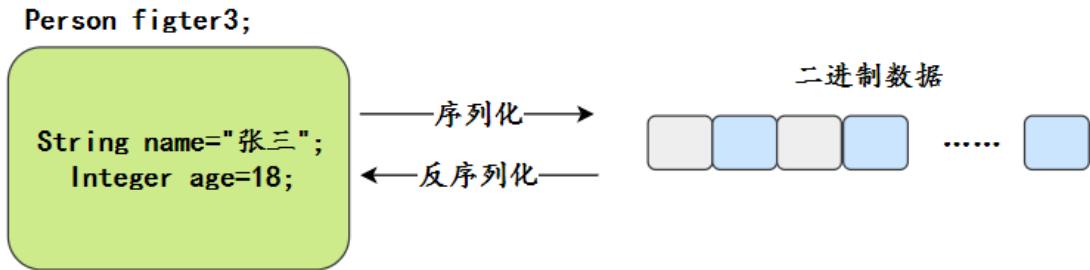
PS：关于同步阻塞IO、同步不阻塞IO、异步不阻塞IO的相关概念可以查看：[面试章节，被操作系统问挂了](#)

序列化

45.什么是序列化？什么是反序列化？

什么是序列化，序列化就是**把Java对象转为二进制流**，方便存储和传输。

所以**反序列化**就是**把二进制流恢复成对象**。



类比我们生活中一些大件物品的运输，运输的时候把它拆了打包，用的时候再拆包组装。

Serializable接口有什么用？

这个接口只是一个标记，没有具体的作用，但是如果不能实现这个接口，在有些序列化场景会报错，所以一般建议，创建的JavaBean类都实现 Serializable。

serialVersionUID 又有什么用？

serialVersionUID 就是起验证作用。

```
1 | private static final long serialVersionUID = 1L;
```

我们经常会看到这样的代码，这个 ID 其实就是用来验证序列化的对象和反序列化对应的对象ID 是否一致。

这个 ID 的数字其实不重要，无论是 1L 还是 IDE自动生成的，只要序列化时候对象的 serialVersionUID 和反序列化时候对象的 serialVersionUID 一致的话就行。

如果没有显示指定 serialVersionUID，则编译器会根据类的相关信息自动生成一个，可以认为是一个指纹。

所以如果你没有定义一个 serialVersionUID，结果序列化一个对象之后，在反序列化之前把对象的类的结构改了，比如增加了一个成员变量，则此时的反序列化会失败。

因为类的结构变了，所以 serialVersionUID 就不一致。

Java 序列化不包含静态变量？

序列化的时候是不包含静态变量的。

如果有些变量不想序列化，怎么办？

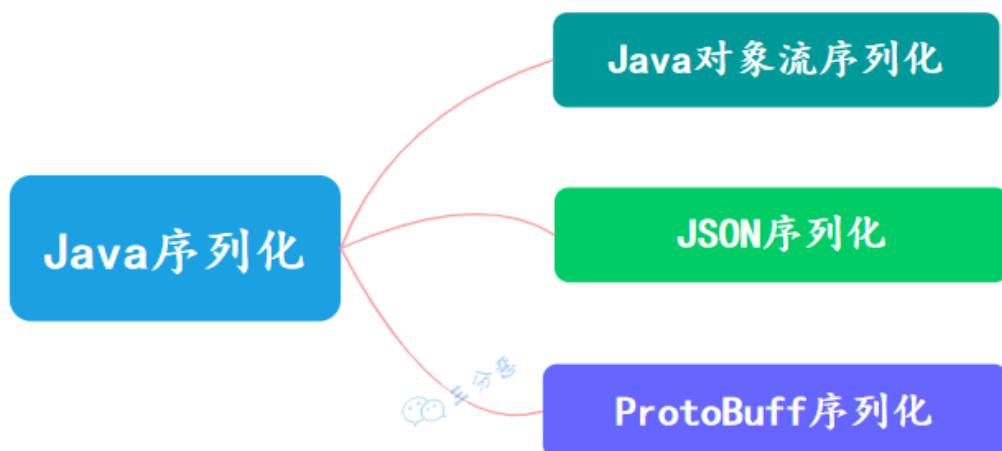
对于不想进行序列化的变量，使用 **transient** 关键字修饰。

transient 关键字的作用是：阻止实例中那些用此关键字修饰的变量序列化；当对象被反序列化时，被 **transient** 修饰的变量值不会被持久化和恢复。

transient 只能修饰变量，不能修饰类和方法。

46. 说说有几种序列化方式？

Java序列化方式有很多，常见的有三种：



- Java对象流序列化：Java原生序列化方法即通过Java原生流(InputStream和OutputStream之间的转化)的方式进行转化，一般是对象输出流 **ObjectOutputStream** 和对象输入流 **ObjectInputStream**。
- JSON序列化：这个可能是我们最常用的序列化方式，JSON序列化的选择很多，一般会使用jackson包，通过 ObjectMapper类来进行一些操作，比如将对象转化为byte数组或者将json串转化为对象。

- ProtoBuff序列化：ProtocolBuffer是一种轻便高效的结构化数据存储格式，ProtoBuff序列化对象可以很大程度上将其压缩，可以大大减少数据传输大小，提高系统性能。

泛型

47. Java 泛型了解么？什么是类型擦除？介绍一下常用的通配符？

什么是泛型？

Java 泛型（generics）是 JDK 5 中引入的一个新特性，泛型提供了编译时类型安全检测机制，该机制允许程序员在编译时检测到非法的类型。泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。

```
1 List<Integer> list = new ArrayList<>();
2
3 list.add(12);
4 //这里直接添加会报错
5 list.add("a");
6 Class<? extends List> clazz = list.getClass();
7 Method add = clazz.getDeclaredMethod("add", Object.class);
8 //但是通过反射添加，是可以的
9 add.invoke(list, "k1");
10
11 System.out.println(list);
```

泛型一般有三种使用方式：**泛型类、泛型接口、泛型方法**。

泛型类

public

class

ClassName

<T>

泛型接口

public

interface

InterfaceName

<T>

泛型方法

public

static

<T>

ReturnType

functionName

public

<T>

ReturnType

functionName

1. 泛型类：

```
1 //此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型
2 //在实例化泛型类时，必须指定T的具体类型
3 public class Generic<T>{
4
5     private T key;
6
7     public Generic(T key) {
8         this.key = key;
9     }
10
11    public T getKey(){
12        return key;
13    }
14}
```

如何实例化泛型类：

```
1 Generic<Integer> genericInteger = new Generic<Integer>
(123456);
```

2. 泛型接口：

```
1 class GeneratorImpl<T> implements Generator<T>{
2     @Override
3     public T method() {
4         return null;
5     }
6 }
```

实现泛型接口，指定类型：

```
1 class GeneratorImpl<T> implements Generator<String>{
2     @Override
3     public String method() {
4         return "hello";
5     }
6 }
```

3.泛型方法：

```
1 public static < E > void printArray( E[] inputArray )
2 {
3     for ( E element : inputArray ){
4         System.out.printf( "%s ", element );
5     }
6     System.out.println();
7 }
```

使用：

```
1 // 创建不同类型数组： Integer, Double 和 Character
2 Integer[] intArray = { 1, 2, 3 };
3 String[] stringArray = { "Hello", "World" };
4 printArray( intArray );
5 printArray( stringArray );
```

泛型常用的通配符有哪些？

常用的通配符为： **T, E, K, V, ?**

- **?** 表示不确定的 java 类型

- T (type) 表示具体的一个 java 类型
- K V (key value) 分别代表 java 键值中的 Key Value
- E (element) 代表 Element

什么是泛型擦除？

所谓的泛型擦除，官方名叫“类型擦除”。

Java 的泛型是伪泛型，这是因为 Java 在编译期间，所有的类型信息都会被擦掉。

也就是说，在运行的时候是没有泛型的。

例如这段代码，往一群猫里放条狗：

```
1 | LinkedList<Cat> cats = new LinkedList<Cat>();
2 | LinkedList list = cats; // 注意我在这里把范型去掉了，但是list和
| cats是同一个链表！
3 | list.add(new Dog()); // 完全没问题！
```

因为Java的范型只存在于源码里，编译的时候给你静态地检查一下范型类型是否正确，而到了运行时就不检查了。上面这段代码在JRE（Java运行环境）看来和下面这段没区别：

```
1 | LinkedList cats = new LinkedList(); // 注意：没有范型！
2 | LinkedList list = cats;
3 | list.add(new Dog());
```

为什么要类型擦除呢？

主要是为了向下兼容，因为JDK5之前是没有泛型的，为了让JVM保持向下兼容，就出了类型擦除这个策略。

注解

48.说一下你对注解的理解？

Java注解本质上是一个标记，可以理解成生活中的一个人的一些小装扮，比如戴什么什么帽子，戴什么眼镜。



注解可以标记在类上、方法上、属性上等，标记自身也可以设置一些值，比如帽子颜色是绿色。

有了标记之后，我们就可以在编译或者运行阶段去识别这些标记，然后搞一些事情，这就是注解的用处。

例如我们常见的AOP，使用注解作为切点就是运行期注解的应用；比如lombok，就是注解在编译期的运行。

注解生命周期有三大类，分别是：

- RetentionPolicy.SOURCE: 给编译器用的，不会写入 class 文件
- RetentionPolicy.CLASS: 会写入 class 文件，在类加载阶段丢弃，也就是运行的时候就没这个信息了
- RetentionPolicy.RUNTIME: 会写入 class 文件，永久保存，可以通过反射获取注解信息

所以我上文写的是解析的时候，没写具体是解析啥，因为不同的生命周期的解析动作是不同的。

像常见的：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override { }
```

就是给编译器用的，编译器编译的时候检查没问题就over了，class文件里面不会有Override这个标记。

再比如 Spring 常见的 Autowired，就是 RUNTIME 的，所以在运行的时候可以通过反射得到注解的信息，还能拿到标记的值 required。

```
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD, ElementType.PARAMETER, ElementType.FIELD, ElementType.ANNOTATION_TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface Autowired {  
    boolean required() default true;  
}
```

反射

49.什么是反射？应用？原理？

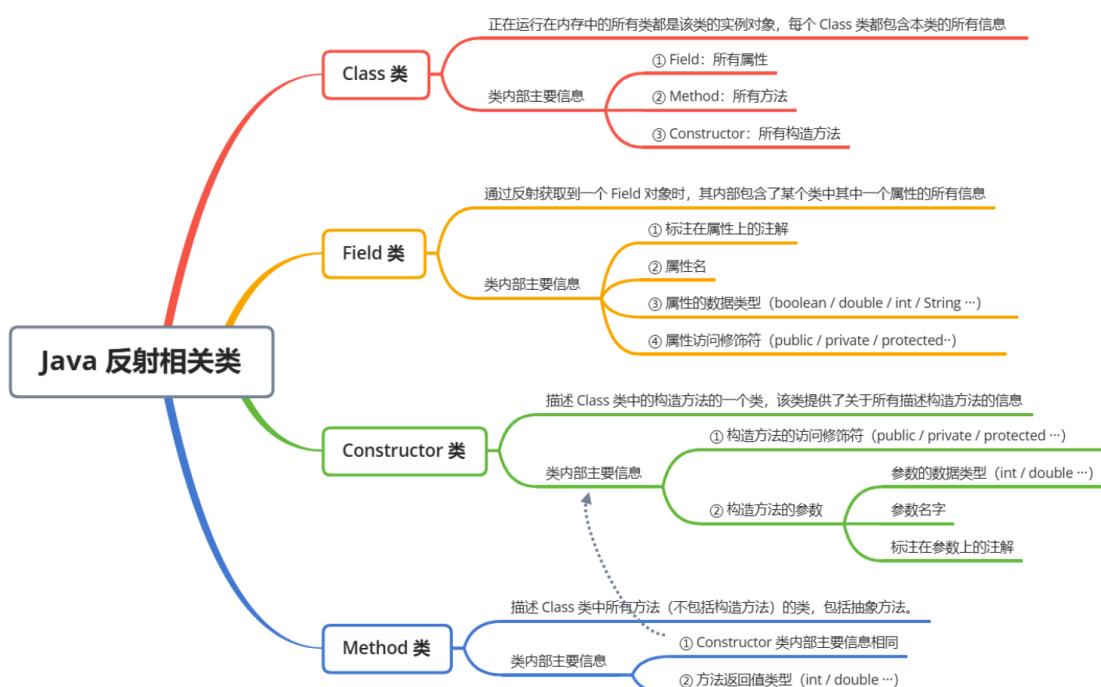
什么是反射？

我们通常都是利用 new 方式来创建对象实例，这可以说就是一种“正射”，这种方式在编译时候就确定了类型信息。

而如果，我们想在时候动态地获取类信息、创建类实例、调用类方法这时候就要用到反射。

通过反射你可以获取任意一个类的所有属性和方法，你还可以调用这些方法和属性。

反射最核心的四个类：



反射的应用场景？

一般我们平时都是在写业务代码，很少会接触到直接使用反射机制的场景。

但是，这并不代表反射没有用。相反，正是因为反射，你才能这么轻松地使用各种框架。像 Spring/Spring Boot、MyBatis 等等框架中都大量使用了反射机制。

像 Spring 里的很多 **注解**，它真正的功能实现就是利用反射。

就像为什么我们使用 Spring 的时候，一个 **@Component** 注解就声明了一个类为 Spring Bean 呢？为什么通过一个 **@Value** 注解就读取到配置文件中的值呢？究竟是怎么起作用的呢？

这些都是因为我们可以基于反射操作类，然后获取到类/属性/方法/方法的参数上的注解，注解这里就有两个作用，一是标记，我们对注解标记的类/属性/方法进行对应的处理；二是注解本身有一些信息，可以参与到处理的逻辑中。

反射的原理？

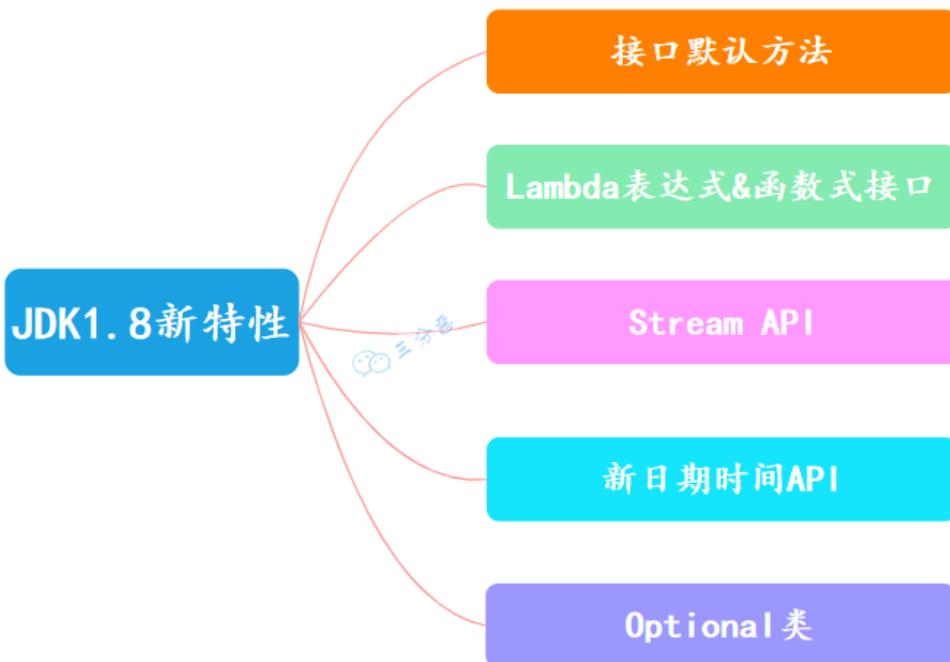
我们都知道 Java 程序的执行分为编译和运行两步，编译之后会生成字节码(.class)文件，JVM 进行类加载的时候，会加载字节码文件，将类型相关的所有信息加载进方法区，反射就是去获取这些信息，然后进行各种操作。

JDK1.8新特性

JDK 已经出到 17 了，但是你迭代你的版本，我用我的 8。JDK 1.8 的一些新特性，当然现在也不新了，其实在工作中已经很常用了。

50.JDK1.8都有哪些新特性？

JDK 1.8 有不少新特性，我们经常接触到的新特性如下：



- 接口默认方法：Java 8允许我们给接口添加一个非抽象的方法实现，只需要使用 `default`关键字修饰即可
- Lambda 表达式和函数式接口：Lambda 表达式本质上是一段匿名内部类，也可以是一段可以传递的代码。Lambda 允许把函数作为一个方法的参数（函数作为参数传递到方法中），使用 Lambda 表达式使代码更加简洁，但是也不要滥用，否则会有可读性等问题，《Effective Java》作者 Josh Bloch 建议使用 Lambda 表达式最好不要超过3行。
- Stream API：用函数式编程方式在集合类上进行复杂操作的工具，配合Lambda 表达式可以方便的对集合进行处理。

Java8 中处理集合的关键抽象概念，它可以指定你希望对集合进行的操作，可以执行非常复杂的查找、过滤和映射数据等操作。使用 Stream API 对集合数据进行操作，就类似于使用 SQL 执行的数据库查询。也可以使用 Stream API 来并行执行操作。

简而言之，Stream API 提供了一种高效且易于使用的处理数据的方式。

- 日期时间API：Java 8 引入了新的日期时间API改进了日期时间的管理。
- Optional 类：用来解决空指针异常的问题。很久以前 Google Guava 项目引入了 Optional 作为解决空指针异常的一种方式，不赞成代码被 null 检查的代码污染，期望程序员写整洁的代码。受Google Guava的鼓励，Optional 现在是Java 8库的一部分。

51.Lambda 表达式了解多少？

Lambda 表达式本质上是一段匿名内部类，也可以是一段可以传递的代码。

比如我们以前使用Runnable创建并运行线程：

```
1 |     new Thread(new Runnable() {
2 |         @Override
3 |         public void run() {
4 |             System.out.println("Thread is running before
5 | Java8!");
6 |         }
7 |     }).start();
```

这是通过内部类的方式来重写run方法，使用Lambda表达式，还可以更加简洁：

```
1 | new Thread( () -> System.out.println("Thread is running since
2 | Java8!") ).start();
```

当然不是每个接口都可以缩写成 Lambda 表达式。只有那些函数式接口（Functional Interface）才能缩写成 Lambda 表达式。

所谓函数式接口（Functional Interface）就是只包含一个抽象方法的声明。针对该接口类型的所有 Lambda 表达式都会与这个抽象方法匹配。

Java8有哪些内置函数式接口？

JDK 1.8 API 包含了很多内置的函数式接口。其中就包括我们在老版本中经常见到的 **Comparator** 和 **Runnable**，Java 8 为他们都添加了 `@FunctionalInterface` 注解，以用来支持 Lambda 表达式。

除了这两个之外，还有**Callable**、**Predicate**、**Function**、**Supplier**、**Consumer**等等。

52.Optional了解吗？

Optional 是用于防范 **NullPointerException**。

可以将 `Optional` 看做是包装对象（可能是 `null`, 也有可能非 `null`）的容器。当我们定义了一个方法，这个方法返回的对象可能是空，也有可能非空的时候，我们就可以考虑用 `Optional` 来包装它，这也是在 Java 8 被推荐使用的方法。

```
1  Optional<String> optional = Optional.of("bam");
2
3  optional.isPresent();           // true
4  optional.get();                // "bam"
5  optional.orElse("fallback");   // "bam"
6
7  optional.ifPresent(s -> System.out.println(s.charAt(0)));
   // "b"
```

53.Stream 流用过吗？

`Stream` 流，简单来说，使用 `java.util.Stream` 对一个包含一个或多个元素的集合做各种操作。这些操作可能是 中间操作 亦或是 终端操作。终端操作会返回一个结果，而中间操作会返回一个 `Stream` 流。

Stream流一般用于集合，我们对一个集合做几个常见操作：

```
1  List<String> stringCollection = new ArrayList<>();
2  stringCollection.add("ddd2");
3  stringCollection.add("aaa2");
4  stringCollection.add("bbb1");
5  stringCollection.add("aaa1");
6  stringCollection.add("bbb3");
7  stringCollection.add("ccc");
8  stringCollection.add("bbb2");
9  stringCollection.add("ddd1");
```

- Filter 过滤

```
1 | stringCollection
2 |     .stream()
3 |     .filter((s) -> s.startsWith("a"))
4 |     .forEach(System.out::println);
5 |
6 | // "aaa2", "aaa1"
7 |
```

- Sorted 排序

```
1 | stringCollection
2 |     .stream()
3 |     .sorted()
4 |     .filter((s) -> s.startsWith("a"))
5 |     .forEach(System.out::println);
6 |
7 | // "aaa1", "aaa2"
```

- Map 转换

```
1 | stringCollection
2 |     .stream()
3 |     .map(String::toUpperCase)
4 |     .sorted((a, b) -> b.compareTo(a))
5 |     .forEach(System.out::println);
6 |
7 | // "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"
```

- Match 匹配

```
1 | // 验证 list 中 string 是否有以 a 开头的，匹配到第一个，即返回 true
2 | boolean anyStartsWithA =
3 |     stringCollection
4 |         .stream()
5 |         .anyMatch((s) -> s.startsWith("a"));
6 |
7 | System.out.println(anyStartsWithA);          // true
8 |
9 | // 验证 list 中 string 是否都是以 a 开头的
```

```
10 boolean allStartsWithA =
11     stringCollection
12         .stream()
13         .allMatch((s) -> s.startsWith("a"));
14
15 System.out.println(allStartsWithA);          // false
16
17 // 验证 list 中 string 是否都不是以 z 开头的,
18 boolean noneStartsWithZ =
19     stringCollection
20         .stream()
21         .noneMatch((s) -> s.startsWith("z"));
22
23 System.out.println(noneStartsWithZ);          // true
```

- Count 计数

`count` 是一个终端操作，它能够统计 `stream` 流中的元素总数，返回值是 `long` 类型。

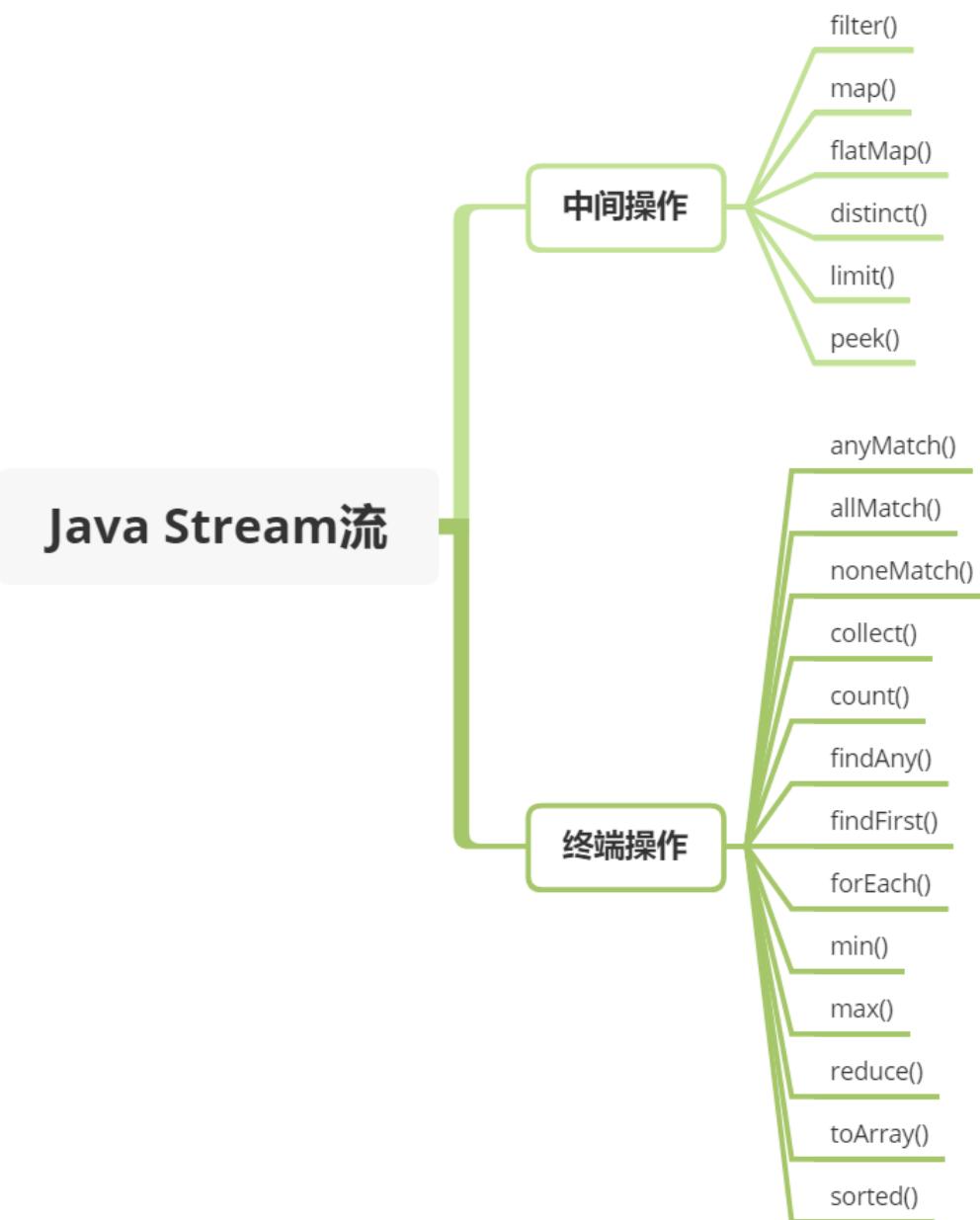
```
1 // 先对 list 中字符串开头为 b 进行过滤, 让后统计数量
2 long startsWithB =
3     stringCollection
4         .stream()
5         .filter((s) -> s.startsWith("b"))
6         .count();
7
8 System.out.println(startsWithB);      // 3
```

- Reduce

`Reduce` 中文翻译为：减少、缩小。通过入参的 `Function`，我们能够将 `list` 归约成一个值。它的返回类型是 `Optional` 类型。

```
1 Optional<String> reduced =  
2     stringCollection  
3         .stream()  
4         .sorted()  
5         .reduce((s1, s2) -> s1 + "#" + s2);  
6  
7     reduced.ifPresent(System.out::println);  
8 // "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"
```

以上是常见的几种流式操作，还有其它的一些流式操作，可以帮助我们更便捷地处理集合数据。



参考：

- [1]. [Java 基础高频面试题（2021年最新版）](#)
 - [2]. [2.7w字！Java基础面试题/知识点总结！（2021 最新版）](#)
 - [3]. [面试题系列第8篇：谈谈String、StringBuffer、StringBuilder的区别？](#)
 - [4]. [面试题系列第2篇：new String\(\)创建几个对象？有你不知道的](#)
 - [5]. [面试题系列第6篇：JVM字符串常量池及String的intern方法详解？](#)
 - [6]. [2W字，52道Java热点必考题，含答案，图文并茂](#)
 - [7]. [BIO、NIO、AIO、Netty面试题（总结最全面的面试题！！！）](#)
 - [8]. [Java基础知识面试题（2020最新版）](#)
 - [9]. [Java基础面试题（2021最新版）](#)
 - [10]. [干货 | Java8 新特性教程](#)
 - [11]. [面向对象和面向过程分别是什么？](#)
 - [12]. [《疯狂Java讲义》](#)
 - [13]. [3. 彤哥说netty系列之Java BIO NIO AIO进化史](#)
 - [14]. [什么是泛型擦除？](#)
 - [15]. [学会反射后，我被录取了（干货）](#)
-

关注公众号：三分恶

手册更新动态
即刻送达



添加个人微信：ThirdFighter

技术交流
加大佬云集微信群

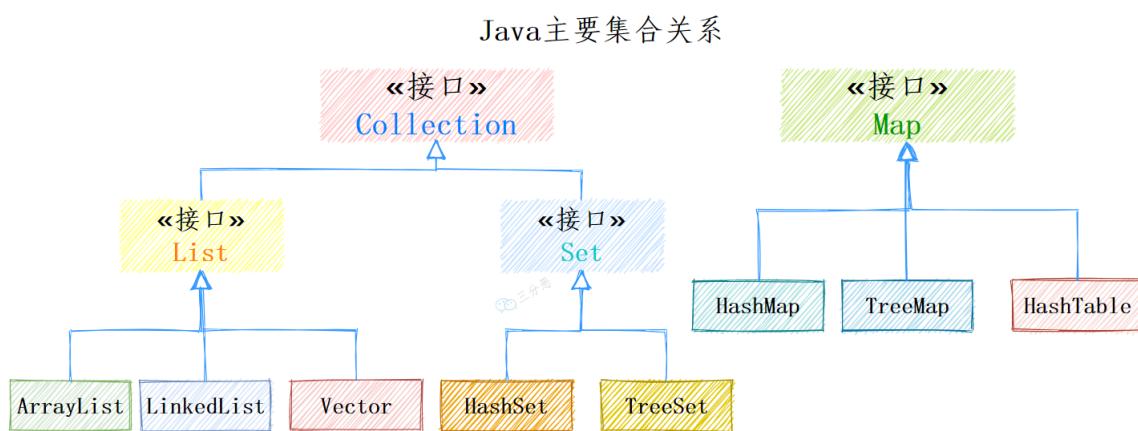


二、Java集合

引言

1. 说说有哪些常见集合？

集合相关类和接口都在java.util中，主要分为3种：List（列表）、Map（映射）、Set(集)。



其中 **Collection** 是集合 **List**、**Set** 的父接口，它主要有两个子接口：

- **List**：存储的元素有序，可重复。
- **Set**：存储的元素无序，不可重复。

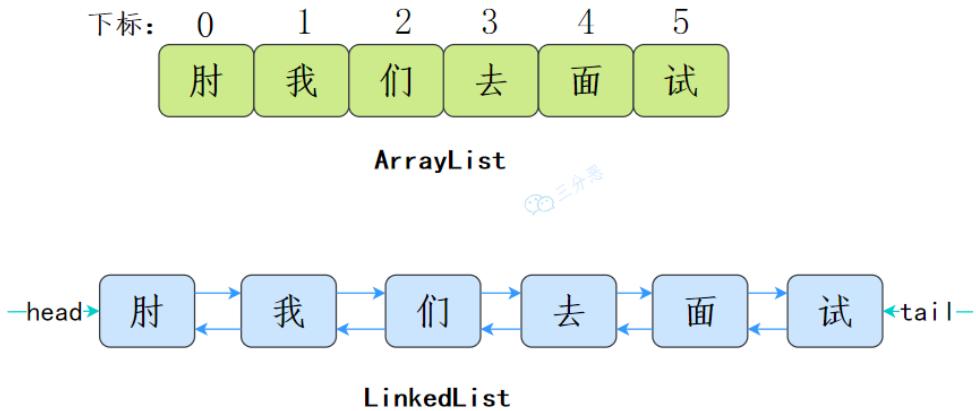
Map 是另外的接口，是键值对映射结构的集合。

List

2. **ArrayList**和**LinkedList**有什么区别？

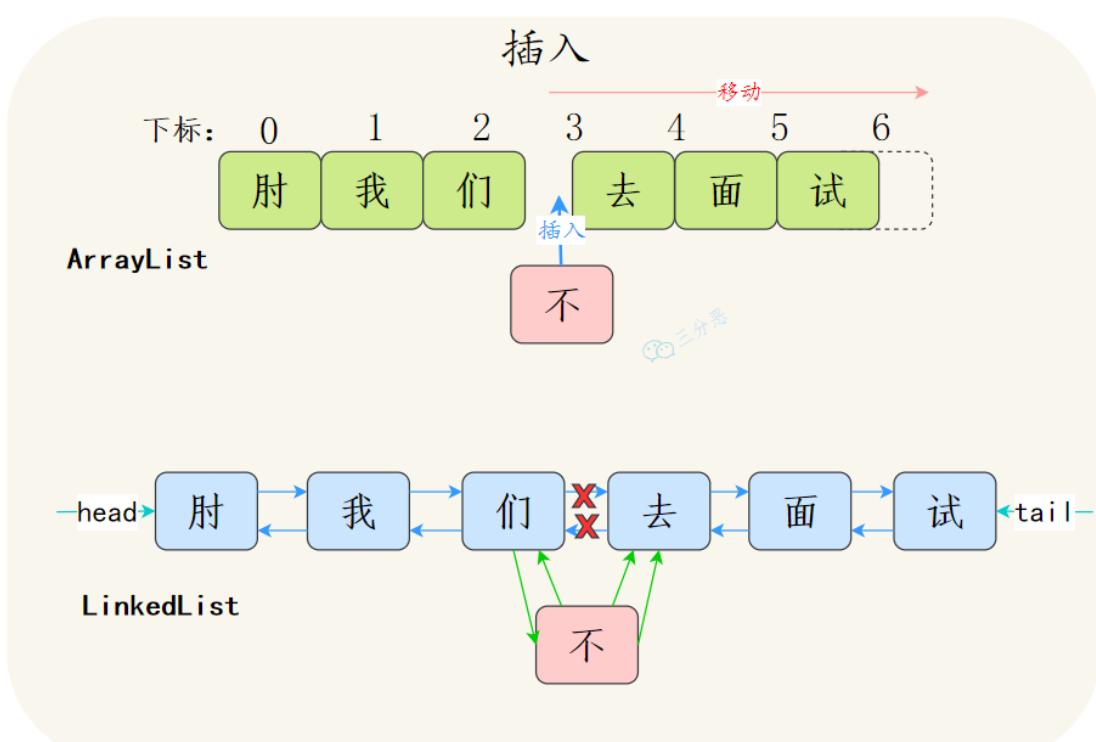
(1) 数据结构不同

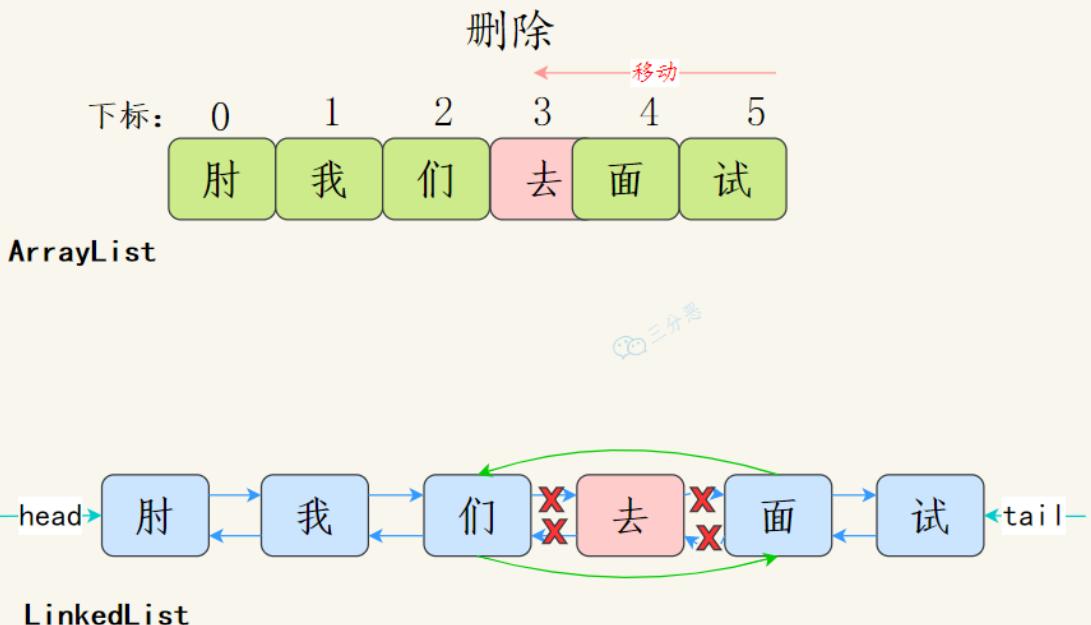
- **ArrayList**基于数组实现
- **LinkedList**基于双向链表实现



(2) 多数情况下，ArrayList更利于查找，LinkedList更利于增删

- ArrayList基于数组实现，`get(int index)`可以直接通过数组下标获取，时间复杂度是O(1)； LinkedList基于链表实现，`get(int index)`需要遍历链表，时间复杂度是O(n)；当然，`get(E element)`这种查找，两种集合都需要遍历，时间复杂度都是O(n)。
- ArrayList增删如果是数组末尾的位置，直接插入或者删除就可以了，但是如果插入中间的位置，就需要把插入位置后的元素都向前或者向后移动，甚至还有可能触发扩容；双向链表的插入和删除只需要改变前驱节点、后继节点和插入节点的指向就行了，不需要移动元素。





注意，这个地方可能会出陷阱，**LinkedList**更利于增删更多是体现在平均步长上，不是体现在时间复杂度上，二者增删的时间复杂度都是O(n)

(3) 是否支持随机访问

- **ArrayList** 基于数组，所以它可以根据下标查找，支持随机访问，当然，它也实现了 **RandomAccess** 接口，这个接口只是用来标识是否支持随机访问。
- **LinkedList** 基于链表，所以它没法根据序号直接获取元素，它没有实现 **RandomAccess** 接口，标记不支持随机访问。

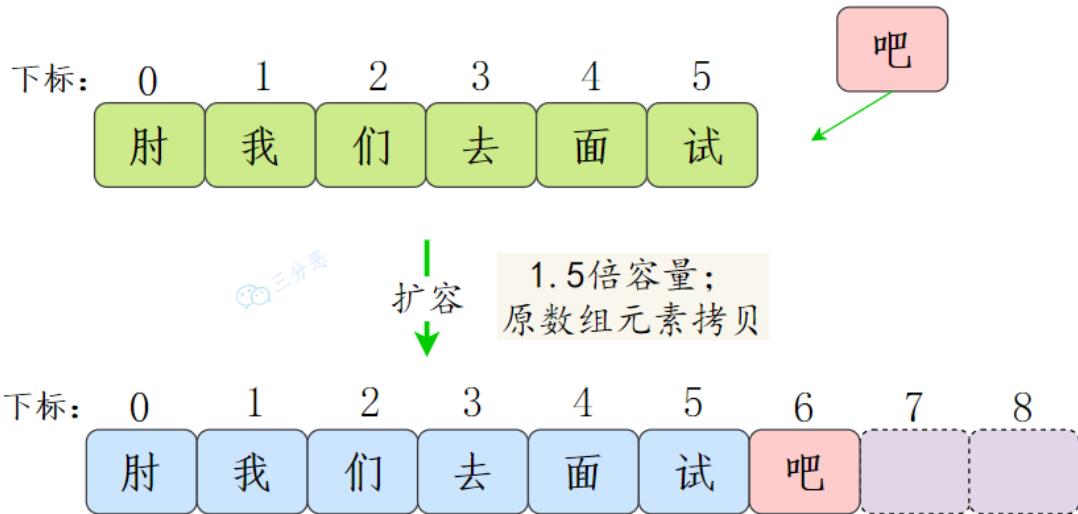
(4) 内存占用，**ArrayList** 基于数组，是一块连续的内存空间，**LinkedList** 基于链表，内存空间不连续，它们在空间占用上都有一些额外的消耗：

- **ArrayList** 是预先定义好的数组，可能会有空的内存空间，存在一定空间浪费
- **LinkedList** 每个节点，需要存储前驱和后继，所以每个节点会占用更多的空间

3. **ArrayList** 的扩容机制了解吗？

ArrayList 是基于数组的集合，数组的容量是在定义的时候确定的，如果数组满了，再插入，就会数组溢出。所以在插入时候，会先检查是否需要扩容，如果当前容量 +1 超过数组长度，就会进行扩容。

ArrayList 的扩容是创建一个**1.5倍**的新数组，然后把原数组的值拷贝过去。



4.ArrayList怎么序列化的知道吗？为什么用transient修饰数组？

ArrayList的序列化不太一样，它使用 `transient` 修饰存储元素的 `elementData` 的数组，`transient` 关键字的作用是让被修饰的成员属性不被序列化。

为什么最ArrayList不直接序列化元素数组呢？

出于效率的考虑，数组可能长度100，但实际只用了50，剩下的50不用其实不用序列化，这样可以提高序列化和反序列化的效率，还可以节省内存空间。

那ArrayList怎么序列化呢？

ArrayList通过两个方法`readObject`、`writeObject`自定义序列化和反序列化策略，实际直接使用两个流 `ObjectOutputStream` 和 `ObjectInputStream` 来进行序列化和反序列化。

```
/*  
 * 自定义序列化  
 */  
private void writeObject(java.io.ObjectOutputStream s)  
    throws java.io.IOException{  
    // fail-fast, 后续判断是否有并发处理  
    int expectedModCount = modCount;  
    // 序列化没有标记为 static、transient 的字段, 包括 size 等。  
    s.defaultWriteObject();  
  
    s.writeInt(size);  
  
    // 序列化数组的前size个元素  
    for (int i=0; i<size; i++) {  
        s.writeObject(elementData[i]);  
    }  
  
    if (modCount != expectedModCount) {  
        throw new ConcurrentModificationException();  
    }  
}  
  
/*  
 * 自定义反序列化  
 */  
private void readObject(java.io.ObjectInputStream s)  
    throws java.io.IOException, ClassNotFoundException {  
    elementData = EMPTY_ELEMENTDATA;  
  
    //反序列化没有标记为 static、transient 的字段, 包括 size 等  
    s.defaultReadObject();  
  
    s.readInt(); // ignored  
  
    if (size > 0) {  
        // 数组扩容  
        int capacity = calculateCapacity(elementData, size);  
        SharedSecrets.getJavaOISAccess().checkArray(s, Object[].class, capacity);  
        ensureCapacityInternal(size);  
  
        Object[] a = elementData;  
        // 反序列化元素并填充到数组中  
        for (int i=0; i<size; i++) {  
            a[i] = s.readObject();  
        }  
    }  
}
```

5.快速失败(fail-fast)和安全失败(fail-safe)了解吗？

快速失败 (fail-fast) : 快速失败是Java集合的一种错误检测机制

- 在用迭代器遍历一个集合对象时，如果线程A遍历过程中，线程B对集合对象的内容进行了修改（增加、删除、修改），则会抛出Concurrent Modification Exception。

- 原理：迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 `modCount` 变量。集合在被遍历期间如果内容发生变化，就会改变 `modCount` 的值。每当迭代器使用 `hashNext()`/`next()` 遍历下一个元素之前，都会检测 `modCount` 变量是否为 `expectedModCount` 值，是的话就返回遍历；否则抛出异常，终止遍历。
- 注意：这里异常的抛出条件是检测到 `modCount != expectedModCount` 这个条件。如果集合发生变化时修改 `modCount` 值刚好又设置为了 `expectedModCount` 值，则异常不会抛出。因此，不能依赖于这个异常是否抛出而进行并发操作的编程，这个异常只建议用于检测并发修改的 bug。
- 场景：`java.util` 包下的集合类都是快速失败的，不能在多线程下发生并发修改（迭代过程中被修改），比如 `ArrayList` 类。

安全失败（fail-safe）

- 采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。
- 原理：由于迭代时是对原集合的拷贝进行遍历，所以在遍历过程中对原集合所作的修改并不能被迭代器检测到，所以不会触发 `Concurrent Modification Exception`。
- 缺点：基于拷贝内容的优点是避免了 `Concurrent Modification Exception`，但同样地，迭代器并不能访问到修改后的内容，即：迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的。
- 场景：`java.util.concurrent` 包下的容器都是安全失败，可以在多线程下并发使用，并发修改，比如 `CopyOnWriteArrayList` 类。

6. 有哪几种实现 `ArrayList` 线程安全的方法？

`fail-fast` 是一种可能触发的机制，实际上，`ArrayList` 的线程安全仍然没有保证，一般，保证 `ArrayList` 的线程安全可以通过这些方案：

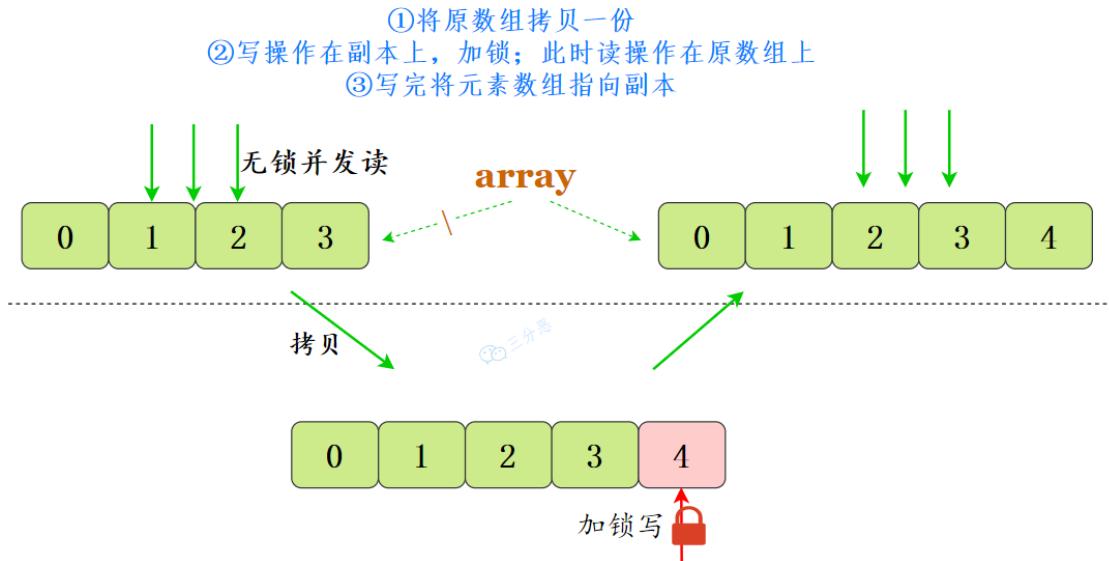
- 使用 `Vector` 代替 `ArrayList`。（不推荐，`Vector` 是一个历史遗留类）
- 使用 `Collections.synchronizedList` 包装 `ArrayList`，然后操作包装后的 `list`。
- 使用 `CopyOnWriteArrayList` 代替 `ArrayList`。
- 在使用 `ArrayList` 时，应用程序通过同步机制去控制 `ArrayList` 的读写。

7. `CopyOnWriteArrayList` 了解多少？

`CopyOnWriteArrayList` 就是线程安全版本的 `ArrayList`。

它的名字叫 **CopyOnWrite**——写时复制，已经明示了它的原理。

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



Map

Map中，毫无疑问，最重要的就是HashMap，面试基本被盘出包浆了，各种问法，一定要好好准备。

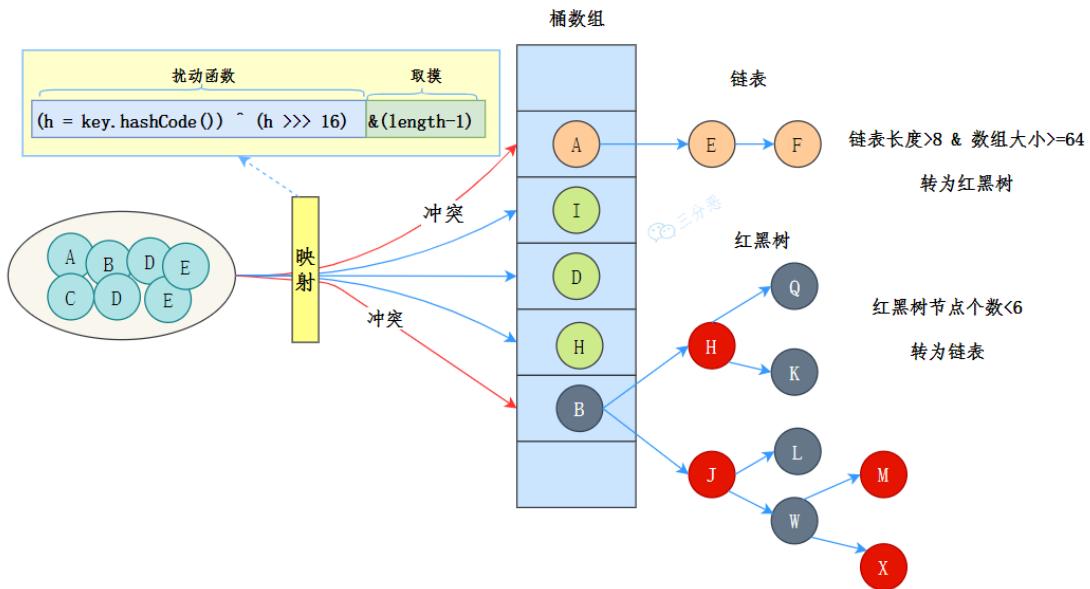
8.能说一下HashMap的数据结构吗？

JDK1.7的数据结构是 **数组 + 链表**，JDK1.7还有人在用？不会吧.....

说一下JDK1.8的数据结构吧：

JDK1.8的数据结构是 **数组 + 链表 + 红黑树**。

数据结构示意图如下：



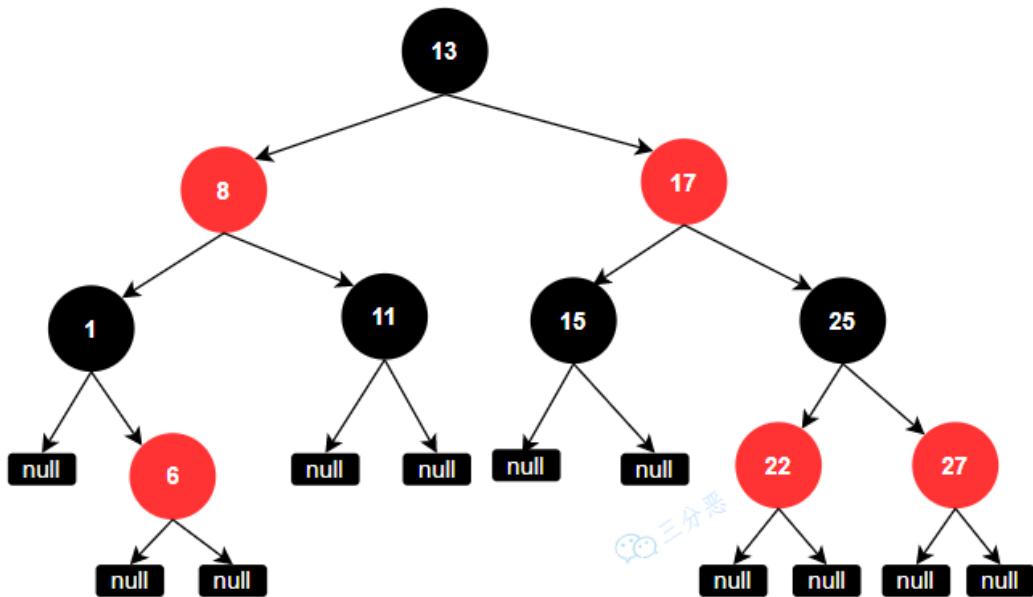
其中，桶数组是用来存储数据元素，链表是用来解决冲突，红黑树是为了提高查询的效率。

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

9.你对红黑树了解多少？为什么不用二叉树/平衡树呢？

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

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



之所以不用二叉树：

红黑树是一种平衡的二叉树，插入、删除、查找的最坏时间复杂度都为 $O(\log n)$ ，避免了二叉树最坏情况下的 $O(n)$ 时间复杂度。

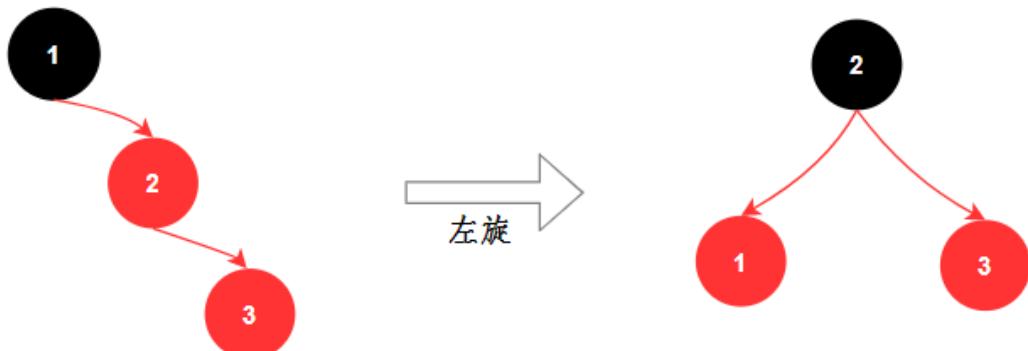
之所以不用平衡二叉树：

平衡二叉树是比红黑树更严格的平衡树，为了保持平衡，需要旋转的次数更多，也就是说平衡二叉树保持平衡的效率更低，所以平衡二叉树插入和删除的效率比红黑树要低。

10. 红黑树怎么保持平衡的知道吗？

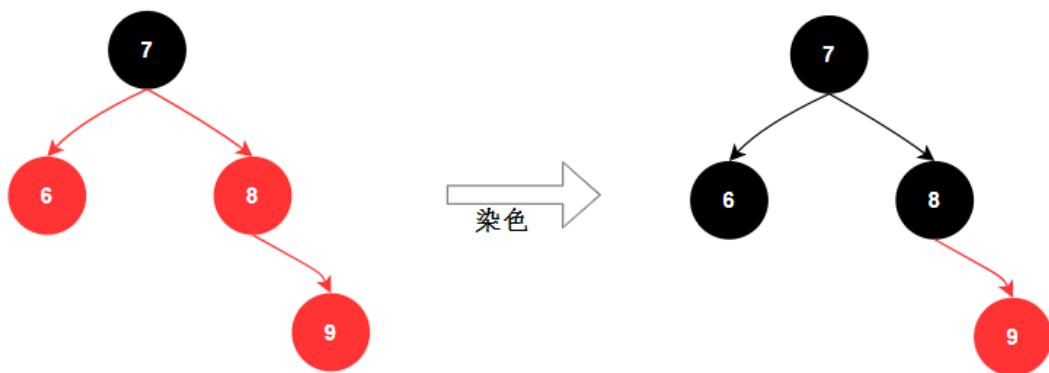
红黑树有两种方式保持平衡： **旋转** 和 **染色**。

- 旋转：旋转分为两种，左旋和右旋



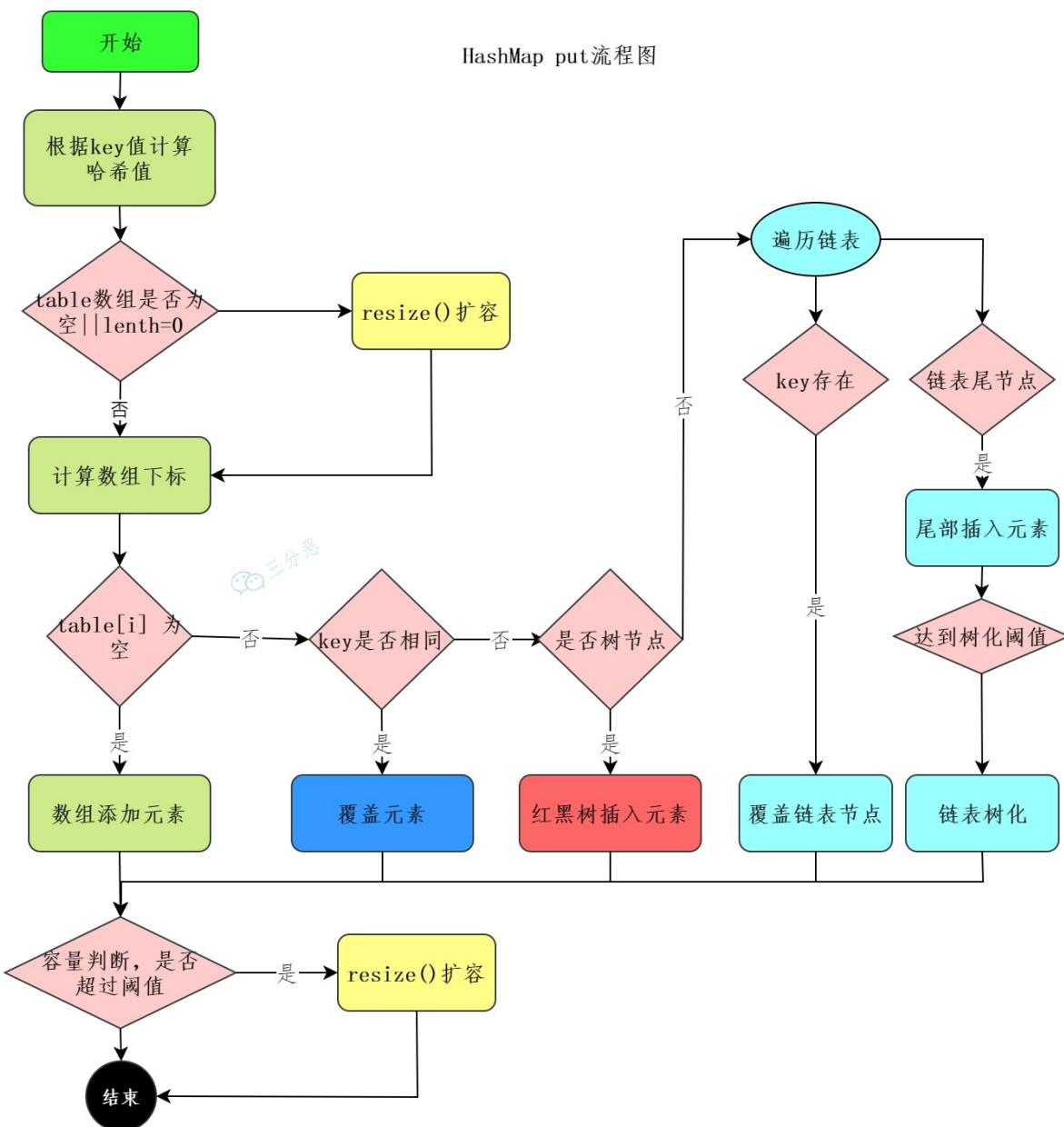


- 染色:



11.HashMap的put流程知道吗？

先上个流程图吧：



- 首先进行哈希值的扰动，获取一个新的哈希值。`(key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);`
- 判断tab是否位空或者长度为0，如果是则进行扩容操作。

```

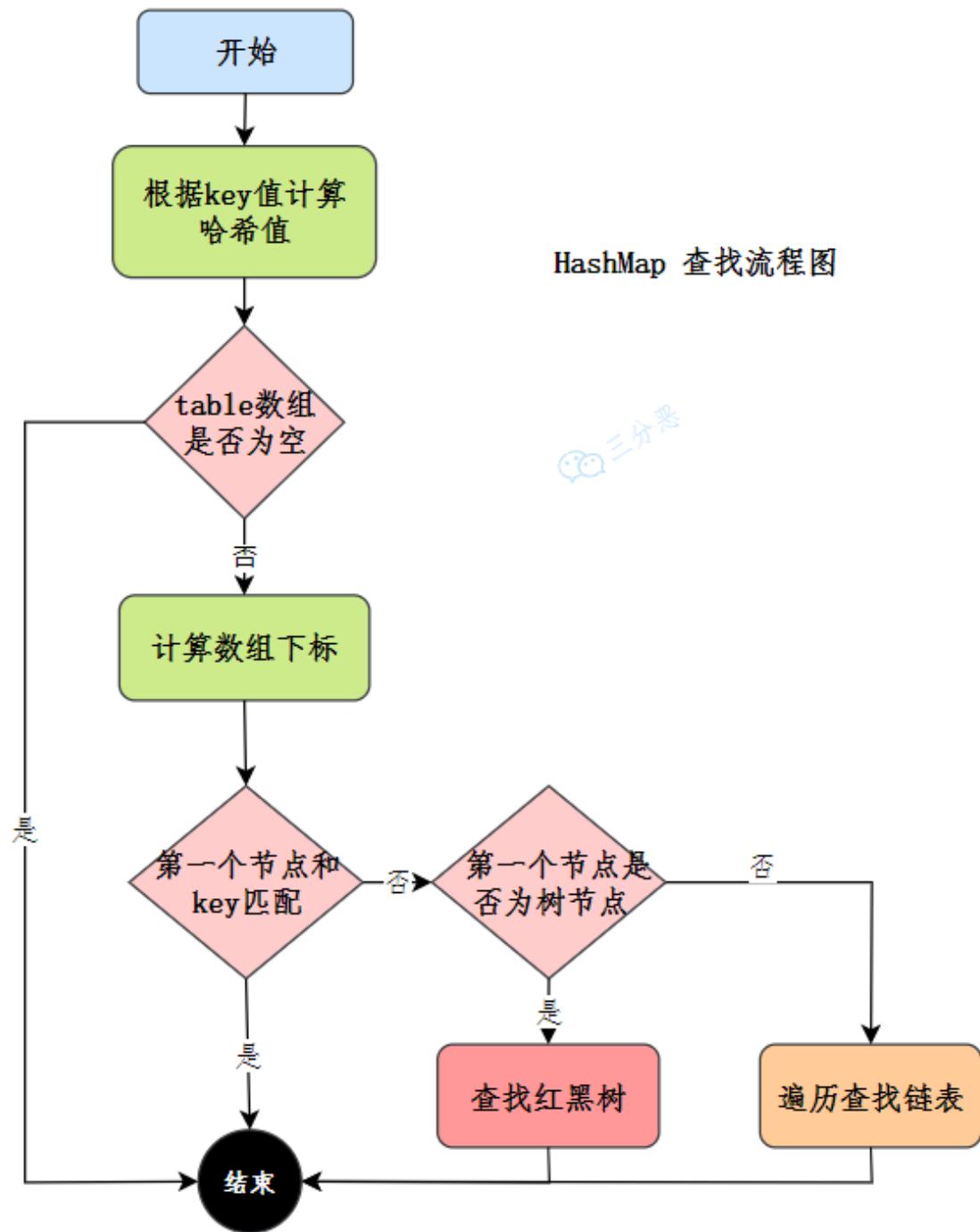
1 | if ((tab = table) == null || (n = tab.length) == 0)
2 |     n = (tab = resize()).length;

```

- 根据哈希值计算下标，如果对应小标正好没有存放数据，则直接插入即可否则需要覆盖。`tab[i = (n - 1) & hash]`
- 判断tab[i]是否为树节点，否则向链表中插入数据，是则向树中插入节点。
- 如果链表中插入节点的时候，链表长度大于等于8，则需要把链表转换为红黑树。`treeifyBin(tab, hash);`
- 最后所有元素处理完成后，判断是否超过阈值；`threshold`，超过则扩容。

12.HashMap怎么查找元素的呢？

先看流程图：



HashMap的查找就简单很多：

1. 使用扰动函数，获取新的哈希值
2. 计算数组下标，获取节点
3. 当前节点和key匹配，直接返回
4. 否则，当前节点是否为树节点，查找红黑树

5. 否则，遍历链表查找

13.HashMap的哈希/扰动函数是怎么设计的？

HashMap的哈希函数是先拿到 key 的hashcode，是一个32位的int类型的数值，然后让hashcode的高16位和低16位进行异或操作。

```
1 |     static final int hash(Object key) {
2 |         int h;
3 |         // key的hashCode和key的hashCode右移16位做异或运算
4 |         return (key == null) ? 0 : (h = key.hashCode()) ^ (h
5 |             >>> 16);
}
```

这么设计是为了降低哈希碰撞的概率。

14.为什么哈希/扰动函数能降hash碰撞？

因为 key.hashCode() 函数调用的是 key 键值类型自带的哈希函数，返回 int 型散列值。int 值范围为 **-2147483648~2147483647**，加起来大概 40 亿的映射空间。

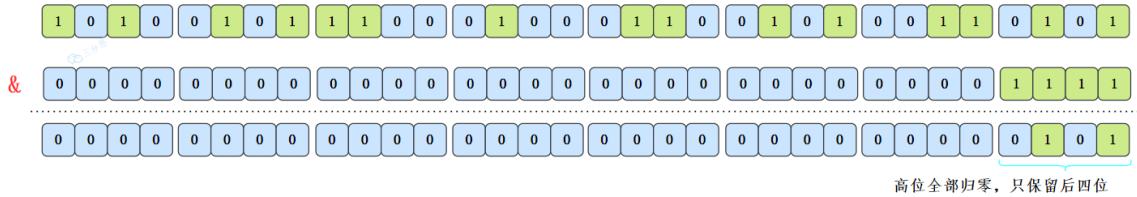
只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个 40 亿长度的数组，内存是放不下的。

假如 HashMap 数组的初始大小才 16，就需要用之前需要对数组的长度取模运算，得到的余数才能用来访问数组下标。

源码中模运算就是把散列值和数组长度 - 1 做一个 "**与&**" 操作，位运算比取余 % 运算要快。

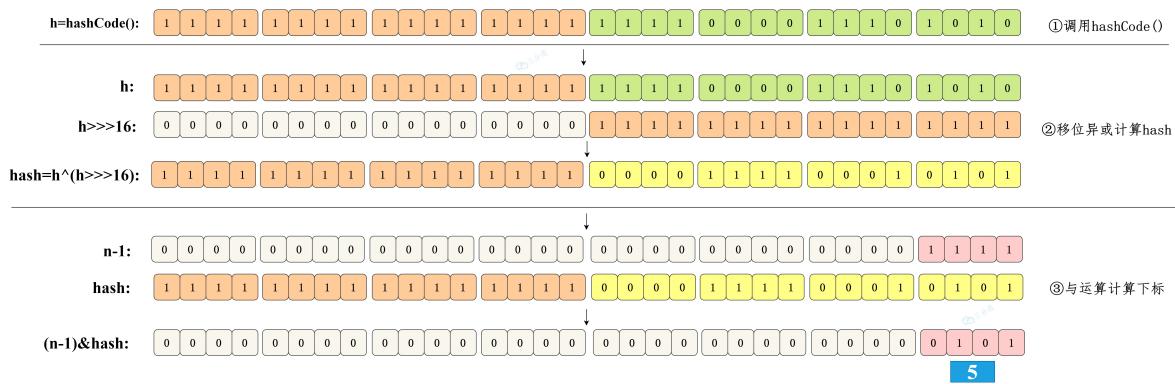
```
1 | bucketIndex = indexFor(hash, table.length);
2 |
3 | static int indexFor(int h, int length) {
4 |     return h & (length-1);
5 | }
```

顺便说一下，这也正好解释了为什么 HashMap 的数组长度要取 2 的整数幂。因为这样（数组长度 - 1）正好相当于一个“低位掩码”。**与** 操作的结果就是散列值的高位全部归零，只保留低位值，用来做数组下标访问。以初始长度 16 为例， $16-1=15$ 。2 进制表示是 **0000 0000 0000 0000 0000 0000 0000 1111**。和某个散列值做 **与** 操作如下，结果就是截取了最低的四位值。



这样是要快捷一些，但是新的问题来了，就算散列值分布再松散，要是只取最后几位的话，碰撞也会很严重。如果散列本身做得不好，分布上成等差数列的漏洞，如果正好让最后几个低位呈现规律性重复，那就更难搞了。

这时候 **扰动函数** 的价值就体现出来了，看一下扰动函数的示意图：



右移 16 位，正好是 32bit 的一半，自己的高半区和低半区做异或，就是为了混合原始哈希码的高位和低位，以此来加大低位的随机性。而且混合后的低位掺杂了高位的部分特征，这样高位的信息也被变相保留下。

15.为什么HashMap的容量是2的倍数呢？

- 第一个原因是为了方便哈希取余：

将元素放在table数组上面，是用hash值%数组大小定位位置，而HashMap是用hash值 &(数组大小-1)，却能和前面达到一样的效果，这就得益于HashMap的大小是2的倍数，2的倍数意味着该数的二进制位只有一位为1，而该数-1就可以得到二进制位上1变成0，后面的0变成1，再通过&运算，就可以得到和%一样的效果，并且位运算比%的效率高得多

HashMap的容量是2的n次幂时，(n-1)的2进制也就是1111111***111这样形式的，这样与添加元素的hash值进行位运算时，能够充分的散列，使得添加的元素均匀分布在HashMap的每个位置上，减少hash碰撞。

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

我们可以简单看看HashMap的扩容机制，HashMap中的元素在超过 **负载因子***HashMap 大小时就会产生扩容。

```
    ++modCount;
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
```

put时，当大小超过threshold，就会扩容

16.如果初始化HashMap，传一个17的值 new HashMap<>，它会怎么处理？

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

我们来看看详情，在HashMap的初始化中，有这样一段方法；

```
1 public HashMap(int initialCapacity, float loadFactor) {
2     ...
3     this.loadFactor = loadFactor;
4     this.threshold = tableSizeFor(initialCapacity);
5 }
```

- 阀值 threshold，通过方法 **tableSizeFor** 进行计算，是根据初始化传的参数来计算的。
- 同时，这个方法也要要寻找比初始值大的，最小的那个2进制数值。比如传了17，我应该找到的是32。

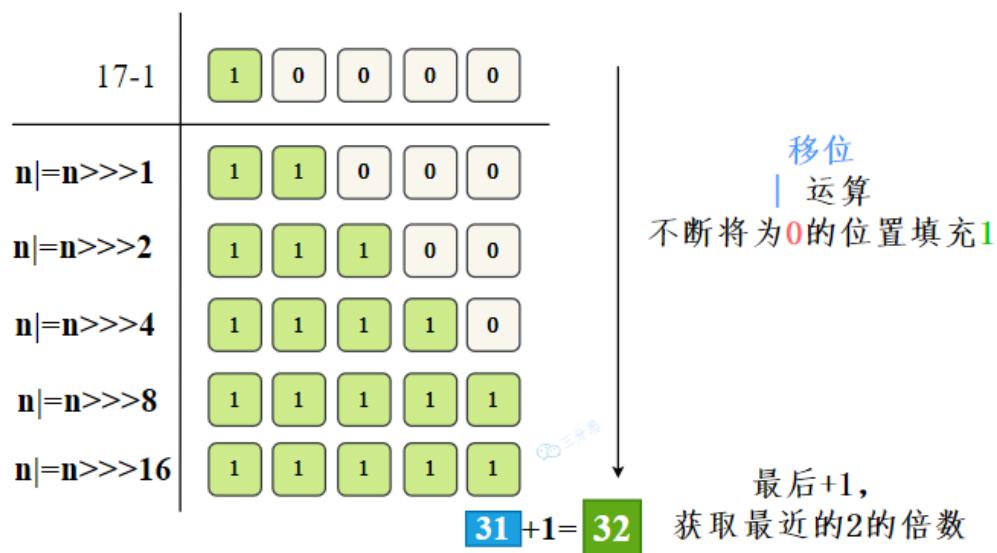
```

1 static final int tableSizeFor(int cap) {
2     int n = cap - 1;
3     n |= n >>> 1;
4     n |= n >>> 2;
5     n |= n >>> 4;
6     n |= n >>> 8;
7     n |= n >>> 16;
8     return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ?
MAXIMUM_CAPACITY : n + 1; }

```

- `MAXIMUM_CAPACITY = 1 << 30`, 这个是临界范围, 也就是最大的Map集合。
- 计算过程是向右移位1、2、4、8、16, 和原来的数做`|`运算, 这主要是为了把二进制的各个位置都填上1, 当二进制的各个位置都是1以后, 就是一个标准的2的倍数减1了, 最后把结果加1再返回即可。

以17为例, 看一下初始化计算table容量的过程:



17.你还知道哪些哈希函数的构造方法呢?

HashMap里哈希构造函数的方法叫:

- 除留取余法 : $H(key)=key \% p$ ($p \leq N$), 关键字除以一个不大于哈希表长度的正整数p, 所得余数为地址, 当然HashMap里进行了优化改造, 效率更高, 散列也更均衡。

除此之外, 还有这几种常见的哈希函数构造方法:

- 直接定址法

直接根据 **key** 来映射到对应的数组位置，例如1232放到下标1232的位置。

- **数字分析法**

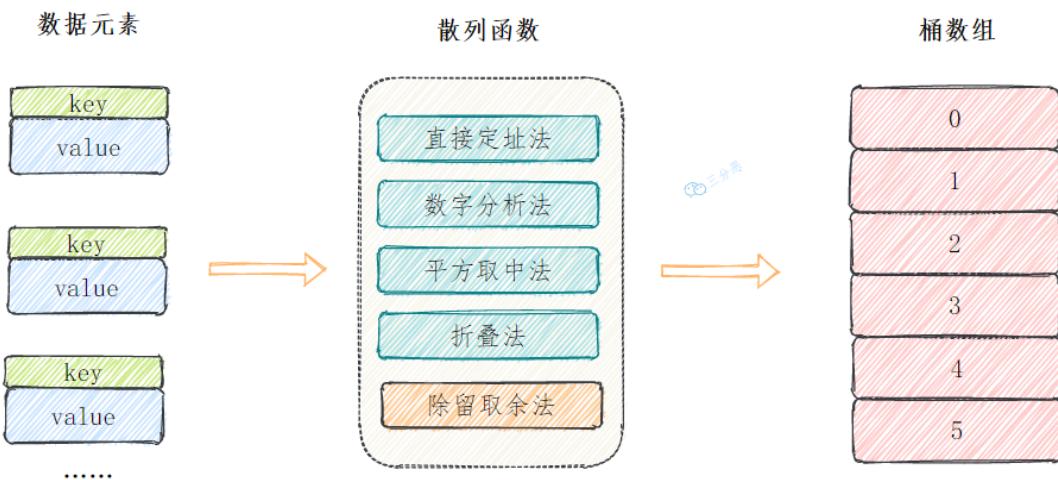
取 **key** 的某些数字（例如十位和百位）作为映射的位置

- **平方取中法**

取 **key** 平方的中间几位作为映射的位置

- **折叠法**

将 **key** 分割成位数相同的几段，然后把它们的叠加和作为映射的位置



18.解决哈希冲突有哪些方法呢？

我们到现在已经知道，HashMap使用链表的原因为了处理哈希冲突，这种方法就是所谓的：

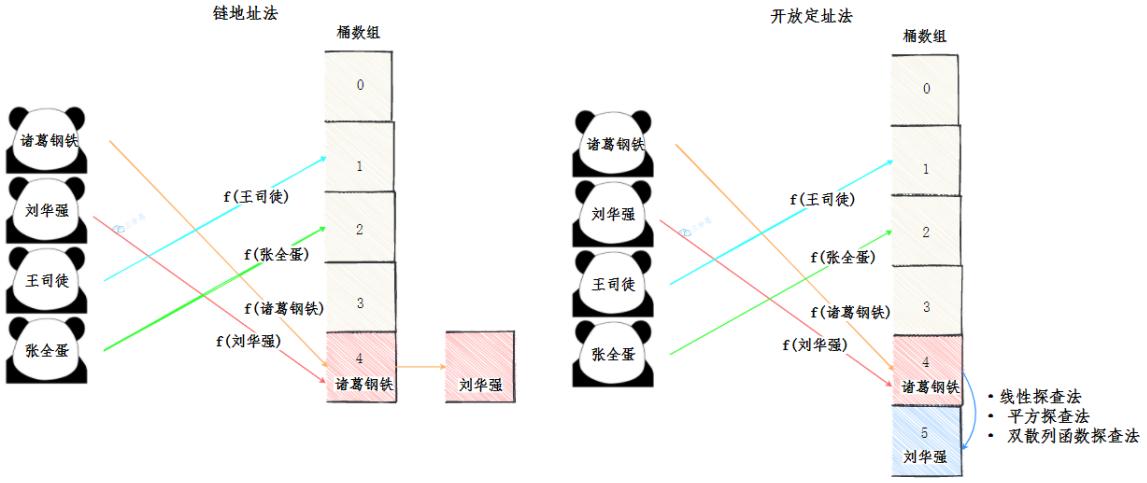
- **链地址法**：在冲突的位置拉一个链表，把冲突的元素放进去。

除此之外，还有一些常见的解决冲突的办法：

- **开放定址法**：开放定址法就是从冲突的位置再接着往下找，给冲突元素找个空位。

找到空闲位置的方法也有很多种：

- 线行探查法：从冲突的位置开始，依次判断下一个位置是否空闲，直至找到空闲位置
- 平方探查法：从冲突的位置x开始，第一次增加 1^2 个位置，第二次增加 2^2 ...，直至找到空闲的位置
-



- 再哈希法：换种哈希函数，重新计算冲突元素的地址。
- 建立公共溢出区：再建一个数组，把冲突的元素放进去。

19.为什么HashMap链表转红黑树的阈值为8呢？

树化发生在table数组的长度大于64，且链表的长度大于8的时候。

为什么是8呢？源码的注释也给出了答案。

```
*  
* Because TreeNodes are about twice the size of regular nodes, we  
* use them only when bins contain enough nodes to warrant use  
* (see TREEIFY_THRESHOLD). And when they become too small (due to  
* removal or resizing) they are converted back to plain bins. In  
* usages with well-distributed user hashCodes, tree bins are  
* rarely used. Ideally, under random hashCodes, the frequency of  
* nodes in bins follows a Poisson distribution  
* (http://en.wikipedia.org/wiki/Poisson\_distribution) with a  
* parameter of about 0.5 on average for the default resizing  
* threshold of 0.75, although with a large variance because of  
* resizing granularity. Ignoring variance, the expected  
* occurrences of list size k are (exp(-0.5) * pow(0.5, k)) /  
* factorial(k)). The first values are:  
*  
* 0: 0.60653066  
* 1: 0.30326533  
* 2: 0.07581633  
* 3: 0.01263606  
* 4: 0.00157952  
* 5: 0.00015795  
* 6: 0.00001316  
* 7: 0.00000094  
* 8: 0.00000006  
* more: less than 1 in ten million
```

红黑树节点的大小大概是普通节点大小的两倍，所以转红黑树，牺牲了空间换时间，更多的是一种兜底的策略，保证极端情况下的查找效率。

阈值为什么要选8呢？和统计学有关。理想情况下，使用随机哈希码，链表里的节点符合泊松分布，出现节点个数的概率是递减的，节点个数为8的情况，发生概率仅为 **0.00000006**。

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

20. 扩容在什么时候呢？为什么扩容因子是0.75？

为了减少哈希冲突发生的概率，当当前HashMap的元素个数达到一个临界值的时候，就会触发扩容，把所有元素rehash之后再放在扩容后的容器中，这是一个相当耗时的操作。

```
    if (++size > threshold)    size达到临界值，就扩容  
        resize();
```

而这个 **临界值threshold** 就是由加载因子和当前容器的容量大小来确定的，假如采用默认的构造方法：

临界值 (threshold) = 默认容量 (DEFAULT_INITIAL_CAPACITY) * 默认扩容因子 (DEFAULT_LOAD_FACTOR)

```
else {                                // zero initial threshold signifies using defaults  
    newCap = DEFAULT_INITIAL_CAPACITY;  
    newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);  
}  
if (newThr == 0) {  
    float ft = (float)newCap * loadFactor;  
    newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?  
              (int)ft : Integer.MAX_VALUE);  
}  
threshold = newThr;
```

那就是大于 $16 \times 0.75 = 12$ 时，就会触发扩容操作。

那么为什么选择了0.75作为HashMap的默认加载因子呢？

简单来说，这是对 **空间** 成本和 **时间** 成本平衡的考虑。

在HashMap中有这样一段注释：

```
*  
* <p>As a general rule, the default load factor (.75) offers a good  
* tradeoff between time and space costs. Higher values decrease the  
* space overhead but increase the lookup cost (reflected in most of  
* the operations of the <tt>HashMap</tt> class, including  
* <tt>get</tt> and <tt>put</tt>). The expected number of entries in  
* the map and its load factor should be taken into account when  
* setting its initial capacity, so as to minimize the number of  
* rehash operations. If the initial capacity is greater than the  
* maximum number of entries divided by the load factor, no rehash  
* operations will ever occur.  
*
```

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

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

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

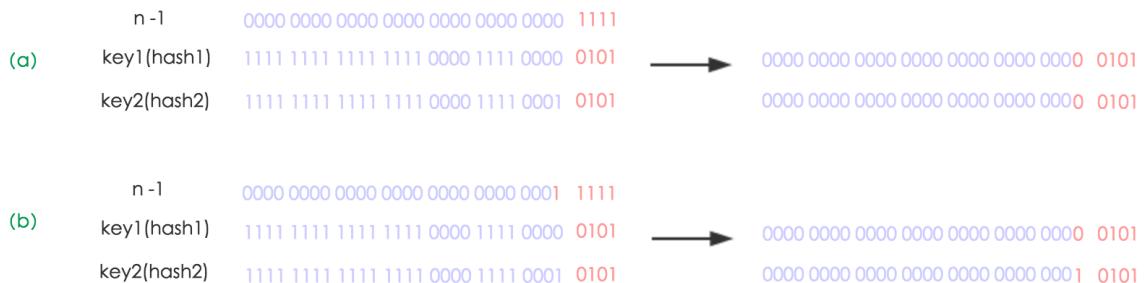
21.那扩容机制了解吗？

HashMap是基于数组+链表和红黑树实现的，但用于存放key值的桶数组的长度是固定的，由初始化参数确定。

那么，随着数据的插入数量增加以及负载因子的作用下，就需要扩容来存放更多的数据。而扩容中有一个非常重要的点，就是jdk1.8中的优化操作，可以不需要再重新计算每一个元素的哈希值。

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

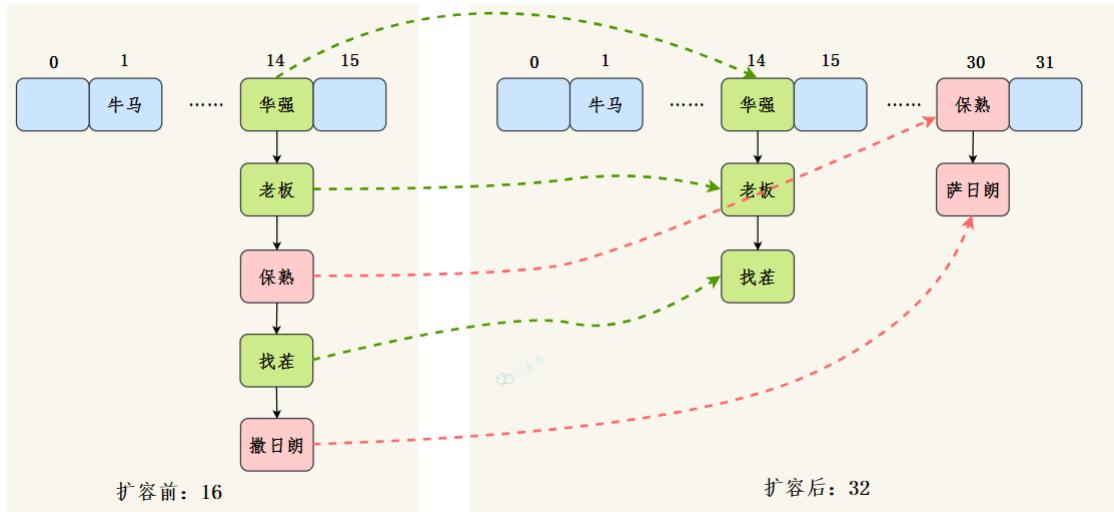
看下这张图，n为table的长度，图 a 表示扩容前的key1和key2两种key确定索引的位置，图 b 表示扩容后key1和key2两种key确定索引位置。



元素在重新计算hash之后，因为n变为2倍，那么n-1的mask范围在高位多1bit(红色)，因此新的index就会发生这样的变化：

$$0101 = 5 \xrightarrow{\text{resize}} \begin{array}{c} 0 \text{ } 0101 = 5 \\ 1 \text{ } 0101 = 21 = 5 + 16 \end{array} \quad \begin{array}{l} \text{原位置} \\ \text{原位置} + \text{oldCap} \end{array}$$

所以在扩容时，只需要看原来的hash值新增的那一位是0还是1就行了，是0的话索引没变，是1的化变成 **原索引+oldCap**，看看如16扩容为32的示意图：



扩容节点迁移主要逻辑：

```
if (e.next == null)
    //未形成链表，直接分配到新table
    newTab[e.hash & (newCap - 1)] = e;
else if (e instanceof TreeNode)
    //树节点分割
    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
else { // preserve order
    //链表拆成两个链表lo、hi，两个链表的头、尾节点
    Node<K,V> loHead = null, loTail = null;
    Node<K,V> hiHead = null, hiTail = null;
    Node<K,V> next;
    do {
        //遍历链表，将节点放到相应的新链表
        next = e.next;
        if ((e.hash & oldCap) == 0) {
            if (loTail == null)
                loHead = e;
            else
                loTail.next = e;
            loTail = e;
        }
        else {
            if (hiTail == null)
                hiHead = e;
            else
                hiTail.next = e;
            hiTail = e;
        }
    } while ((e = next) != null);
    //lo链表放到新table原位置
    if (loTail != null) {
        loTail.next = null;
        newTab[j] = loHead;
    }
    //hi链表放到j+oldCap位置
    if (hiTail != null) {
        hiTail.next = null;
        newTab[j + oldCap] = hiHead;
    }
}
```

22.jdk1.8对HashMap主要做了哪些优化呢？为什么？

jdk1.8 的HashMap主要有五点优化：

1. 数据结构：数组 + 链表改成了数组 + 链表或红黑树

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

2. 链表插入方式：链表的插入方式从头插法改成了尾插法

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

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

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

原因：提高扩容的效率，更快地扩容。

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

5. 散列函数：1.7 做了四次移位和四次异或，jdk1.8 只做一次。

原因：做 4 次的话，边际效用也不大，改为一次，提升效率。

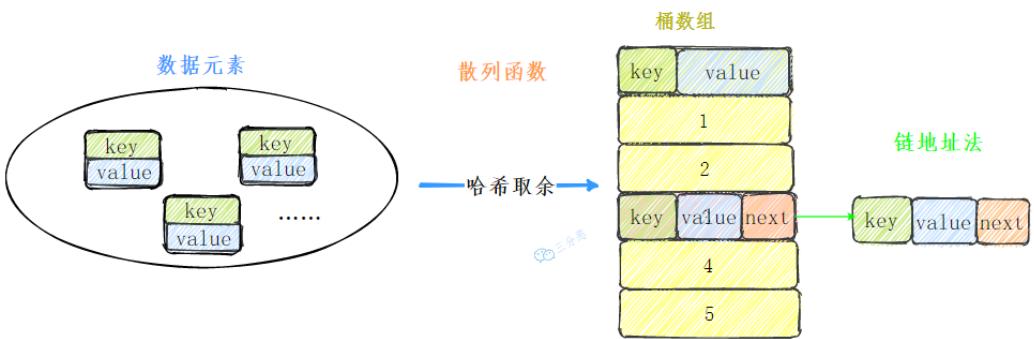
23. 你能自己设计实现一个HashMap吗？

这道题 **快手** 常考。

不要慌，红黑树版咱们多半是写不出来，但是数组+链表版还是问题不大的，详细可见：[手写HashMap，快手面试官直呼内行！](#)。

整体的设计：

- 散列函数： hashCode() + 除留余数法
- 冲突解决：链地址法
- 扩容：节点重新hash获取位置



完整代码:

```
/***
 * @Author 三分恶
 * @Date 2021/11/21
 * @Description 自己实现HashMap
 */
public class ThirdHashMap<K, V> {

    /**
     * 节点类
     *
     * @param <K>
     * @param <V>
     */
    class Node<K, V> {
        //键值对
        private K key;
        private V value;

        //链表，后继
        private Node<K, V> next;

        public Node(K key, V value) {
            this.key = key;
            this.value = value;
        }

        public Node(K key, V value, Node<K, V> next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }

    //默认容量
    final int DEFAULT_CAPACITY = 16;
    //负载因子
    final float LOAD_FACTOR = 0.75f;
    //HashMap的大小
    private int size;
    //桶数组
    Node<K, V>[] buckets;

    /**
     * 无参构造器，设置桶数组默认容量
     */
    public ThirdHashMap() {
        buckets = new Node[DEFAULT_CAPACITY];
        size = 0;
    }

    /**
     * 有参构造器，指定桶数组容量
     *
     * @param capacity
     */
    public ThirdHashMap(int capacity) {
        buckets = new Node[capacity];
        size = 0;
    }

    /**
     * 哈希函数，获取地址
     *
```

```
    */
    * @param key
    * @return
    */
    private int getIndex(K key, int length) {
        //获取hash code
        int hashCode = key.hashCode();
        //和桶数组长度取余
        int index = hashCode % length;
        return Math.abs(index);
    }

    /**
     * put方法
     *
     * @param key
     * @param value
     * @return
     */
    public void put(K key, V value) {
        //判断是否需要进行扩容
        if (size >= buckets.length * LOAD_FACTOR) resize();
        putVal(key, value, buckets);
    }

    /**
     * 将元素存入指定的node数组
     *
     * @param key
     * @param value
     * @param table
     */
    private void putVal(K key, V value, Node<K, V>[] table) {
        //获取位置
        int index = getIndex(key, table.length);
        Node node = table[index];
        //插入的位置为空
        if (node == null) {
            table[index] = new Node<>(key, value);
            size++;
            return;
        }
        //插入位置不为空，说明发生冲突，使用链地址法，遍历链表
        while (node != null) {
            //如果key相同，就覆盖掉
            if ((node.key.hashCode() == key.hashCode())
                && (node.key == key || node.key.equals(key))) {
                node.value = value;
                return;
            }
            node = node.next;
        }
        //当前key不在链表中，插入链表头部
        Node newNode = new Node(key, value, table[index]);
        table[index] = newNode;
        size++;
    }

    /**
     * 扩容
     */
    private void resize() {
        //创建一个两倍容量的桶数组
        Node<K, V>[] newBuckets = new Node[buckets.length * 2];
        //将当前元素重新散列到新的桶数组
        rehash(newBuckets);
        buckets = newBuckets;
    }
}
```

```
    /**
     * 重新散列当前元素
     *
     * @param newBuckets
     */
    private void rehash(Node<K, V>[ ] newBuckets) {
        //map大小重新计算
        size = 0;
        //将旧的桶数组的元素全部刷到新的桶数组里
        for (int i = 0; i < buckets.length; i++) {
            //为空，跳过
            if (buckets[i] == null) {
                continue;
            }
            Node<K, V> node = buckets[i];
            while (node != null) {
                //将元素放入新数组
                putVal(node.key, node.value, newBuckets);
                node = node.next;
            }
        }
    }

    /**
     * 获取元素
     *
     * @param key
     * @return
     */
    public V get(K key) {
        //获取key对应的地址
        int index = getIndex(key, buckets.length);
        if (buckets[index] == null) return null;
        Node<K, V> node = buckets[index];
        //查找链表
        while (node != null) {
            if ((node.key.hashCode() == key.hashCode())
                && (node.key == key || node.key.equals(key))) {
                return node.value;
            }
            node = node.next;
        }
        return null;
    }

    /**
     * 返回HashMap大小
     *
     * @return
     */
    public int size() {
        return size;
    }
}
```

24. HashMap 是线程安全的吗？多线程下会有什么问题？

HashMap不是线程安全的，可能会发生这些问题：

- 多线程下扩容死循环。JDK1.7 中的 HashMap 使用头插法插入元素，在多线程的环境下，扩容的时候有可能导致环形链表的出现，形成死循环。因此，JDK1.8 使用尾插法插入元素，在扩容时会保持链表元素原本的顺序，不会出现环形链表的问题。
- 多线程的 put 可能导致元素的丢失。多线程同时执行 put 操作，如果计算出来的索引位置是相同的，那会造成前一个 key 被后一个 key 覆盖，从而导致元素的丢失。此问题在 JDK 1.7 和 JDK 1.8 中都存在。
- put 和 get 并发时，可能导致 get 为 null。线程 1 执行 put 时，因为元素个数超出 threshold 而导致 rehash，线程 2 此时执行 get，有可能导致这个问题。这个问题在 JDK 1.7 和 JDK 1.8 中都存在。

25.有什么办法能解决HashMap线程不安全的问题呢？

Java 中有 HashTable、Collections.synchronizedMap、以及 ConcurrentHashMap 可以实现线程安全的 Map。

- HashTable 是直接在操作方法上加 synchronized 关键字，锁住整个table数组，粒度比较大；
- Collections.synchronizedMap 是使用 Collections 集合工具的内部类，通过传入 Map 封装出一个 SynchronizedMap 对象，内部定义了一个对象锁，方法内通过对对象锁实现；
- ConcurrentHashMap 在 jdk1.7 中使用分段锁，在 jdk1.8 中使用 CAS+synchronized。

26.能具体说一下ConcurrentHashmap的实现吗？

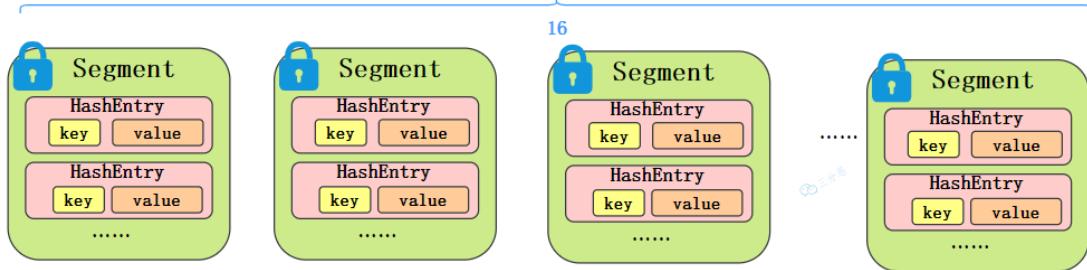
ConcurrentHashmap线程安全在jdk1.7版本是基于 **分段锁** 实现，在jdk1.8是基于 **CAS+synchronized** 实现。

H5 1.7分段锁

从结构上说，1.7版本的ConcurrentHashMap采用分段锁机制，里面包含一个Segment数组，Segment继承于ReentrantLock，Segment则包含HashEntry的数组，HashEntry本身就是一个链表的结构，具有保存key、value的能力能指向下一个节点的指针。

实际上就是相当于每个Segment都是一个HashMap，默认的Segment长度是16，也就是支持16个线程的并发写，Segment之间相互不会受到影响。

ConcurrentHashMap



put流程

整个流程和HashMap非常类似，只不过是先定位到具体的Segment，然后通过ReentrantLock去操作而已，后面的流程，就和HashMap基本上是一样的。

1. 计算hash，定位到segment，segment如果是空就先初始化
2. 使用ReentrantLock加锁，如果获取锁失败则尝试自旋，自旋超过次数就阻塞获取，保证一定获取锁成功
3. 遍历HashEntry，就是和HashMap一样，数组中key和hash一样就直接替换，不存在就再插入链表，链表同样操作

get流程

get也很简单，key通过hash定位到segment，再遍历链表定位到具体的元素上，需要注意的是value是volatile的，所以get是不需要加锁的。

H5 1.8 CAS+synchronized

jdk1.8实现线程安全不是在数据结构上下功夫，它的数据结构和HashMap是一样的，数组+链表+红黑树。它实现线程安全的关键点在于put流程。

put流程

1. 首先计算hash，遍历node数组，如果node是空的话，就通过CAS+自旋的方式初始化

```
1 | tab = initTable();
```

node数组初始化：

```
1 | private final Node<K,V>[] initTable() {  
2 |     Node<K,V>[] tab; int sc;  
3 |     while ((tab = table) == null || tab.length == 0) {
```

```

4         //如果正在初始化或者扩容
5         if ((sc = sizeCtl) < 0)
6             //等待
7                 Thread.yield(); // lost initialization race;
8         just spin
9         else if (U.compareAndSwapInt(this, SIZECTL, sc,
10             -1)) { //CAS操作
11             try {
12                 if ((tab = table) == null || tab.length
13                     == 0) {
14                     int n = (sc > 0) ? sc :
15                         DEFAULT_CAPACITY;
16                     @SuppressWarnings("unchecked")
17                     Node<K,V>[] nt = (Node<K,V>[])new
18                     Node<?, ?>[n];
19                     table = tab = nt;
20                     sc = n - (n >>> 2);
21                 }
22             } finally {
23                 sizeCtl = sc;
24             }
25             break;
26         }
27     }
28     return tab;
29 }

```

2.如果当前数组位置是空则直接通过CAS自旋写入数据

```

1     static final <K,V> boolean casTabAt(Node<K,V>[] tab, int
2         i,
3             Node<K,V> c,
4             Node<K,V> v) {
5                 return U.compareAndSwapObject(tab, ((long)i <<
6                     ASHIFT) + ABASE, c, v);
7             }

```

3. 如果hash==MOVED, 说明需要扩容, 执行扩容

```

1     else if ((fh = f.hash) == MOVED)
2             tab = helpTransfer(tab, f);

```

```

1     final Node<K,V>[ ] helpTransfer(Node<K,V>[ ] tab, Node<K,V>
f) {
2         Node<K,V>[ ] nextTab; int sc;
3         if (tab != null && (f instanceof ForwardingNode) &&
4             (nextTab = ((ForwardingNode<K,V>)f).nextTable) !=
null) {
5             int rs = resizeStamp(tab.length);
6             while (nextTab == nextTable && table == tab &&
7                 (sc = sizeCtl) < 0) {
8                 if ((sc >> RESIZE_STAMP_SHIFT) != rs || sc
9                     == rs + 1 ||
10                     sc == rs + MAX_RESIZERS || transferIndex
11                     <= 0)
12                     break;
13                 if (U.compareAndSwapInt(this, SIZECTL, sc, sc
14                     + 1)) {
15                     transfer(tab, nextTab);
16                     break;
17                 }
18             }
19         }

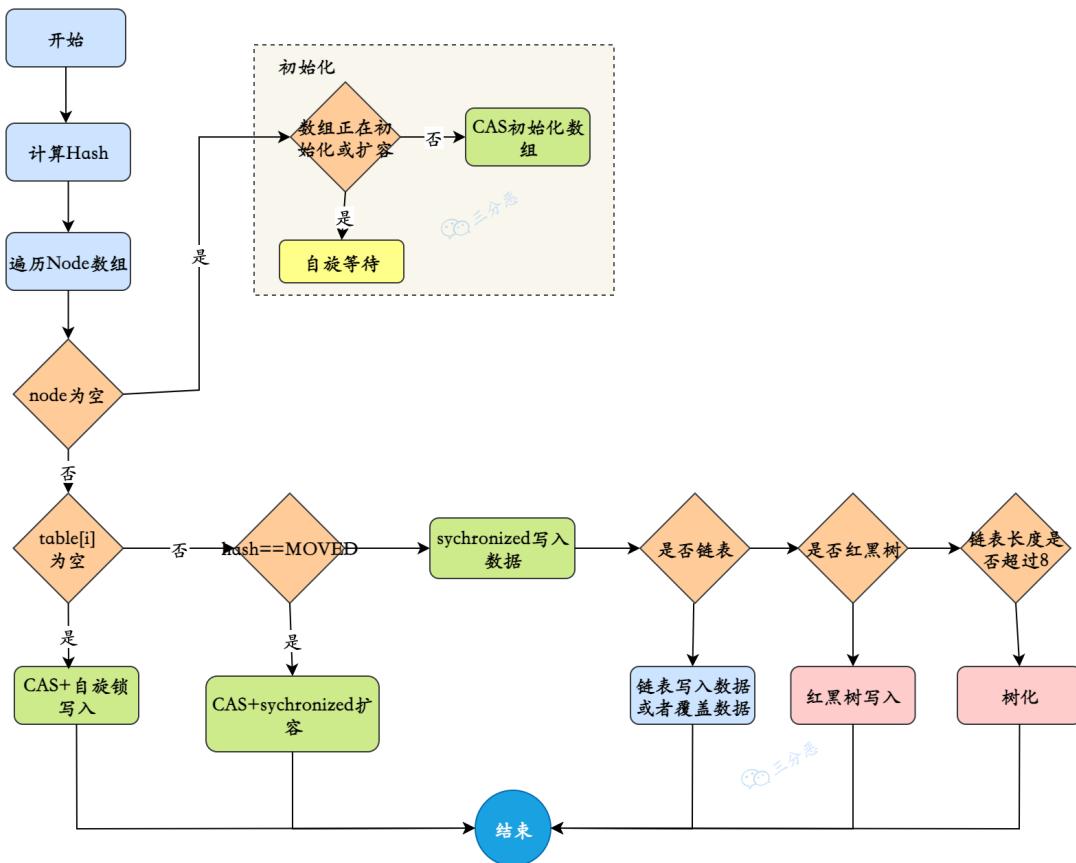
```

4. 如果都不满足，就使用synchronized写入数据，写入数据同样判断链表、红黑树，链表写入和HashMap的方式一样，key hash一样就覆盖，反之就尾插法，链表长度超过8就转换成红黑树

```

1 synchronized (f){
2     .....
3 }

```



get查询

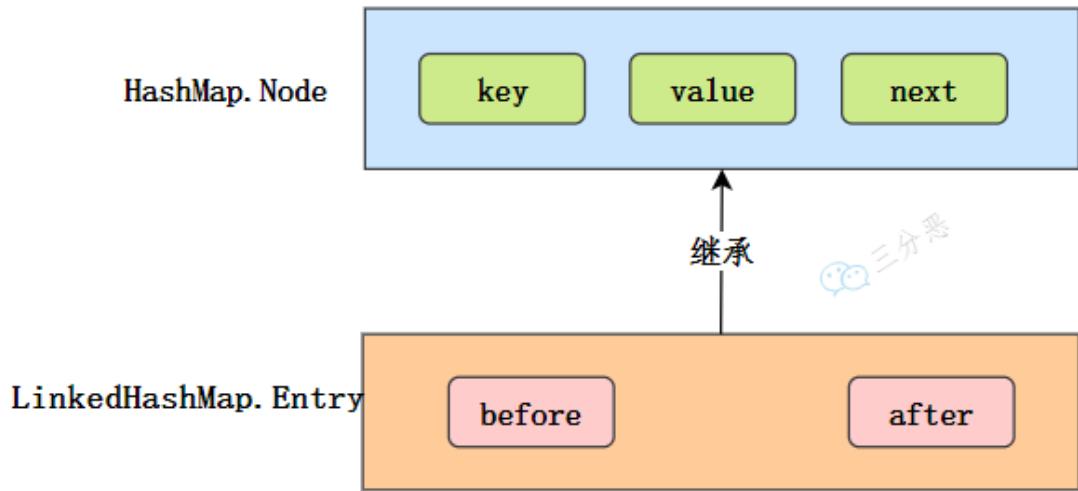
get很简单，和HashMap基本相同，通过key计算位置，table该位置key相同就返回，如果是红黑树按照红黑树获取，否则就遍历链表获取。

27. HashMap 内部节点是有序的吗？

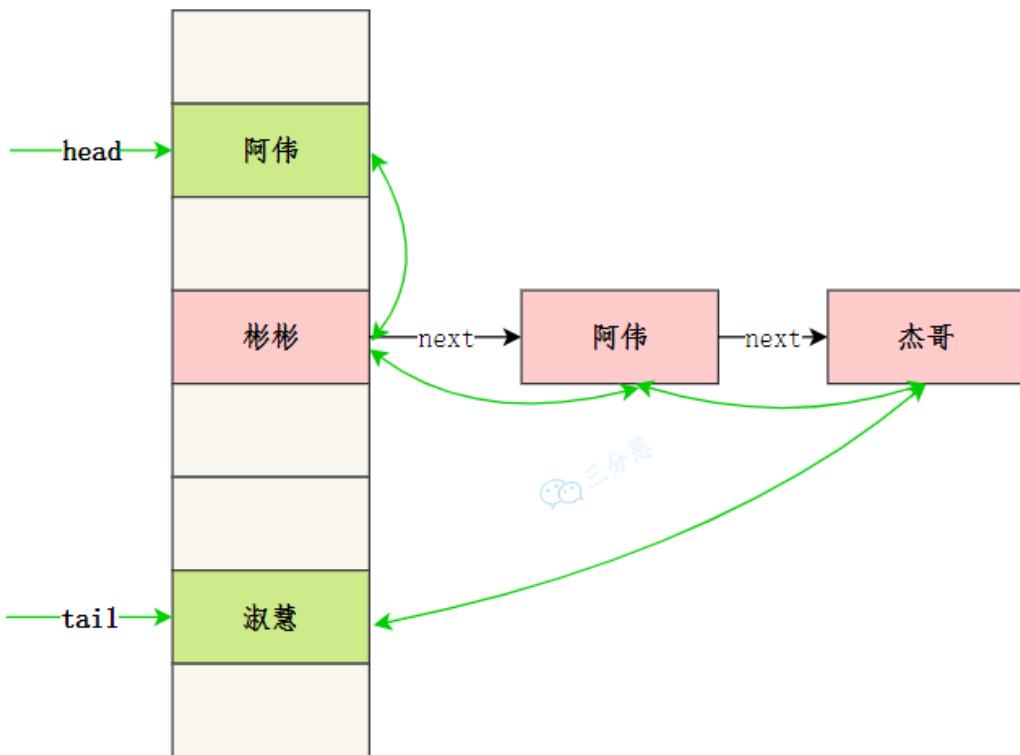
HashMap是无序的，根据 hash 值随机插入。如果想使用有序的Map，可以使用 LinkedHashMap 或者 TreeMap。

28. 讲讲 LinkedHashMap 怎么实现有序的？

LinkedHashMap维护了一个双向链表，有头尾节点，同时 LinkedHashMap 节点 Entry 内部除了继承 HashMap 的 Node 属性，还有 before 和 after 用于标识前置节点和后置节点。

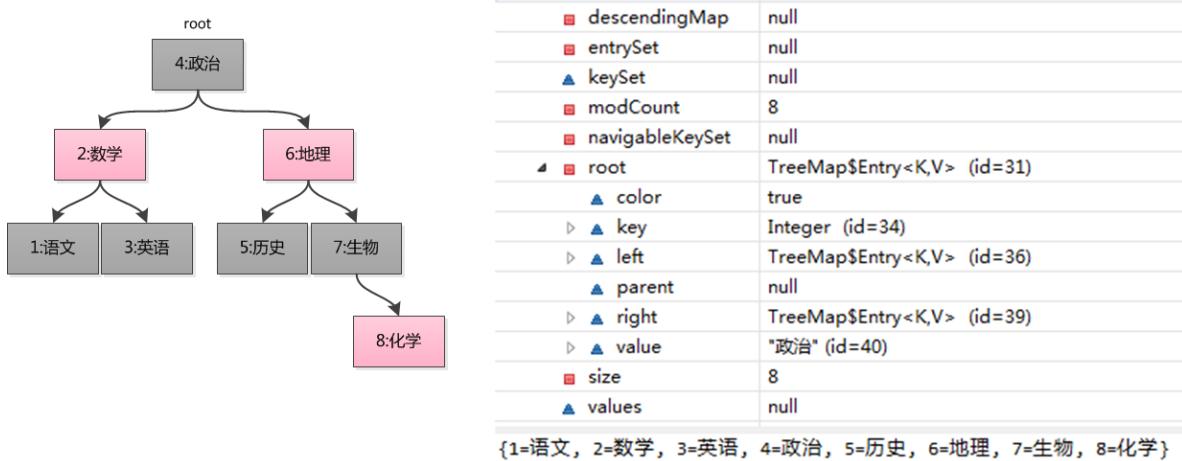


可以实现按插入的顺序或访问顺序排序。



29.讲讲 TreeMap 怎么实现有序的？

`TreeMap` 是按照 Key 的自然顺序或者 `Comprator` 的顺序进行排序，内部是通过红黑树来实现。所以要么 key 所属的类实现 `Comparable` 接口，或者自定义一个实现了 `Comparator` 接口的比较器，传给 `TreeMap` 用于 key 的比较。



Set

Set面试没啥好问的，拿HashSet来凑个数。

30.讲讲HashSet的底层实现？

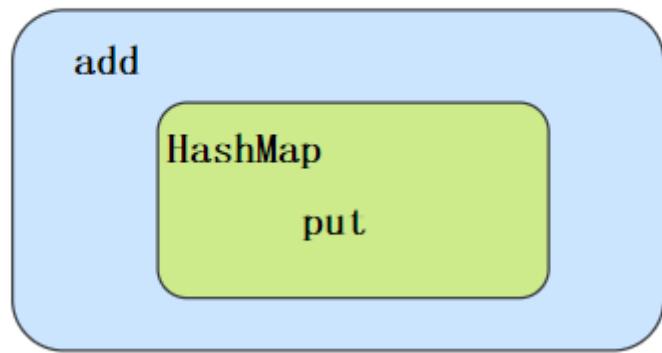
HashSet 底层就是基于 HashMap 实现的。（ HashSet 的源码非常非常少，因为除了 clone() 、 writeObject() 、 readObject() 是 HashSet 自己不得不实现之外，其他方法都是直接调用 HashMap 中的方法。）

HashSet 的 add 方法，直接调用 HashMap 的 put 方法，将添加的元素作为 key， new 一个 Object 作为 value，直接调用 HashMap 的 put 方法，它会根据返回值是否为空来判断是否插入元素成功。

```

1 |     public boolean add(E e) {
2 |         return map.put(e, PRESENT)==null;
3 |
    }
```

HashSet



而在HashMap的putVal方法中，进行了一系列判断，最后的结果是，只有在key在table数组中不存在的时候，才会返回插入的值。

```
1  if (e != null) { // existing mapping for key
2      V oldValue = e.value;
3      if (!onlyIfAbsent || oldValue == null)
4          e.value = value;
5      afterNodeAccess(e);
6      return oldValue;
7  }
```

参考：

- [1]. [一个HashMap跟面试官扯了半个小时](#)
- [2]. [《大厂面试》—Java 集合连环30问](#)
- [3]. [面经手册 · 第4篇《HashMap数据插入、查找、删除、遍历，源码分析》](#)
- [4]. [《我想进大厂》之Java基础夺命连环16问](#)
- [5]. [数据结构之LinkedHashMap](#)
- [6]. [面经手册 · 第3篇《HashMap核心知识，扰动函数、负载因子、扩容链表拆分，深度学习》](#)
- [7]. [面试官：为什么 HashMap 的加载因子是0.75？](#)
- [8]. [面试旧敌之红黑树（直白介绍深入理解）](#)
- [9]. [Java TreeMap工作原理及实现](#)
- [10]. [手写HashMap，快去面试官直呼内行！](#)
- [11]. [Java 8系列之重新认识HashMap](#)

关注公众号：三分恶

手册更新动态

即刻送达



添加个人微信：ThirdFighter

技术交流

加大佬云集微信群



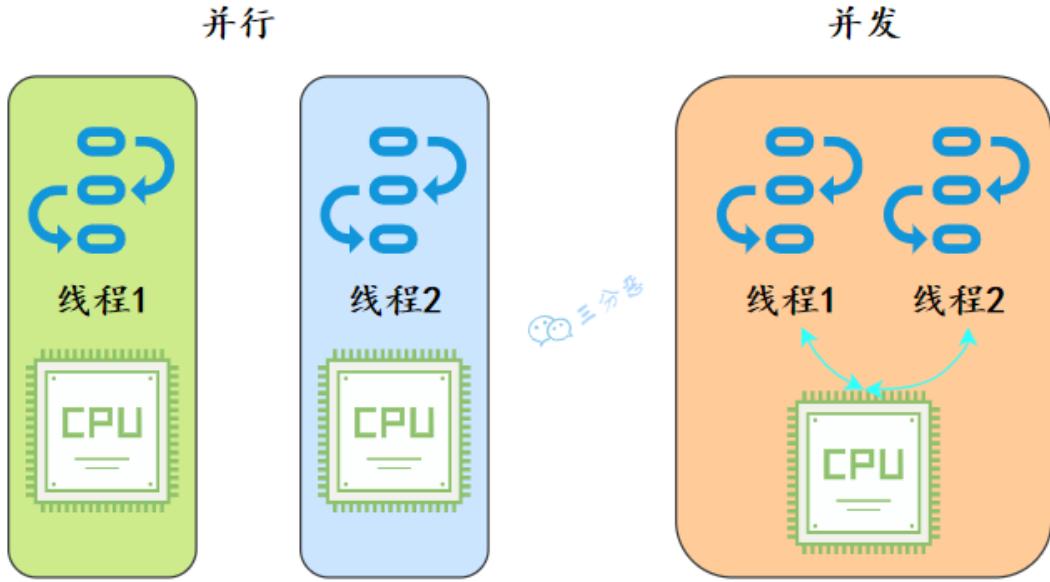
三、Java并发

基础

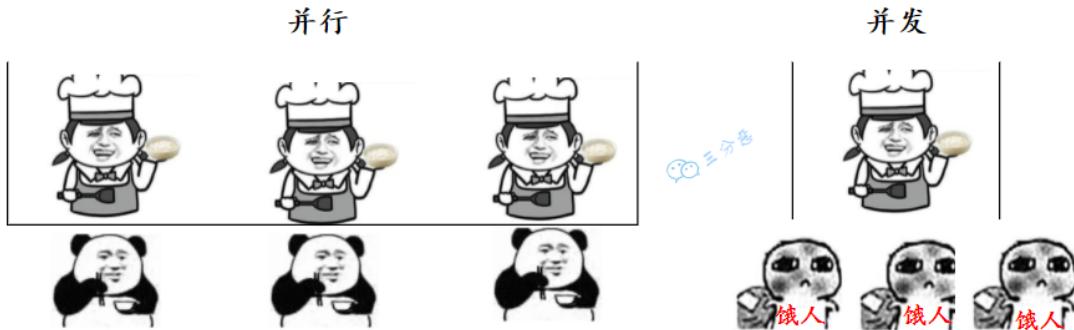
1. 并行跟并发有什么区别？

从操作系统的角度来看，线程是CPU分配的最小单位。

- 并行就是同一时刻，两个线程都在执行。这就要求有两个CPU去分别执行两个线程。
- 并发就是同一时刻，只有一个执行，但是一个时间段内，两个线程都执行了。并发的实现依赖于CPU切换线程，因为切换的时间特别短，所以基本对于用户是无感知的。



就好像我们去食堂打饭，并行就是我们在多个窗口排队，几个阿姨同时打菜；并发就是我们挤在一个窗口，阿姨给这个打一勺，又手忙脚乱地给那个打一勺。



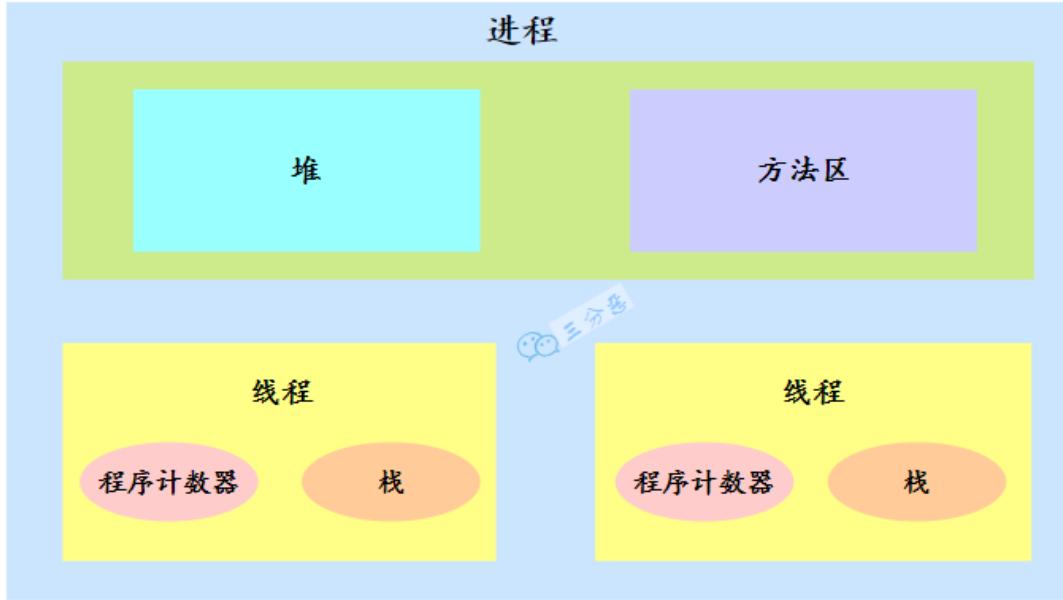
2. 说说什么是进程和线程？

要说线程，必须得先说说进程。

- **进程：**进程是代码在数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位。
- **线程：**线程是进程的一个执行路径，一个进程中至少有一个线程，进程中的多个线程共享进程的资源。

操作系统在分配资源时是把资源分配给进程的，但是CPU资源比较特殊，它是被分配到线程的，因为真正要占用CPU运行的是线程，所以也说线程是CPU分配的基本单位。

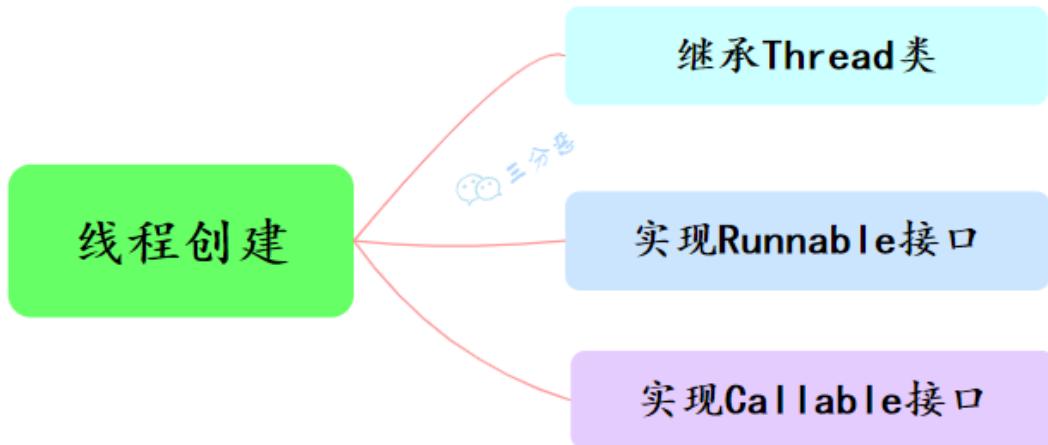
比如在Java中，当我们启动 main 函数其实就启动了一个JVM进程，而 main 函数所在的线程就是这个进程中的一个线程，也称主线程。



一个进程中多个线程，多个线程共用进程的堆和方法区资源，但是每个线程有自己的程序计数器和栈。

3.说说线程有几种创建方式？

Java中创建线程主要有三种方式，分别为继承Thread类、实现Runnable接口、实现 Callable接口。



- 继承Thread类，重写run()方法，调用start()方法启动线程

```
1 | public class ThreadTest {
```

```
2
3     /**
4      * 继承Thread类
5     */
6     public static class MyThread extends Thread {
7         @Override
8         public void run() {
9             System.out.println("This is child thread");
10        }
11    }
12
13    public static void main(String[] args) {
14        MyThread thread = new MyThread();
15        thread.start();
16    }
17}
18
```

- 实现 Runnable 接口，重写run()方法

```
1 public class RunnableTask implements Runnable {
2     public void run() {
3         System.out.println("Runnable!");
4     }
5
6     public static void main(String[] args) {
7         RunnableTask task = new RunnableTask();
8         new Thread(task).start();
9     }
10}
11
```

上面两种都是没有返回值的，但是如果我们需要获取线程的执行结果，该怎么办呢？

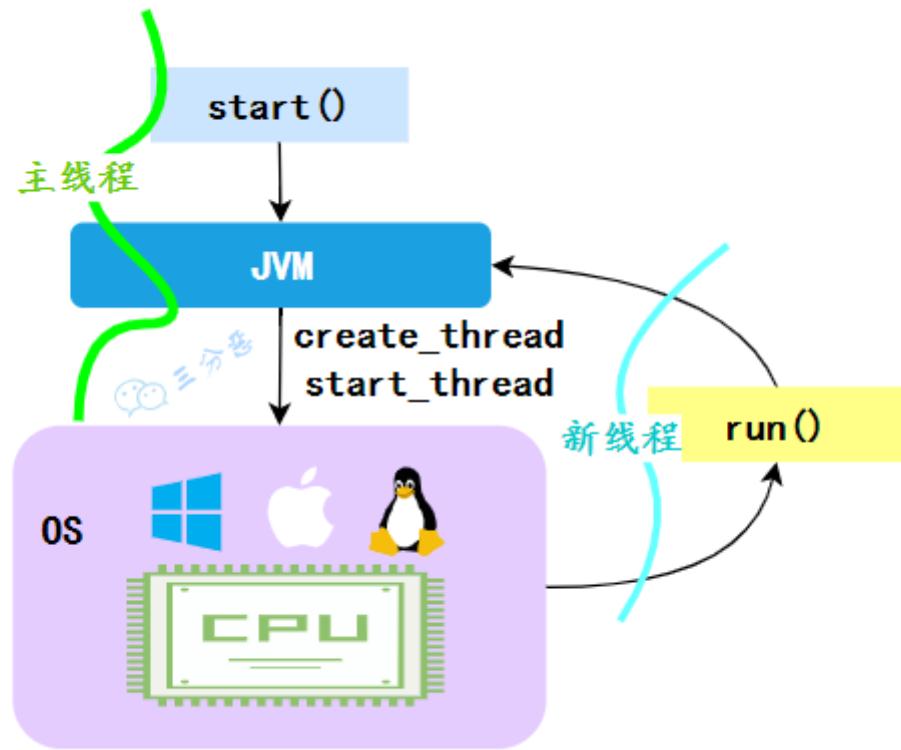
- 实现Callable接口，重写call()方法，这种方式可以通过FutureTask获取任务执行的返回值

```
1 public class CallerTask implements Callable<String> {
2     public String call() throws Exception {
3         return "Hello, i am running!";
```

```
4     }
5
6     public static void main(String[] args) {
7         //创建异步任务
8         FutureTask<String> task=new FutureTask<String>(new
9             CallerTask());
10        //启动线程
11        new Thread(task).start();
12        try {
13            //等待执行完成，并获取返回结果
14            String result=task.get();
15            System.out.println(result);
16        } catch (InterruptedException e) {
17            e.printStackTrace();
18        } catch (ExecutionException e) {
19            e.printStackTrace();
20        }
21    }
22 }
```

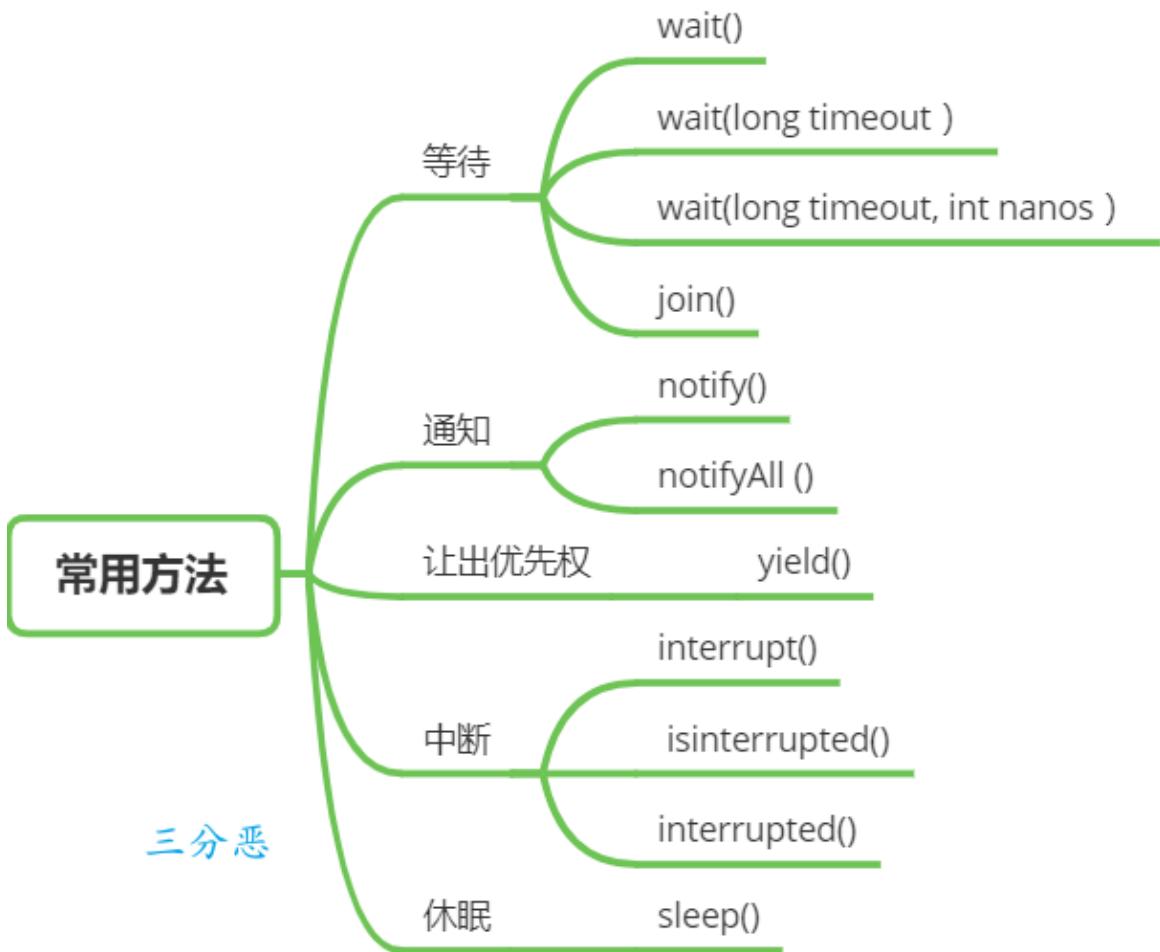
4.为什么调用start()方法时会执行run()方法，那怎么不直接调用run()方法？

JVM执行start方法，会先创建一条线程，由创建出来的新线程去执行thread的run方法，这才起到多线程的效果。



为什么我们不能直接调用`run()`方法？也很清楚，如果直接调用`Thread`的`run()`方法，那么`run`方法还是运行在主线程中，相当于顺序执行，就起不到多线程的效果。

5. 线程有哪些常用的调度方法？



线程等待与通知

在Object类中有一些函数可以用于线程的等待与通知。

- **wait()**: 当一个线程A调用一个共享变量的 wait()方法时， 线程A会被阻塞挂起，发生下面几种情况才会返回：
 - (1) 线程A调用了共享对象 notify()或者 notifyAll()方法；
 - (2) 其他线程调用了线程A的 interrupt() 方法， 线程A抛出 InterruptedException异常返回。
- **wait(long timeout)** : 这个方法相比 wait() 方法多了一个超时参数，它的不同之处在于，如果线程A调用共享对象的wait(long timeout)方法后，没有在指定的 timeout ms时间内被其它线程唤醒，那么这个方法还是会因为超时而返回。
- **wait(long timeout, int nanos)**, 其内部调用的是 wait(long timeout) 函数。

上面是线程等待的方法，而唤醒线程主要是下面两个方法：

- **notify()** : 一个线程A调用共享对象的 notify() 方法后，会唤醒一个在这个共享变量上调用 wait 系列方法后被挂起的线程。一个共享变量上可能会有多个线程在等待，具体唤醒哪个等待的线程是随机的。

- `notifyAll()`：不同于在共享变量上调用 `notify()` 函数会唤醒被阻塞到该共享变量上的一个线程，`notifyAll()`方法则会唤醒所有在该共享变量上由于调用 `wait` 系列方法而被挂起的线程。

Thread类也提供了一个方法用于等待的方法：

- `join()`：如果一个线程A执行了`thread.join()`语句，其含义是：当前线程A等待`thread`线程终止之后才从`thread.join()`返回。

线程休眠

- `sleep(long millis)` :Thread类中的静态方法，当一个执行中的线程A调用了Thread的`sleep`方法后，线程A会暂时让出指定时间的执行权，但是线程A所拥有的监视器资源，比如锁还是持有不让出的。指定的睡眠时间到了后该函数会正常返回，接着参与CPU的调度，获取到CPU资源后就可以继续运行。

让出优先权

- `yield()`：Thread类中的静态方法，当一个线程调用 `yield` 方法时，实际就是在暗示线程调度器当前线程请求让出自己的CPU，但是线程调度器可以无条件忽略这个暗示。

线程中断

Java 中的线程中断是一种线程间的协作模式，通过设置线程的中断标志并不能直接终止该线程的执行，而是被中断的线程根据中断状态自行处理。

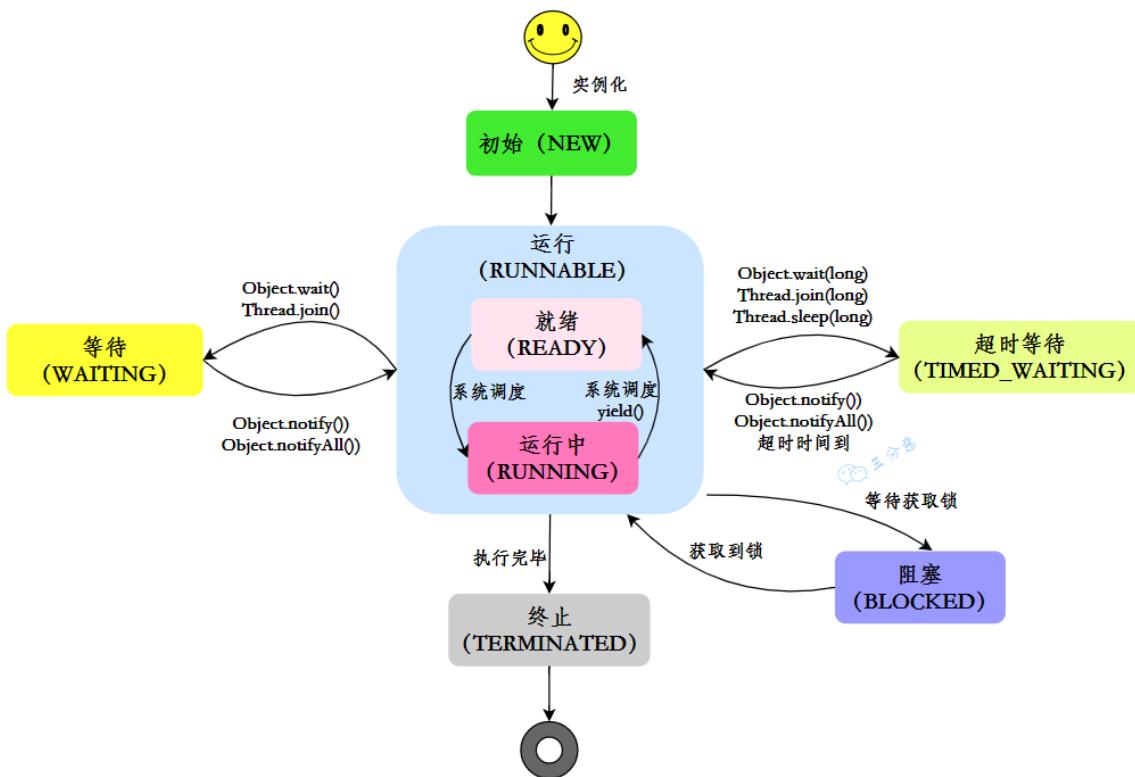
- `void interrupt()`：中断线程，例如，当线程A运行时，线程B可以调用线程`interrupt()`方法来设置线程的中断标志为`true` 并立即返回。设置标志仅仅是设置标志，线程A实际并没有被中断，会继续往下执行。
- `boolean isInterrupted()` 方法：检测当前线程是否被中断。
- `boolean interrupted()` 方法：检测当前线程是否被中断，与 `isInterrupted` 不同的是，该方法如果发现当前线程被中断，则会清除中断标志。

6.线程有几种状态？

在Java中，线程共有六种状态：

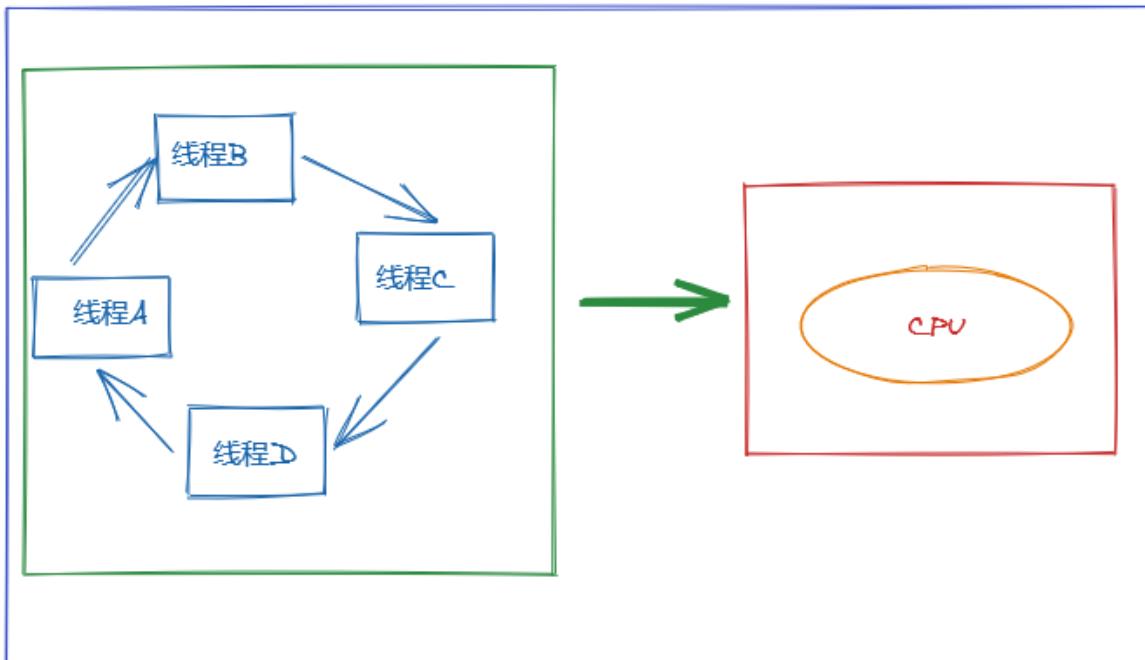
状态	说明
NEW	初始状态：线程被创建，但还没有调用start()方法
RUNNABLE	运行状态：Java线程将操作系统中的就绪和运行两种状态笼统的称作“运行”
BLOCKED	阻塞状态：表示线程阻塞于锁
WAITING	等待状态：表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态：该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态：表示当前线程已经执行完毕

线程在自身的生命周期中，并不是固定地处于某个状态，而是随着代码的执行在不同的状态之间进行切换，Java线程状态变化如图示：

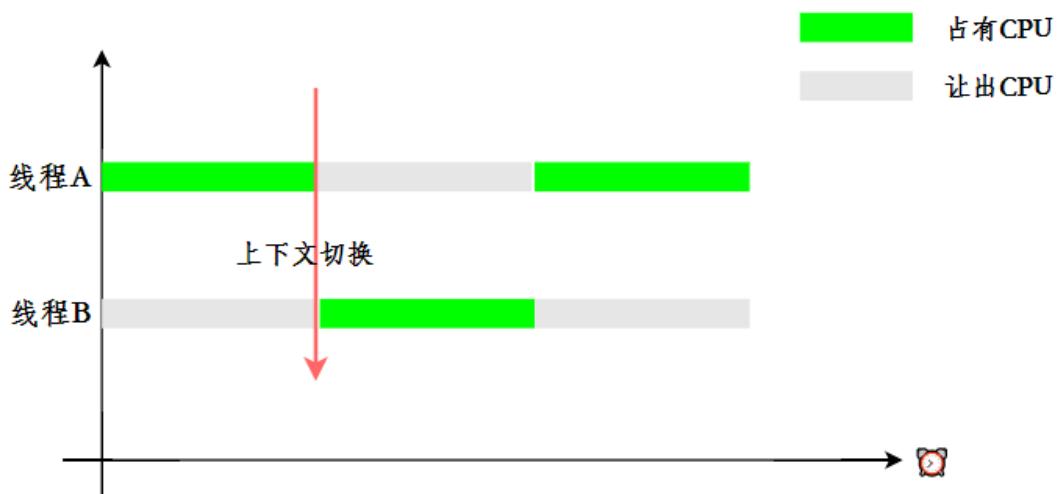


7.什么是线程上下文切换？

使用多线程的目的是为了充分利用CPU，但是我们知道，并发其实是一个CPU来应付多个线程。



为了让用户感觉多个线程是在同时执行的，CPU 资源的分配采用了时间片轮转也就是给每个线程分配一个时间片，线程在时间片内占用 CPU 执行任务。当线程使用完时间片后，就会处于就绪状态并让出 CPU 让其他线程占用，这就是上下文切换。



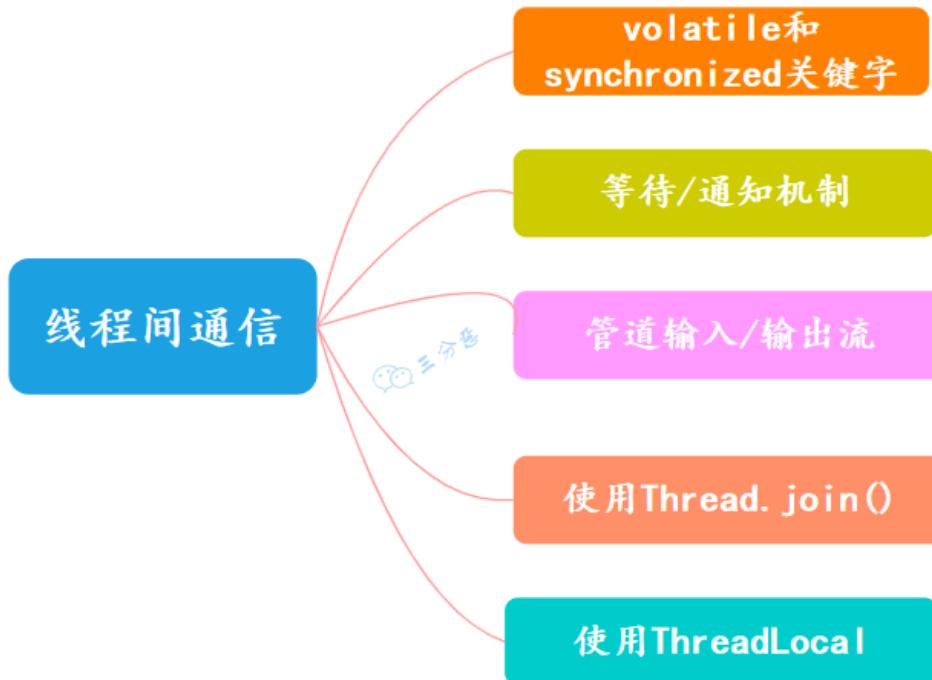
8. 守护线程了解吗？

Java中的线程分为两类，分别为 daemon 线程（守护线程）和 user 线程（用户线程）。

在JVM 启动时会调用 main 函数，main函数所在的钱程就是一个用户线程。其实在 JVM 内部同时还启动了很多守护线程，比如垃圾回收线程。

那么守护线程和用户线程有什么区别呢？区别之一是当最后一个非守护线程结束时，JVM会正常退出，而不管当前是否存在守护线程，也就是说守护线程是否结束并不影响JVM退出。换而言之，只要有一个用户线程还没结束，正常情况下JVM就不会退出。

9. 线程间有哪些通信方式？



- volatile和synchronized关键字

关键字volatile可以用来修饰字段（成员变量），就是告知程序任何对该变量的访问均需要从共享内存中获取，而对它的改变必须同步刷新回共享内存，它能保证所有线程对变量访问的可见性。

关键字synchronized可以修饰方法或者以同步块的形式来进行使用，它主要确保多个线程在同一个时刻，只能有一个线程处于方法或者同步块中，它保证了线程对变量访问的可见性和排他性。

- 等待/通知机制

可以通过Java内置的等待/通知机制（wait()/notify()）实现一个线程修改一个对象的值，而另一个线程感知到了变化，然后进行相应的操作。

- 管道输入/输出流

管道输入/输出流和普通的文件输入/输出流或者网络输入/输出流不同之处在于，它主要用于线程之间的数据传输，而传输的媒介为内存。

管道输入/输出流主要包括了如下4种具体实现：`PipedOutputStream`、`PipedInputStream`、`PipedReader`和`PipedWriter`，前两种面向字节，而后两种面向字符。

- 使用`Thread.join()`

如果一个线程A执行了`thread.join()`语句，其含义是：当前线程A等待`thread`线程终止之后才从`thread.join()`返回。。线程`Thread`除了提供`join()`方法之外，还提供了`join(long millis)`和`join(long millis,int nanos)`两个具备超时特性的方法。

- 使用`ThreadLocal`

`ThreadLocal`，即线程变量，是一个以`ThreadLocal`对象为键、任意对象为值的存储结构。这个结构被附带在线程上，也就是说一个线程可以根据一个`ThreadLocal`对象查询到绑定在这个线程上的一个值。

可以通过`set(T)`方法来设置一个值，在当前线程下再通过`get()`方法获取到原先设置的值。

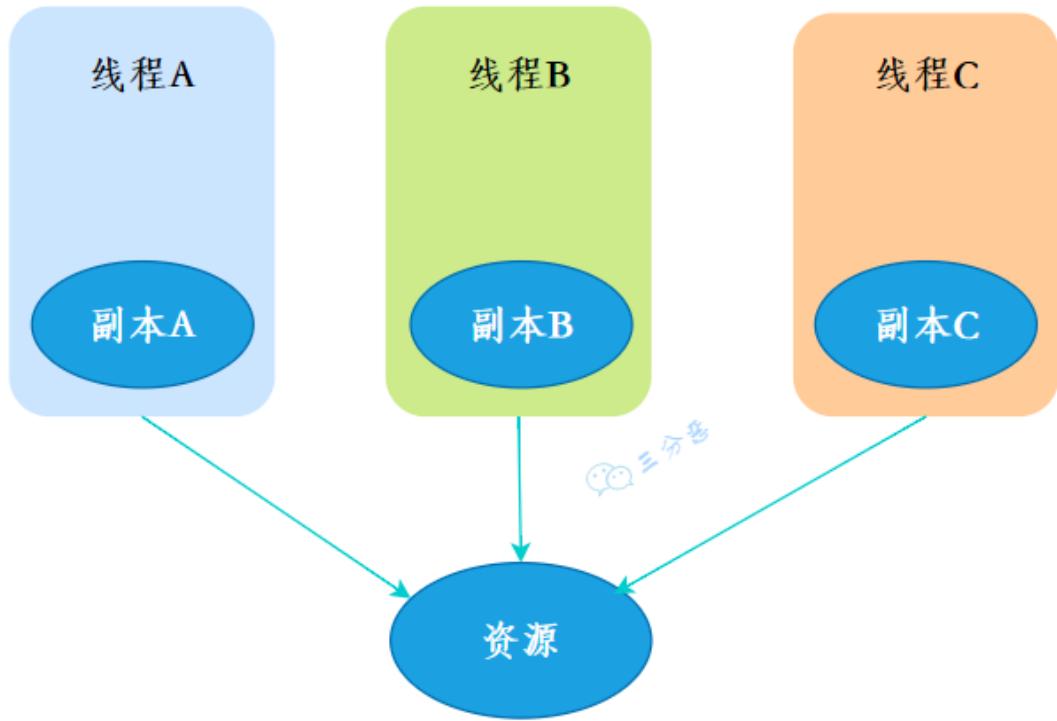
关于多线程，其实很大概率还会出一些笔试题，比如交替打印、银行转账、生产消费模型等等，后面老三会单独出一期来盘点一下常见的多线程笔试题。

ThreadLocal

`ThreadLocal`其实应用场景不是很多，但却是被炸了千百遍的面试老油条，涉及到多线程、数据结构、JVM，可问的点比较多，一定要拿下。

10. ThreadLocal是什么？

`ThreadLocal`，也就是线程本地变量。如果你创建了一个`ThreadLocal`变量，那么访问这个变量的每个线程都会有这个变量的一个本地拷贝，多个线程操作这个变量的时候，实际是操作自己本地内存里面的变量，从而起到线程隔离的作用，避免了线程安全问题。



- 创建

创建了一个ThreadLoca变量localVariable，任何一个线程都能并发访问localVariable。

```
1 | //创建一个ThreadLocal变量
2 | public static ThreadLocal<String> localVariable = new
   ThreadLocal<>();
```

- 写入

线程可以在任何地方使用localVariable，写入变量。

```
1 | localVariable.set("鄙人三某");
```

- 读取

线程在任何地方读取的都是它写入的变量。

```
1 | localVariable.get();
```

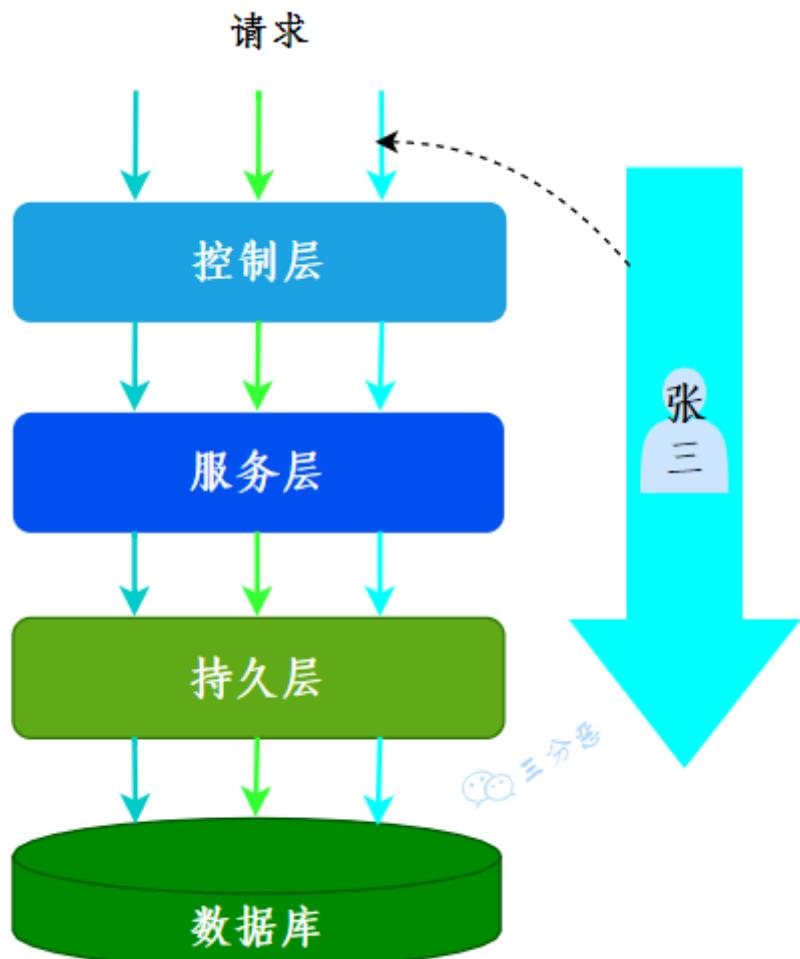
11.你在工作中用到过ThreadLocal吗？

有用到过的，用来做用户信息上下文的存储。

我们的系统应用是一个典型的MVC架构，登录后的用户每次访问接口，都会在请求头中携带一个token，在控制层可以根据这个token，解析出用户的基本信息。那么问题来了，假如在服务层和持久层都要用到用户信息，比如rpc调用、更新用户获取等等，那应该怎么办呢？

一种办法是显式定义用户相关的参数，比如账号、用户名……这样一来，我们可能需要大面积地修改代码，多少有点瓜皮，那该怎么办呢？

这时候我们就可以用到ThreadLocal，在控制层拦截请求把用户信息存入ThreadLocal，这样我们在任何一个地方，都可以取出ThreadLocal中存的用户数据。



很多其它场景的cookie、session等等数据隔离也都可以通过ThreadLocal去实现。

我们常用的数据库连接池也用到了ThreadLocal：

- 数据库连接池的连接交给ThreadLoca进行管理，保证当前线程的操作都是同一个Connnection。

12.ThreadLocal怎么实现的呢？

我们看一下ThreadLocal的set(T)方法，发现先获取到当前线程，再获取 ThreadLocalMap，然后把元素存到这个map中。

```
1 public void set(T value) {  
2     //获取当前线程  
3     Thread t = Thread.currentThread();  
4     //获取ThreadLocalMap  
5     ThreadLocalMap map = getMap(t);  
6     //将当前元素存入map  
7     if (map != null)  
8         map.set(this, value);  
9     else  
10        createMap(t, value);  
11 }
```

ThreadLocal实现的秘密都在这个 ThreadLocalMap 了，可以在Thread类中定义了一个类型为 ThreadLocal.ThreadLocalMap 的成员变量 threadLocals。

```
1 public class Thread implements Runnable {  
2     //ThreadLocal.ThreadLocalMap是Thread的属性  
3     ThreadLocal.ThreadLocalMap threadLocals = null;  
4 }
```

ThreadLocalMap既然被称为Map，那么毫无疑问它是<key,value>型的数据结构。我们都应该知道map的本质是一个个<key,value>形式的节点组成的数组，那ThreadLocalMap的节点是什么样的呢？

```

1      static class Entry extends
2          WeakReference<ThreadLocal<?>> {
3              /**
4               * The value associated with this ThreadLocal.
5               */
6              Object value;
7
8              //节点类
9              Entry(ThreadLocal<?> k, Object v) {
10                 //key赋值
11                 super(k);
12                 //value赋值
13                 value = v;
14             }
15         }

```

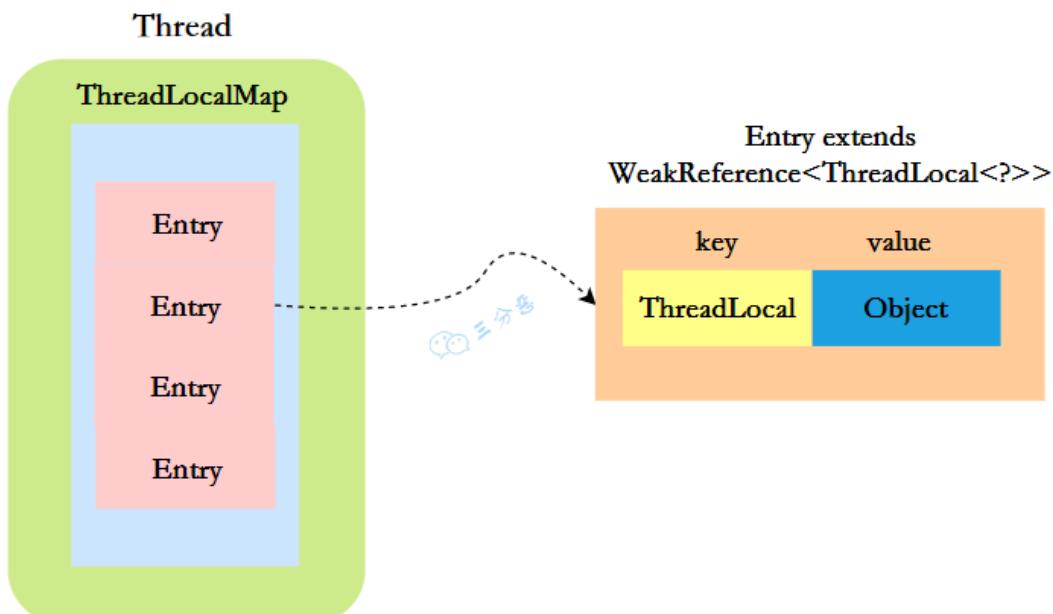
这里的节点，key可以简单地视作ThreadLocal，value为代码中放入的值，当然实际上key并不是ThreadLocal本身，而是它的一个弱引用，可以看到Entry的key继承了WeakReference（弱引用），再来看一下key怎么赋值的：

```

1  public WeakReference(T referent) {
2      super(referent);
3  }

```

key的赋值，使用的是WeakReference的赋值。



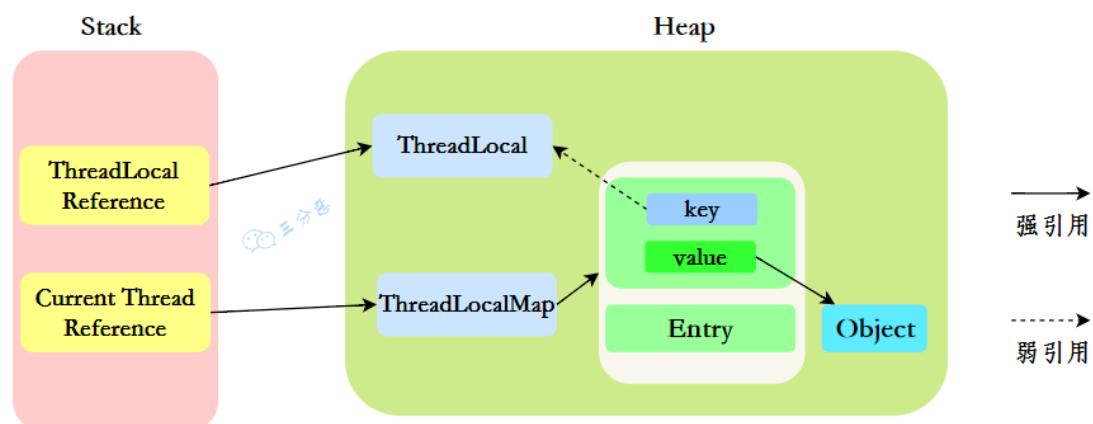
所以，怎么回答ThreadLocal原理？要答出这几个点：

- Thread类有一个类型为ThreadLocal.ThreadLocalMap的实例变量threadLocals，每个线程都有一个属于自己的ThreadLocalMap。
- ThreadLocalMap内部维护着Entry数组，每个Entry代表一个完整的对象，key是ThreadLocal的弱引用，value是ThreadLocal的泛型值。
- 每个线程在往ThreadLocal里设置值的时候，都是往自己的ThreadLocalMap里存，读也是以某个ThreadLocal作为引用，在自己的map里找对应的key，从而实现了线程隔离。
- ThreadLocal本身不存储值，它只是作为一个key来让线程往ThreadLocalMap里存取值。

13. ThreadLocal 内存泄露是怎么回事？

我们先来分析一下使用ThreadLocal时的内存，我们都知道，在JVM中，栈内存线程私有，存储了对象的引用，堆内存线程共享，存储了对象实例。

所以呢，栈中存储了ThreadLocal、Thread的引用，堆中存储了它们的具体实例。



ThreadLocalMap中使用的key为ThreadLocal的弱引用。

“弱引用：只要垃圾回收机制一运行，不管JVM的内存空间是否充足，都会回收该对象占用的内存。”

那么现在问题就来了，弱引用很容易被回收，如果ThreadLocal（ThreadLocalMap的Key）被垃圾回收器回收了，但是ThreadLocalMap生命周期和Thread是一样的，它这时候如果不被回收，就会出现这种情况：ThreadLocalMap的key没了，value还在，这就会造成了内存泄漏问题。

那怎么解决内存泄漏问题呢？

很简单，使用完ThreadLocal后，及时调用remove()方法释放内存空间。

```
1 ThreadLocal<String> localVariable = new ThreadLocal();
2 try {
3     localVariable.set("鄙人三某");
4     .....
5 } finally {
6     localVariable.remove();
7 }
```

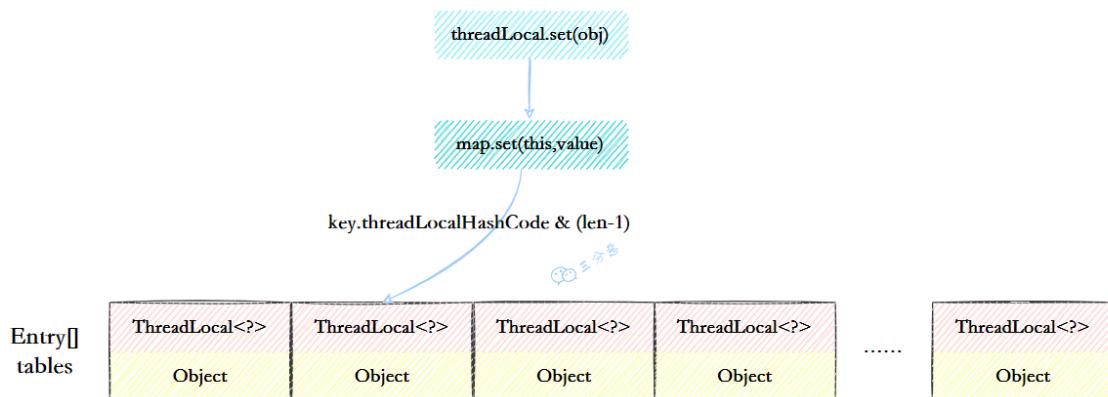
那为什么key还要设计成弱引用？

key设计成弱引用同样是为了防止内存泄漏。

假如key被设计成强引用，如果ThreadLocal Reference被销毁，此时它指向ThreadLocal的强引用就没有了，但是此时key还强引用指向ThreadLocal，就会导致ThreadLocal不能被回收，这时候就发生了内存泄漏的问题。

14. ThreadLocalMap的结构了解吗？

ThreadLocalMap虽然被叫做Map，其实它是没有实现Map接口的，但是结构还是和HashMap比较类似的，主要关注的是两个要素：**元素数组** 和 **散列方法**。



- 元素数组

一个table数组，存储Entry类型的元素，Entry是ThreaLocal弱引用作为key，Object作为value的结构。

```
1 | private Entry[] table;
```

- 散列方法

散列方法就是怎么把对应的key映射到table数组的相应下标，ThreadLocalMap用的是哈希取余法，取出key的threadLocalHashCode，然后和table数组长度减一&运算（相当于取余）。

```
1 | int i = key.threadLocalHashCode & (table.length - 1);
```

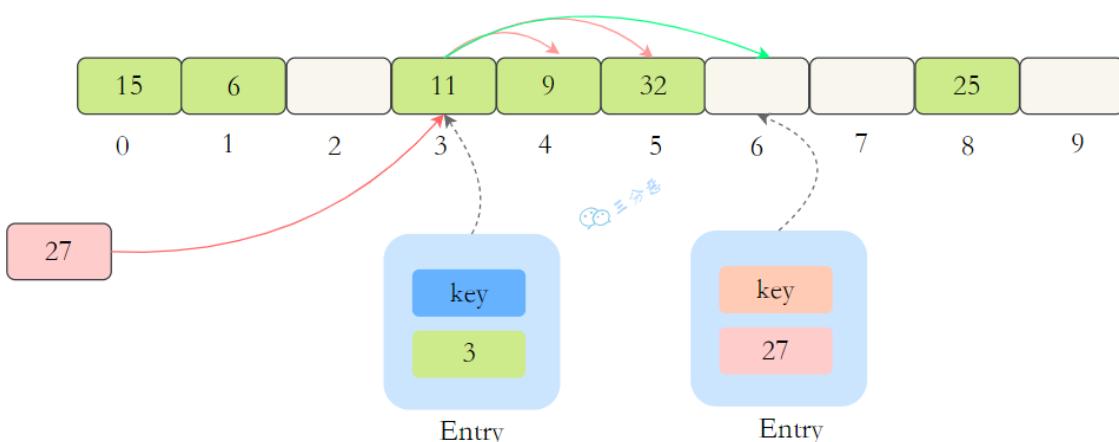
这里的threadLocalHashCode计算有点东西，每创建一个ThreadLocal对象，它就会新增 **0x61c88647**，这个值很特殊，它是**斐波那契数**也叫**黄金分割数**。**hash** 增量为这个数字，带来的好处就是 **hash** 分布非常均匀。

```
1 | private static final int HASH_INCREMENT = 0x61c88647;
2 |
3 | private static int nextHashCode() {
4 |     return nextHashCode.getAndAdd(HASH_INCREMENT);
5 | }
```

15.ThreadLocalMap怎么解决Hash冲突的？

我们可能都知道HashMap使用了链表来解决冲突，也就是所谓的链地址法。

ThreadLocalMap没有使用链表，自然也不是用链地址法来解决冲突了，它用的是另外一种方式——**开放定址法**。开放定址法是什么意思呢？简单来说，就是这个坑被人占了，那就接着去找空着的坑。



如上图所示，如果我们插入一个value=27的数据，通过 hash计算后应该落入第 4 个槽位中，而槽位 4 已经有了 Entry数据，而且Entry数据的key和当前不相等。此时就会线性向后查找，一直找到 Entry为 null的槽位才会停止查找，把元素放到空的槽中。

在get的时候，也会根据ThreadLocal对象的hash值，定位到table中的位置，然后判断该槽位Entry对象中的key是否和get的key一致，如果不一致，就判断下一个位置。

16.ThreadLocalMap扩容机制了解吗？

在ThreadLocalMap.set()方法的最后，如果执行完启发式清理工作后，未清理到任何数据，且当前散列数组中 Entry 的数量已经达到了列表的扩容阈值 ($\text{len} \times 2/3$)，就开始执行 rehash() 逻辑：

```
1 | if (!cleanSomeSlots(i, sz) && sz >= threshold)
2 |     rehash();
```

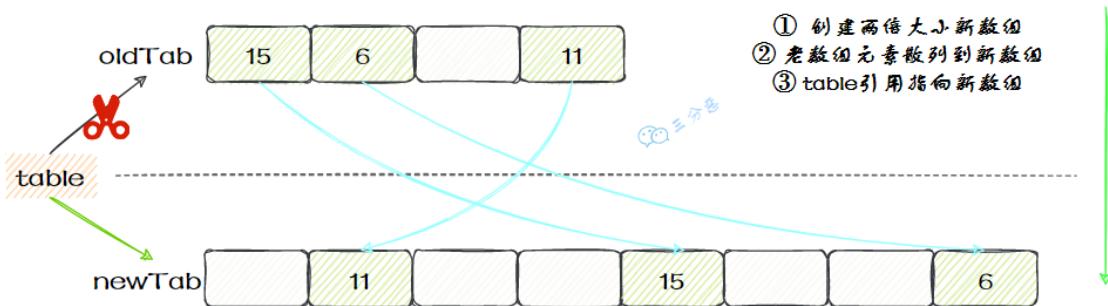
再看rehash()具体实现：这里会先去清理过期的Entry，然后还要根据条件判断 $\text{size} >= \text{threshold} - \text{threshold} / 4$ 也就是 $\text{size} >= \text{threshold} \times 3/4$ 来决定是否需要扩容。

```
1 | private void rehash() {
2 |     //清理过期Entry
3 |     expungeStaleEntries();
4 |
5 |     //扩容
6 |     if (size >= threshold - threshold / 4)
7 |         resize();
8 |
9 |
10 |    //清理过期Entry
11 |    private void expungeStaleEntries() {
12 |        Entry[] tab = table;
13 |        int len = tab.length;
14 |        for (int j = 0; j < len; j++) {
15 |            Entry e = tab[j];
16 |            if (e != null && e.get() == null)
17 |                expungeStaleEntry(j);
18 |        }
19 |    }
```

19 }

20

接着看看具体的 `resize()` 方法，扩容后的 `newTab` 的大小为老数组的两倍，然后遍历老的 `table` 数组，散列方法重新计算位置，开放地址解决冲突，然后放到新的 `newTab`，遍历完成之后，`oldTab` 中所有的 `entry` 数据都已经放入到 `newTab` 中了，然后 `table` 引用指向 `newTab`



具体代码：

```
private void resize() {
    Entry[] oldTab = table;
    int oldLen = oldTab.length;
    int newLen = oldLen * 2;
    Entry[] newTab = new Entry[newLen];
    int count = 0;

    for (int j = 0; j < oldLen; ++j) {
        Entry e = oldTab[j];
        if (e != null) {
            ThreadLocal<?> k = e.get();
            if (k == null) {
                e.value = null; // Help the GC
            } else {
                int h = k.threadLocalHashCode & (newLen - 1);
                while (newTab[h] != null)
                    h = nextIndex(h, newLen);
                newTab[h] = e;
                count++;
            }
        }
    }

    setThreshold(newLen);
    size = count;
    table = newTab;
}
```

17.父子线程怎么共享数据？

父线程能用ThreadLocal来给子线程传值吗？毫无疑问，不能。那该怎么办？

这时候可以用到另外一个类——**InheritableThreadLocal**。

使用起来很简单，在主线程的InheritableThreadLocal实例设置值，在子线程中就可以拿到了。

```
1 public class InheritableThreadLocalTest {  
2  
3     public static void main(String[] args) {  
4         final ThreadLocal threadLocal = new  
5             InheritableThreadLocal();  
6             // 主线程  
7             threadLocal.set("不擅技术");  
8             //子线程  
9             Thread t = new Thread() {  
10                 @Override  
11                 public void run() {  
12                     super.run();  
13                     System.out.println("鄙人三某 , " +  
14                         threadLocal.get());  
15                 }  
16             };  
17         t.start();  
18     }  
19 }
```

那原理是什么呢？

原理很简单，在Thread类里还有另外一个变量：

```
1 | ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
```

在Thread.init的时候，如果父线程的**inheritableThreadLocals**不为空，就把它赋给当前线程（子线程）的**inheritableThreadLocals**。

```
1     if (inheritThreadLocals &&
2         parent.inheritableThreadLocals != null)
3             this.inheritableThreadLocals =
4
5     ThreadLocal.createInheritedMap(parent.inheritableThreadLocals
6 );
```

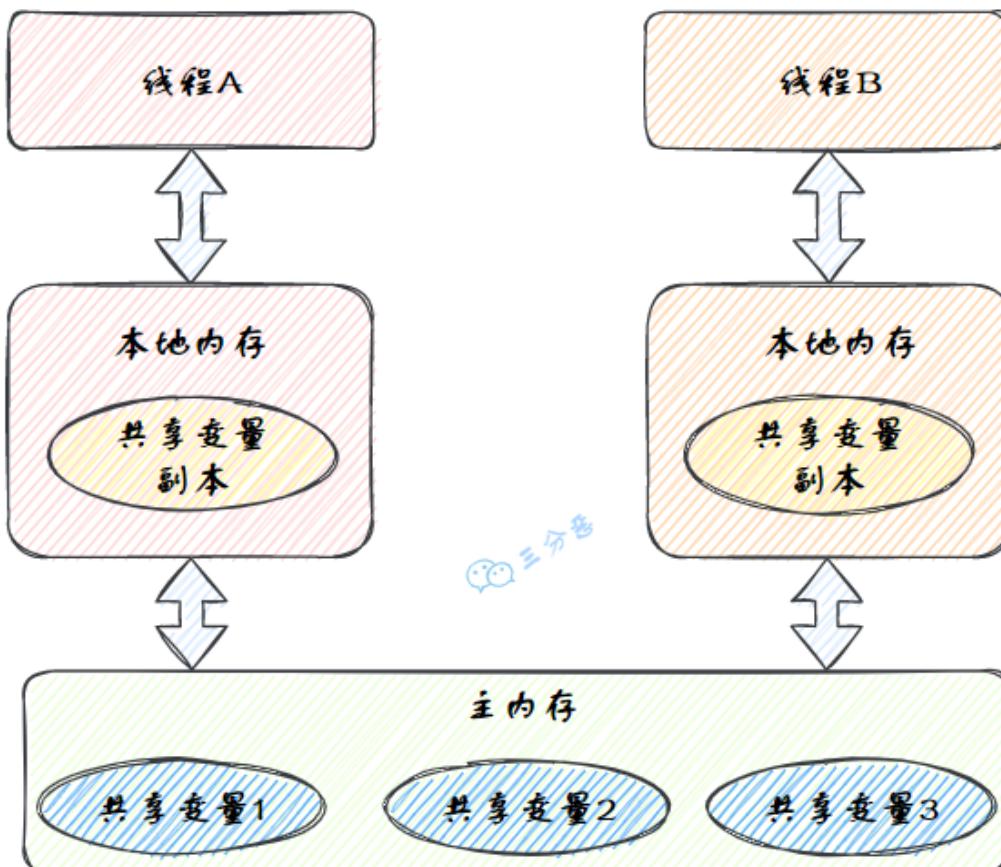
Java内存模型

18.说一下你对Java内存模型（JMM）的理解？

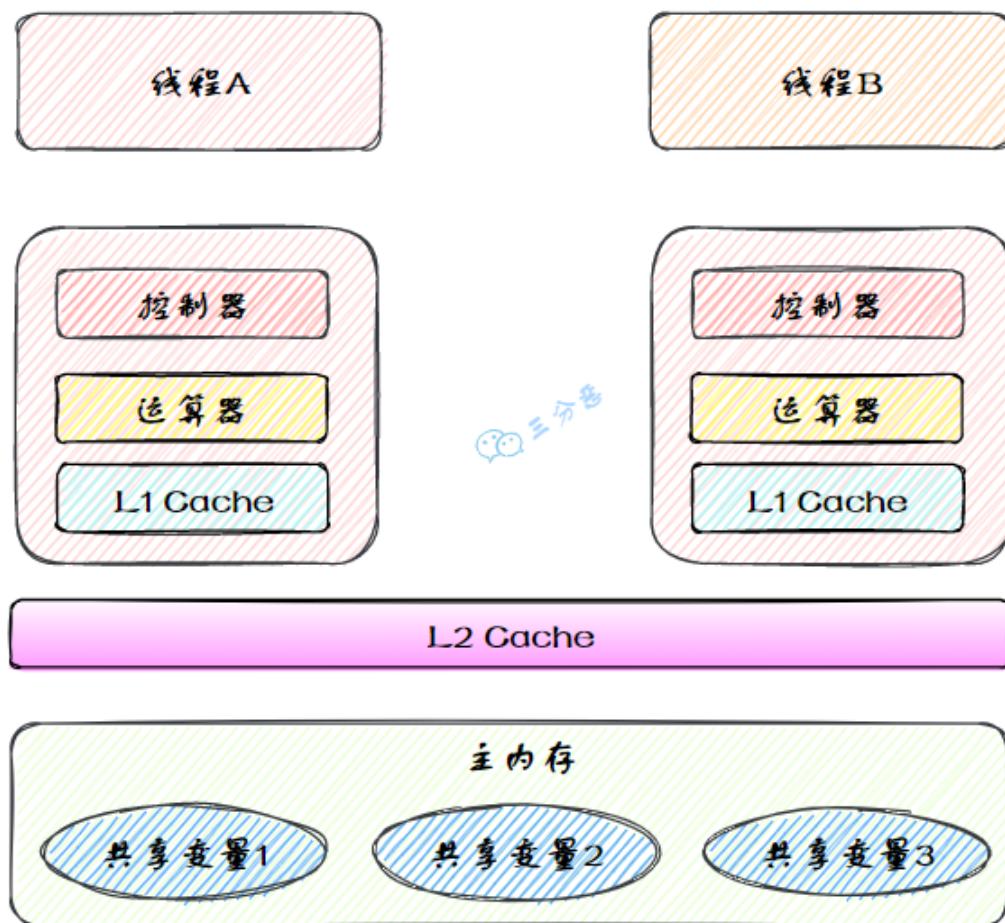
Java内存模型（Java Memory Model, JMM），是一种抽象的模型，被定义出来屏蔽各种硬件和操作系统的内存访问差异。

JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在 **主内存**（Main Memory）中，每个线程都有一个私有的 **本地内存**（Local Memory），本地内存中存储了该线程以读/写共享变量的副本。

Java内存模型的抽象图：



本地内存是JMM的一个抽象概念，并不真实存在。它其实涵盖了缓存、写缓冲区、寄存器以及其他硬件和编译器优化。



图里面的是一个双核 CPU 系统架构，每个核有自己的控制器和运算器，其中控制器包含一组寄存器和操作控制器，运算器执行算术逻辑运算。每个核都有自己的一级缓存，在有些架构里面还有一个所有 CPU 共享的二级缓存。那么 Java 内存模型里面的工作内存，就对应这里的 L1 缓存或者 L2 缓存或者 CPU 寄存器。

19. 说说你对原子性、可见性、有序性的理解？

原子性、有序性、可见性是并发编程中非常重要的基础概念，JMM的很多技术都是围绕着这三大特性展开。

- 原子性：原子性指的是一个操作是不可分割、不可中断的，要么全部执行并且执行的过程不会被任何因素打断，要么就全不执行。
- 可见性：可见性指的是一个线程修改了某一个共享变量的值时，其它线程能够立即知道这个修改。

- 有序性：有序性指的是对于一个线程的执行代码，从前往后依次执行，单线程下可以认为程序是有序的，但是并发时有可能会发生指令重排。

分析下面几行代码的原子性？

```
1 | int i = 2;
2 | int j = i;
3 | i++;
4 | i = i + 1;
```

- 第1句是基本类型赋值，是原子性操作。
- 第2句先读i的值，再赋值到j，两步操作，不能保证原子性。
- 第3和第4句其实是等效的，先读取i的值，再+1，最后赋值到i，三步操作了，不能保证原子性。

原子性、可见性、有序性都应该怎么保证呢？

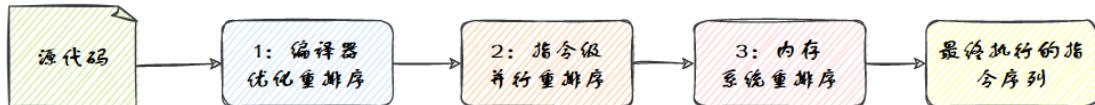
- 原子性：JMM只能保证基本的原子性，如果要保证一个代码块的原子性，需要使用 `synchronized`。
- 可见性：Java是利用 `volatile` 关键字来保证可见性的，除此之外，`final` 和 `synchronized` 也能保证可见性。
- 有序性：`synchronized` 或者 `volatile` 都可以保证多线程之间操作的有序性。

20.那说说什么是指令重排？

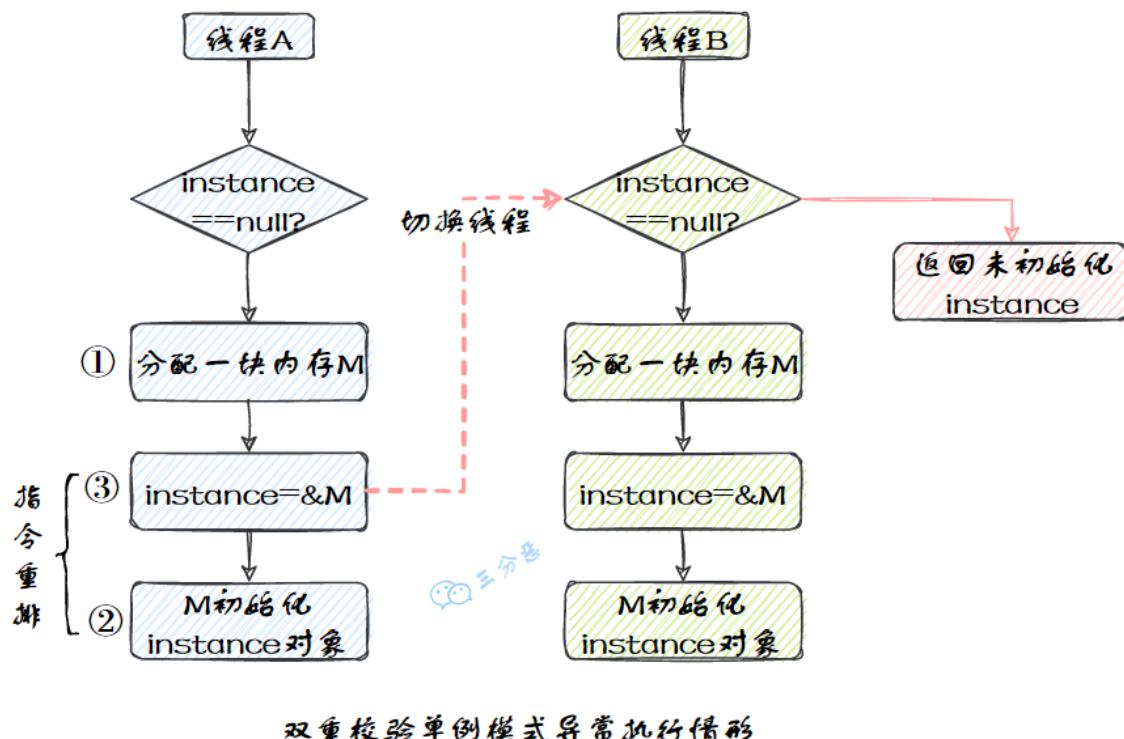
在执行程序时，为了提高性能，编译器和处理器常常会对指令做重排序。重排序分3种类型。

- 编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。
- 指令级并行的重排序。现代处理器采用了指令级并行技术（Instruction-Level Parallelism, ILP）来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。
- 内存系统的重排序。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

从Java源代码到最终实际执行的指令序列，会分别经历下面3种重排序，如图：



我们比较熟悉的双重校验单例模式就是一个经典的指令重排的例子，`Singleton instance=new Singleton();` 对应的JVM指令分为三步：分配内存空间-->初始化对象-->对象指向分配的内存空间，但是经过了编译器的指令重排序，第二步和第三步就可能会重排序。



JMM属于语言级的内存模型，它确保在不同的编译器和不同的处理器平台之上，通过禁止特定类型的编译器重排序和处理器重排序，为程序员提供一致的内存可见性保证。

21. 指令重排有限制吗？happens-before了解吗？

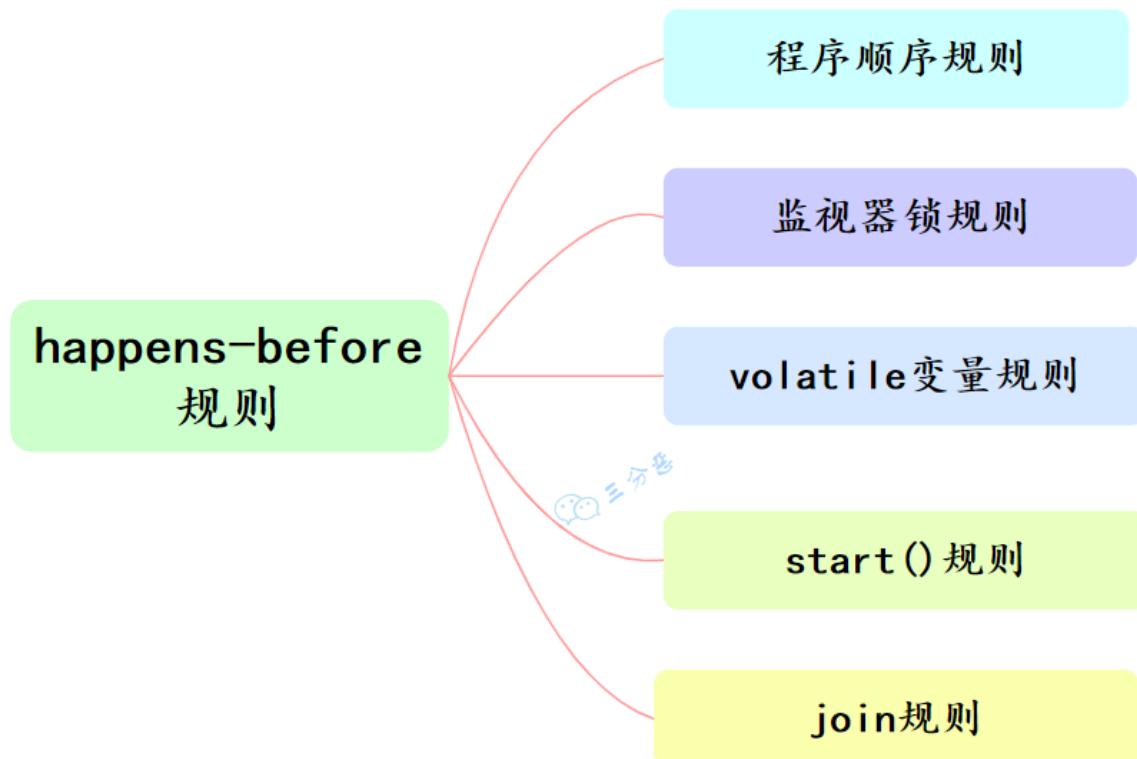
指令重排也是有一些限制的，有两个规则 `happens-before` 和 `as-if-serial` 来约束。

`happens-before` 的定义：

- 如果一个操作 `happens-before` 另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。

- 两个操作之间存在happens-before关系，并不意味着Java平台的具体实现必须要按照 happens-before关系指定的顺序来执行。如果重排序之后的执行结果，与按 happens-before关系来执行的结果一致，那么这种重排序并不非法

happens-before和我们息息相关的有六大规则：



- 程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作。
- 监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁。
- volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个 volatile域的读。
- 传递性：如果A happens-before B，且B happens-before C，那么A happens-before C。
- start()规则：如果线程A执行操作ThreadB.start()（启动线程B），那么A线程的 ThreadB.start()操作happens-before于线程B中的任意操作。
- join()规则：如果线程A执行操作ThreadB.join()并成功返回，那么线程B中的任意操作 happens-before于线程A从ThreadB.join()操作成功返回。

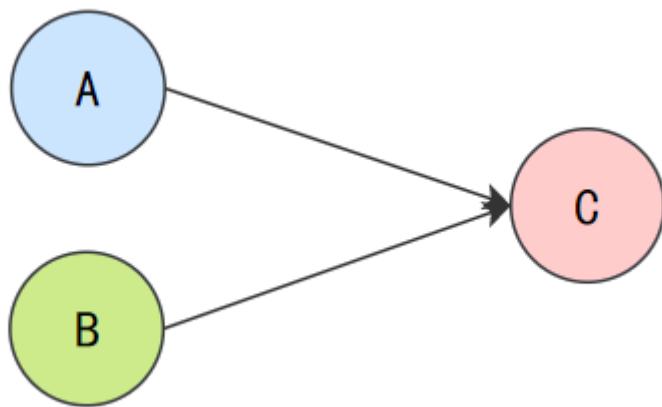
22.as-if-serial又是什么？单线程的程序一定是顺序的吗？

as-if-serial语义的意思是：不管怎么重排序（编译器和处理器为了提高并行度），**单线程程序的执行结果不能被改变**。编译器、runtime和处理器都必须遵守as-if-serial语义。

为了遵守as-if-serial语义，编译器和处理器不会对存在数据依赖关系的操作做重排序，因为这种重排序会改变执行结果。但是，如果操作之间不存在数据依赖关系，这些操作就可能被编译器和处理器重排序。为了具体说明，请看下面计算圆面积的代码示例。

```
1 | double pi = 3.14;    // A
2 | double r = 1.0;      // B
3 | double area = pi * r * r; // C
```

上面3个操作的数据依赖关系：



A和C之间存在数据依赖关系，同时B和C之间也存在数据依赖关系。因此在最终执行的指令序列中，C不能被重排序到A和B的前面（C排到A和B的前面，程序的结果将会被改变）。但A和B之间没有数据依赖关系，编译器和处理器可以重排序A和B之间的执行顺序。

所以最终，程序可能会有两种执行顺序：



as-if-serial语义把单线程程序保护了起来，遵守as-if-serial语义的编译器、runtime和处理器共同编织了这么一个“楚门的世界”：单线程程序是按程序的“顺序”来执行的。as-if-serial语义使单线程情况下，我们不需要担心重排序的问题，可见性的问题。

23.volatile实现原理了解吗？

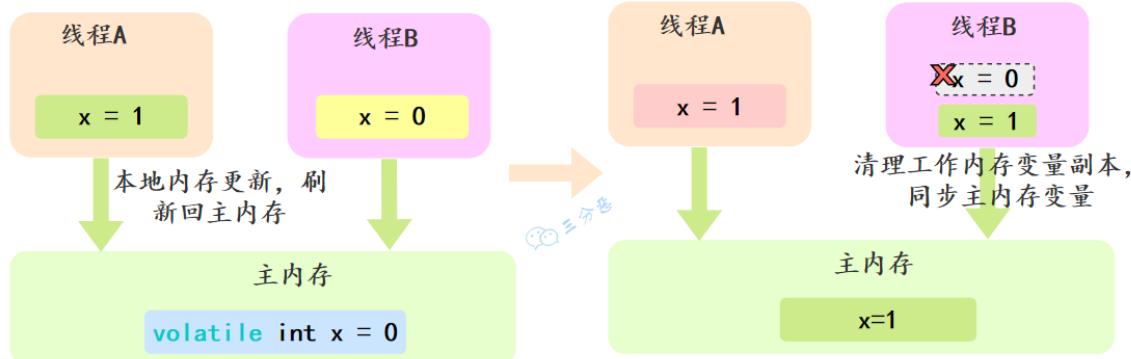
volatile有两个作用，保证可见性和有序性。

volatile怎么保证可见性的呢？

相比synchronized的加锁方式来解决共享变量的内存可见性问题，volatile就是更轻量的选择，它没有上下文切换的额外开销成本。

volatile可以确保对某个变量的更新对其他线程马上可见，一个变量被声明为volatile时，线程在写入变量时不会把值缓存在寄存器或者其他地方，而是会把值刷新回主内存 当其它线程读取该共享变量，会从主内存重新获取最新值，而不是使用当前线程的本地内存中的值。

例如，我们声明一个 volatile 变量 volatile int x = 0，线程A修改x=1，修改完之后就会把新的值刷新回主内存，线程B读取x的时候，就会清空本地内存变量，然后再从主内存获取最新值。



volatile怎么保证有序性的呢？

重排序可以分为编译器重排序和处理器重排序，volatile保证有序性，就是通过分别限制这两种类型的重排序。

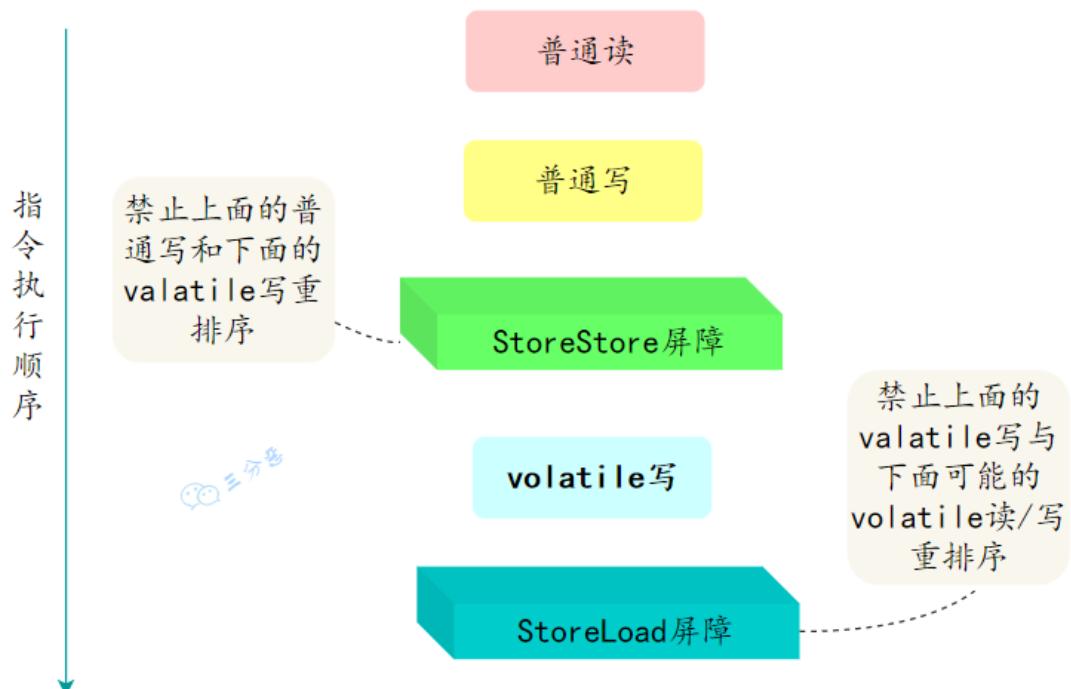
volatile重排序规则表

是否能重排序	第二个操作		
	第一个操作	普通读/写	volatile读
普通读/写	✓	✓ <small>三分多</small>	✗
volatile读	✗	✗	✗
volatile写	✓	✗	✗

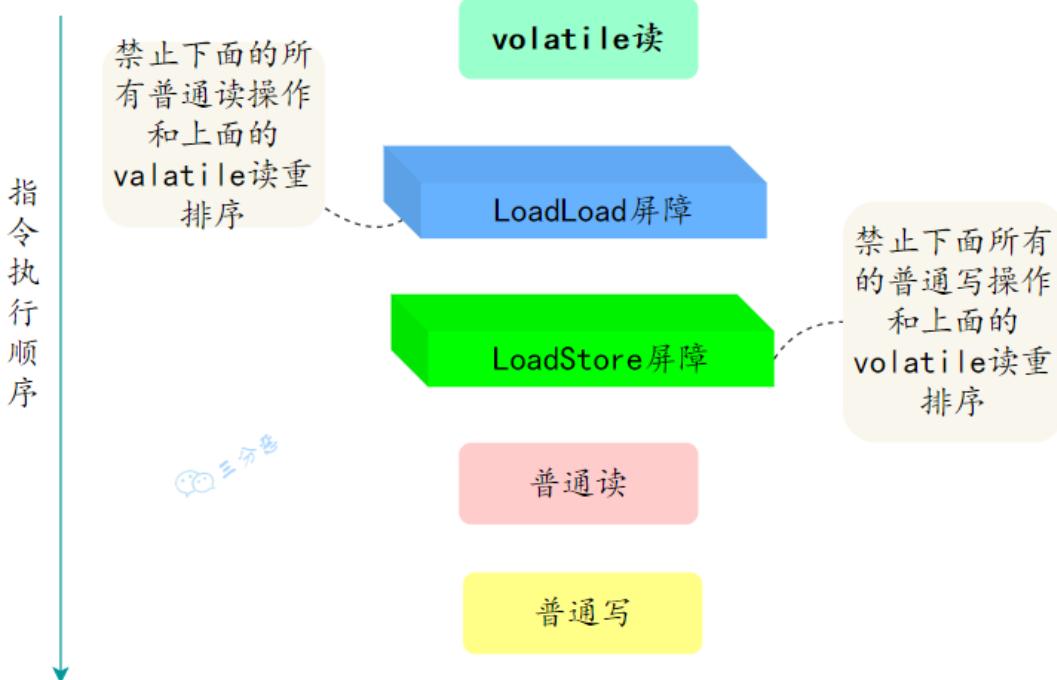
为了实现volatile的内存语义，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。

1. 在每个volatile写操作的前面插入一个 `StoreStore` 屏障
2. 在每个volatile写操作的后面插入一个 `StoreLoad` 屏障
3. 在每个volatile读操作的后面插入一个 `LoadLoad` 屏障
4. 在每个volatile读操作的后面插入一个 `LoadStore` 屏障

volatile写插入内存屏障后生成的指令序列示意图



volatile写插入内存屏障后生成的指令序列示意图



锁

24.synchronized用过吗？怎么使用？

synchronized经常用的，用来保证代码的原子性。

synchronized主要有三种用法：

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

```
1 synchronized void method() {  
2     //业务代码  
3 }
```

- 修饰静态方法：也就是给当前类加锁，会作用于类的所有对象实例，进入同步代码前要获得当前 class 的锁。因为静态成员不属于任何一个实例对象，是类成员（static 表明这是该类的一个静态资源，不管 new 了多少个对象，只有一份）。

如果一个线程 A 调用一个实例对象的非静态 synchronized 方法，而线程 B 需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁。

```
1 | synchronized void static method() {  
2 |     //业务代码  
3 | }
```

- 修饰代码块：指定加锁对象，对给定对象/类加锁。synchronized(this|object) 表示进入同步代码库前要获得给定对象的锁。synchronized(类.class) 表示进入同步代码前要获得当前 class 的锁

```
1 | synchronized(this) {  
2 |     //业务代码  
3 | }
```

25.synchronized的实现原理？

synchronized是怎么加锁的呢？

我们使用synchronized的时候，发现不用自己去lock和unlock，是因为JVM帮我们把这个事情做了。

1. synchronized修饰代码块时，JVM采用 `monitorenter`、`monitorexit` 两个指令来实现同步，`monitorenter` 指令指向同步代码块的开始位置，`monitorexit` 指令则指向同步代码块的结束位置。

反编译一段synchronized修饰代码块代码，`javap -c -s -v -l SynchronizedDemo.class`，可以看到相应的字节码指令。

```
public void method();
descriptor: ()V
flags: ACC_PUBLIC
Code:
stack=2, locals=3, args_size=1
  0: aload_0
  1: dup
  2: astore_1
  3: monitorenter
  4: getstatic      #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
  7: ldc           #3                  // String synchronized demo
  9: invokevirtual #4                  // Method java/io/PrintStream println:(Ljava/lang/String;)V
 12: aload_1
 13: monitorexit
 14: goto          22
 17: astore_2
 18: aload_1
 19: monitorexit
 20: aload_2
 21: athrow
 22: return
```

2. synchronized修饰同步方法时，JVM采用 **ACC_SYNCHRONIZED** 标记符来实现同步，这个标识指明了该方法是一个同步方法。

同样可以写段代码反编译看一下。

```
public synchronized void method();
descriptor: ()V
flags: ACC_PUBLIC, ACC_SYNCHRONIZED
Code:
stack=2, locals=1, args_size=1
  0: getstatic      #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
  3: ldc           #3                  // String synchronized method
  5: invokevirtual #4                  // Method java/io/PrintStream println:(Ljava/lang/String;)V
  8: return
LineNumberTable:
  line 10: 0
  line 11: 8
}
```

synchronized锁住的是什么呢？

monitorenter、monitorexit或者ACC_SYNCHRONIZED都是**基于Monitor实现的**。

实例对象结构里有对象头，对象头里面有一块结构叫Mark Word，Mark Word指针指向了**monitor**。

所谓的Monitor其实是一种**同步工具**，也可以说是一种**同步机制**。在Java虚拟机(HotSpot)中，Monitor是由**ObjectMonitor**实现的，可以叫做内部锁，或者Monitor锁。

ObjectMonitor的工作原理：

- ObjectMonitor有两个队列：WaitSet、EntryList，用来保存ObjectWaiter对象列表。
- _owner，获取Monitor对象的线程进入_owner区时，_count + 1。如果线程调用了wait()方法，此时会释放Monitor对象，_owner恢复为空，_count - 1。同时该等待线程进入_WaitSet中，等待被唤醒。

```
1 | ObjectMonitor() {
2 |     _header      = NULL;
3 |     _count       = 0; // 记录线程获取锁的次数
4 |     _waiters     = 0,
5 |     _recursions  = 0; // 锁的重入次数
6 |     _object      = NULL;
7 |     _owner       = NULL; // 指向持有ObjectMonitor对象的线程
8 |     _WaitSet     = NULL; // 处于wait状态的线程，会被加入到
|     _WaitSet
9 |     _WaitSetLock = 0 ;
10 |    _Responsible = NULL ;
11 |    _succ        = NULL ;
12 |    _cxq         = NULL ;
13 |    FreeNext     = NULL ;
14 |    _EntryList   = NULL ; // 处于等待锁block状态的线程，会被加入
| 到该列表
15 |    _SpinFreq    = 0 ;
16 |    _SpinClock   = 0 ;
17 |    OwnerIsThread = 0 ;
18 }
```

可以类比一个去医院就诊的例子[18]：

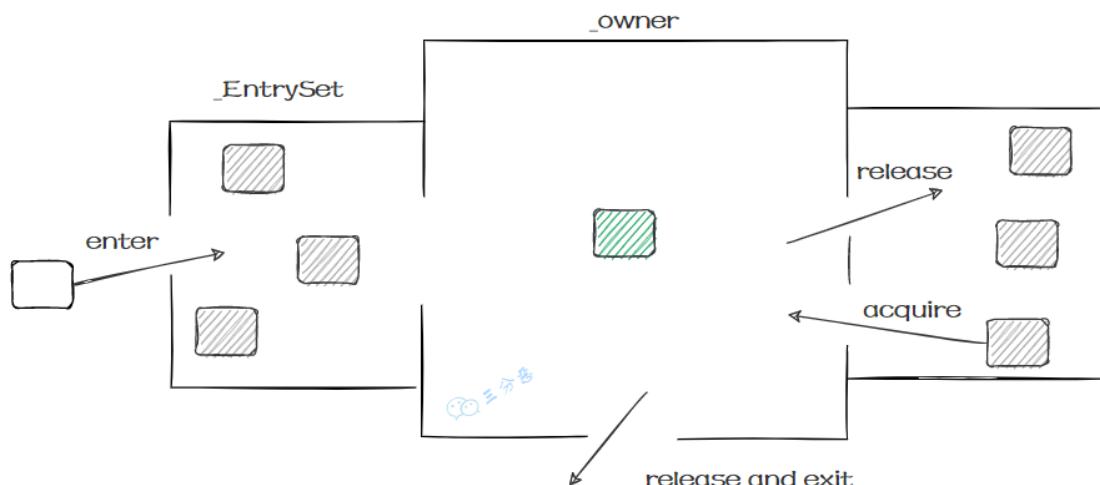
- 首先，患者在**门诊大厅**前台或自助挂号机**进行挂号**；
- 随后，挂号结束后患者找到对应的**诊室就诊**：
 - 诊室每次只能有一个患者就诊；
 - 如果此时诊室空闲，直接进入就诊；
 - 如果此时诊室内有其它患者就诊，那么当前患者进入**候诊室**，等待叫号；

- 就诊结束后，走出就诊室，候诊室的**下一位候诊患者**进入就诊室。



这个过程就和Monitor机制比较相似：

- 门诊大厅：所有待进入的线程都必须先在入口Entry Set 挂号才有资格；
- 就诊室：就诊室 _Owner 里只能有一个线程就诊，就诊完线程就自行离开
- 候诊室：就诊室繁忙时，进入 等待区（Wait Set），就诊室空闲的时候就从 等待区（Wait Set）叫新的线程



所以我们就知道了，同步是锁住的什么东西：

- monitorenter，在判断拥有同步标识 ACC_SYNCHRONIZED 抢先进入此方法的线程会优先拥有 Monitor 的 owner，此时计数器 +1。
- monitorexit，当执行完退出后，计数器 -1，归 0 后被其他进入的线程获得。

26.除了原子性，synchronized可见性，有序性，可重入性怎么实现？

synchronized怎么保证可见性？

- 线程加锁前，将清空工作内存中共享变量的值，从而使用共享变量时需要从主内存中重新读取最新的值。
- 线程加锁后，其它线程无法获取主内存中的共享变量。
- 线程解锁前，必须把共享变量的最新值刷新到主内存中。

| synchronized怎么保证有序性？

synchronized同步的代码块，具有排他性，一次只能被一个线程拥有，所以synchronized保证同一时刻，代码是单线程执行的。

因为as-if-serial语义的存在，单线程的程序能保证最终结果是有序的，但是不保证不会指令重排。

所以synchronized保证的有序是执行结果的有序性，而不是防止指令重排的有序性。

| synchronized怎么实现可重入的呢？

synchronized 是可重入锁，也就是说，允许一个线程二次请求自己持有对象锁的临界资源，这种情况称为可重入锁。

synchronized 锁对象的时候有个计数器，他会记录下线程获取锁的次数，在执行完对应的代码块之后，计数器就会-1，直到计数器清零，就释放锁了。

之所以，是可重入的。是因为 synchronized 锁对象有个计数器，会随着线程获取锁后 +1 计数，当线程执行完毕后 -1，直到清零释放锁。

27. 锁升级？synchronized优化了解吗？

了解锁升级，得先知道，不同锁的状态是什么样的。这个状态指的是什么呢？

Java对象头里，有一块结构，叫 **Mark Word** 标记字段，这块结构会随着锁的状态变化而变化。

64 位虚拟机 Mark Word 是 64bit，我们来看看它的状态变化：

Mark Word变化示意图

锁状态	61 bit					偏向锁标记 1 bit	锁类型 2 bit
	未使用 25 bit	HashCode 32 bit	未使用 1bit	分代年龄 4bit			
无锁	未使用 25 bit	HashCode 32 bit	未使用 1bit	分代年龄 4bit	0	01	
偏向锁	当前线程指针 54 bit	Epoch 2 bit	未使用 1bit	分代年龄 4bit	1	01	
轻量锁	指向栈中锁记录的指针 62 bit					00	
重量级锁	指向重量级锁的指针 62 bit					10	

Mark Word存储对象自身的运行数据，如哈希码、GC分代年龄、锁状态标志、偏向时间戳（Epoch）等。

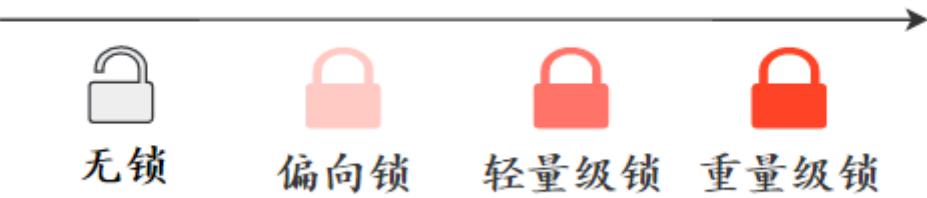
synchronized做了哪些优化？

在JDK1.6之前，synchronized的实现直接调用ObjectMonitor的enter和exit，这种锁被称之为重量级锁。从JDK6开始，HotSpot虚拟机开发团队对Java中的锁进行优化，如增加了适应性自旋、锁消除、锁粗化、轻量级锁和偏向锁等优化策略，提升了synchronized的性能。

- 偏向锁：在无竞争的情况下，只是在Mark Word里存储当前线程指针，CAS操作都不做。
- 轻量级锁：在没有多线程竞争时，相对重量级锁，减少操作系统互斥量带来的性能消耗。但是，如果存在锁竞争，除了互斥量本身开销，还额外有CAS操作的开销。
- 自旋锁：减少不必要的CPU上下文切换。在轻量级锁升级为重量级锁时，就使用了自旋加锁的方式
- 锁粗化：将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁。
- 锁消除：虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。

锁升级的过程是什么样的？

锁升级方向：无锁-->偏向锁---> 轻量级锁---->重量级锁，这个方向基本上是不可逆的。



我们看一下升级的过程：

H5 偏向锁：

偏向锁的获取：

1. 判断是否为可偏向状态--MarkWord中锁标志是否为‘01’，是否偏向锁是否为‘1’
2. 如果是可偏向状态，则查看线程ID是否为当前线程，如果是，则进入步骤‘5’，否则进入步骤‘3’
3. 通过CAS操作竞争锁，如果竞争成功，则将MarkWord中线程ID设置为当前线程ID，然后执行‘5’；竞争失败，则执行‘4’
4. CAS获取偏向锁失败表示有竞争。当达到safepoint时获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码块
5. 执行同步代码

偏向锁的撤销：

1. 偏向锁不会主动释放(撤销)，只有遇到其他线程竞争时才会执行撤销，由于撤销需要知道当前持有该偏向锁的线程栈状态，因此要等到safepoint时执行，此时持有该偏向锁的线程（T）有‘2’，‘3’两种情况；
2. 撤销----T线程已经退出同步代码块，或者已经不再存活，则直接撤销偏向锁，变成无锁状态----该状态达到阈值20则执行批量重偏向
3. 升级----T线程还在同步代码块中，则将T线程的偏向锁升级为轻量级锁，当前线程执行轻量级锁状态下的锁获取步骤----该状态达到阈值40则执行批量撤销

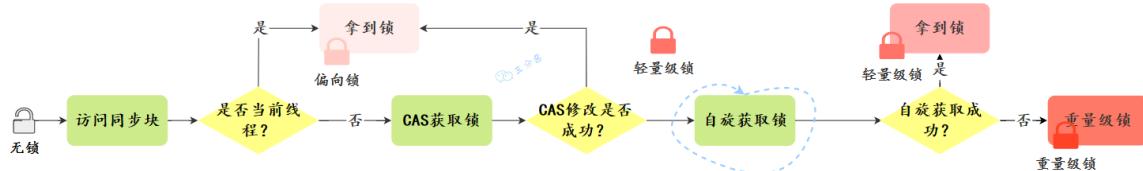
H5 轻量级锁：

轻量级锁的获取：

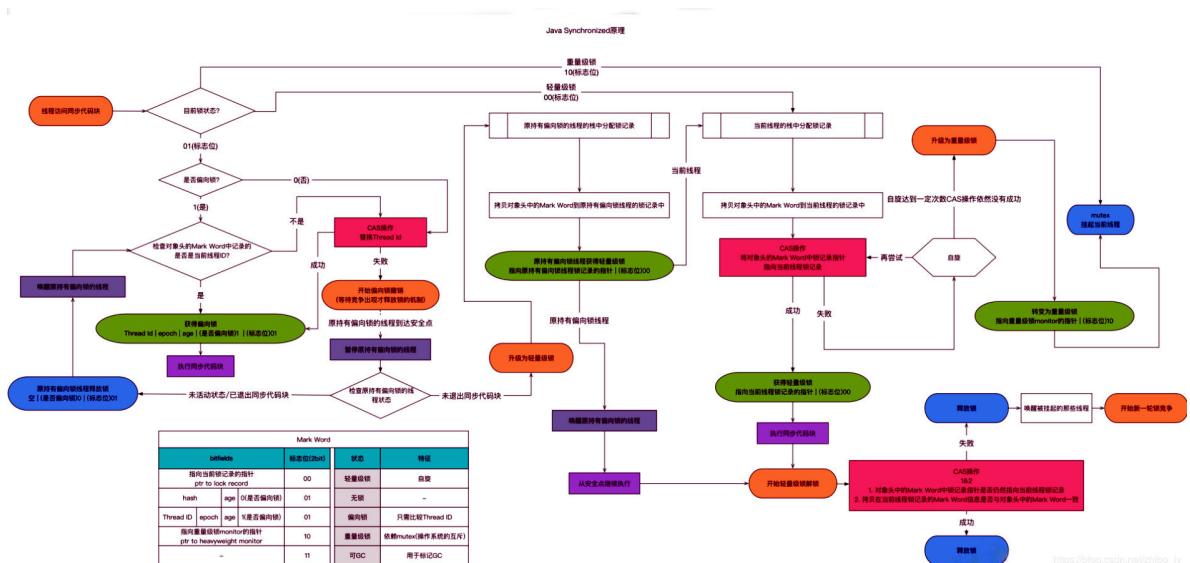
1. 进行加锁操作时，jvm会判断是否已经时重量级锁，如果不是，则会在当前线程栈帧中划出一块空间，作为该锁的锁记录，并且将锁对象MarkWord复制到该锁记录中
2. 复制成功之后，jvm使用CAS操作将对象头MarkWord更新为指向锁记录的指针，并将锁记录里的owner指针指向对象头的MarkWord。如果成功，则执行‘3’，否则执行‘4’

3. 更新成功，则当前线程持有该对象锁，并且对象MarkWord锁标志设置为‘00’，即表示此对象处于轻量级锁状态
4. 更新失败，jvm先检查对象MarkWord是否指向当前线程栈帧中的锁记录，如果是则执行‘5’，否则执行‘4’
5. 表示锁重入；然后当前线程栈帧中增加一个锁记录第一部分（Displaced Mark Word）为null，并指向Mark Word的锁对象，起到一个重入计数器的作用。
6. 表示该锁对象已经被其他线程抢占，则进行自旋等待（默认10次），等待次数达到阈值仍未获取到锁，则升级为重量级锁

大体上省简的升级过程：



完整的升级过程：



28. 说说synchronized和ReentrantLock的区别？

可以从锁的实现、功能特点、性能等几个维度去回答这个问题：

- **锁的实现：** synchronized是Java语言的关键字，基于JVM实现。而ReentrantLock是基于JDK的API层面实现的（一般是lock()和unlock()方法配合try/finally语句块来完成。）
- **性能：** 在JDK1.6锁优化以前，synchronized的性能比ReentrantLock差很多。但是JDK6开始，增加了适应性自旋、锁消除等，两者性能就差不多了。

- **功能特点：** ReentrantLock 比 synchronized 增加了一些高级功能，如等待可中断、可实现公平锁、可实现选择性通知。
 - ReentrantLock提供了一种能够中断等待锁的线程的机制，通过 lock.lockInterruptibly()来实现这个机制
 - ReentrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。
 - synchronized与wait()和notify()/notifyAll()方法结合实现等待/通知机制，ReentrantLock类借助Condition接口与newCondition()方法实现。
 - ReentrantLock需要手工声明来加锁和释放锁，一般跟finally配合释放锁。而synchronized不用手动释放锁。

下面的表格列出了两种锁之间的区别：

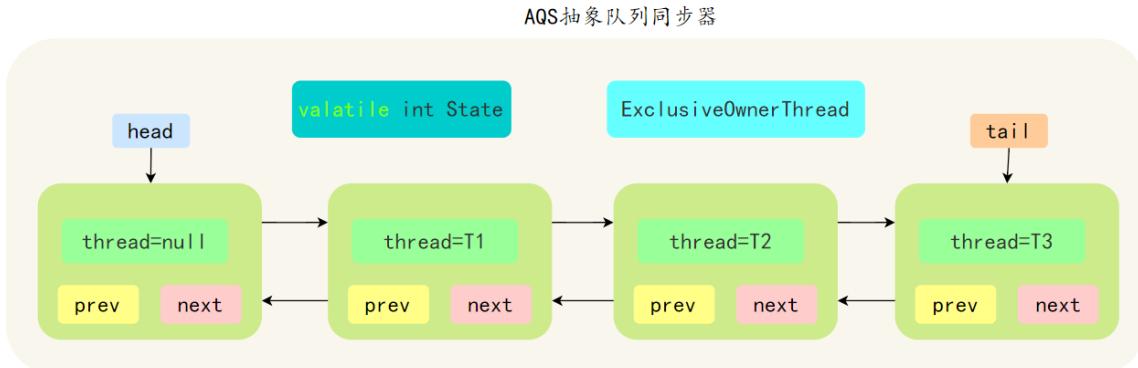
区别	synchronized	ReentrantLock
锁实现机制	对象头监视器模式	依赖AQS
灵活性	不灵活	支持响应中断、超时、尝试获取锁
释放锁形式	自动释放锁	显式调用unlock()
支持锁类型	非公平锁	公平锁&非公平锁
条件队列	单条件队列	多个条件队列
可重入支持	支持	支持

29.AQS了解多少？

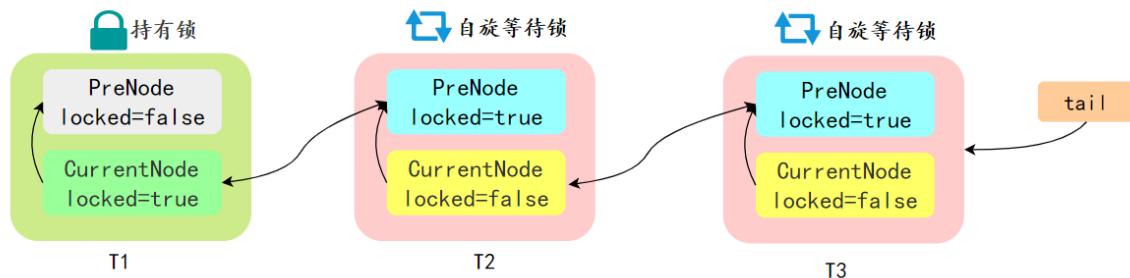
AbstractQueuedSynchronizer 抽象同步队列，简称 AQS，它是Java并发包的根基，并发包中的锁就是基于AQS实现的。

- AQS是基于一个FIFO的双向队列，其内部定义了一个节点类Node，Node 节点内部的 SHARED 用来标记该线程是获取共享资源时被阻挂起后放入AQS 队列的，EXCLUSIVE 用来标记线程是取独占资源时被挂起后放入AQS 队列
- AQS 使用一个 volatile 修饰的 int 类型的成员变量 state 来表示同步状态，修改同步状态成功即为获得锁，volatile 保证了变量在多线程之间的可见性，修改 State 值时通过 CAS 机制来保证修改的原子性

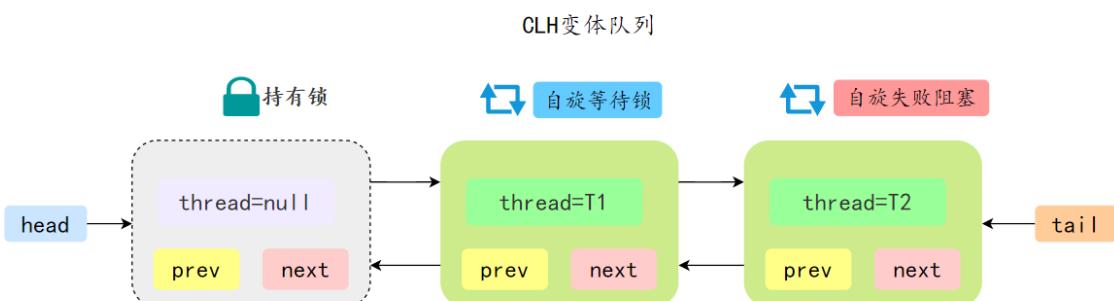
- 获取state的方式分为两种，独占方式和共享方式，一个线程使用独占方式获取了资源，其它线程就会在获取失败后被阻塞。一个线程使用共享方式获取了资源，另外一个线程还可以通过CAS的方式进行获取。
- 如果共享资源被占用，需要一定的阻塞等待唤醒机制来保证锁的分配，AQS中会将竞争共享资源失败的线程添加到一个变体的CLH队列中。



先简单了解一下CLH：Craig、Landin and Hagersten 队列，是 **单向链表实现的队列**。申请线程只在本地变量上自旋，**它不断轮询前驱的状态**，如果发现前驱节点释放了锁就结束自旋



AQS 中的队列是 CLH 变体的虚拟双向队列，通过将每条请求共享资源的线程封装成一个节点来实现锁的分配：



AQS 中的 CLH 变体等待队列拥有以下特性：

- AQS 中队列是个双向链表，也是 FIFO 先进先出的特性
- 通过 Head、Tail 头尾两个节点来组成队列结构，通过 volatile 修饰保证可见性
- Head 指向节点为已获得锁的节点，是一个虚拟节点，节点本身不持有具体线程

- 获取不到同步状态，会将节点进行自旋获取锁，自旋一定次数失败后会将线程阻塞，相对于 CLH 队列性能较好

ps:AQS源码里面有很多细节可问，建议有时间好好看看AQS源码。

30. ReentrantLock 实现原理？

ReentrantLock 是可重入的独占锁，只能有一个线程可以获取该锁，其它获取该锁的线程会被阻塞而被放入该锁的阻塞队列里面。

看看ReentrantLock的加锁操作：

```
1 // 创建非公平锁
2 ReentrantLock lock = new ReentrantLock();
3 // 获取锁操作
4 lock.lock();
5 try {
6     // 执行代码逻辑
7 } catch (Exception ex) {
8     // ...
9 } finally {
10    // 解锁操作
11    lock.unlock();
12 }
```

`new ReentrantLock()` 构造函数默认创建的是非公平锁 NonfairSync。

公平锁 FairSync

1. 公平锁是指多个线程按照申请锁的顺序来获取锁，线程直接进入队列中排队，队列中的第一个线程才能获得锁
2. 公平锁的优点是等待锁的线程不会饿死。缺点是整体吞吐效率相对非公平锁要低，等待队列中除第一个线程以外的所有线程都会阻塞，CPU 唤醒阻塞线程的开销比非公平锁大

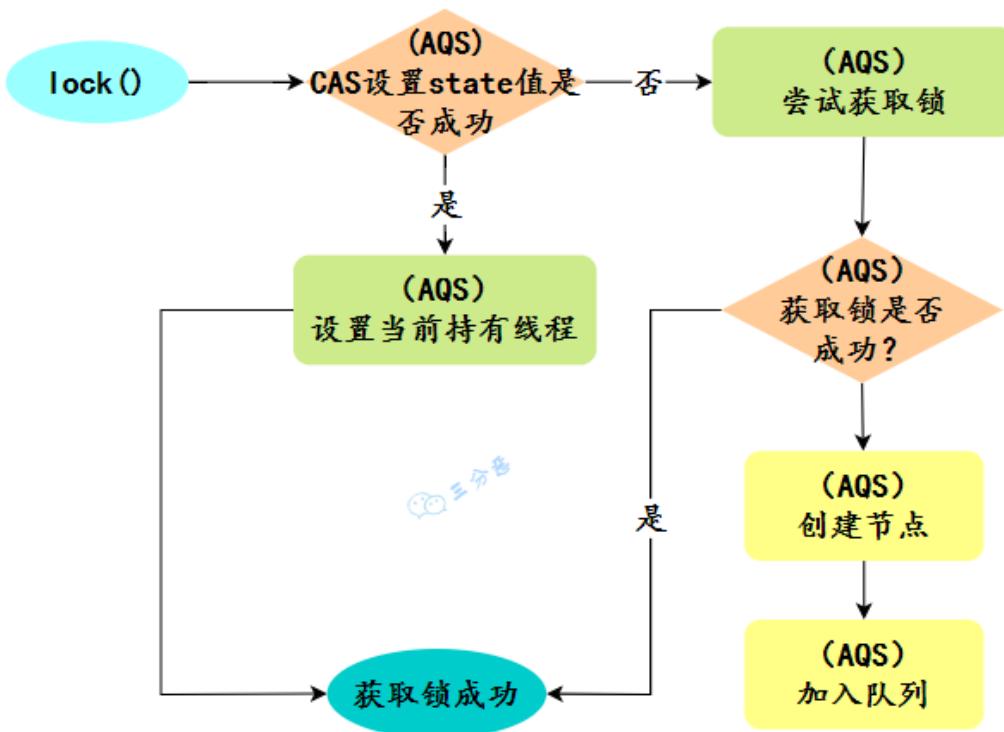
非公平锁 NonfairSync

- 非公平锁是多个线程加锁时直接尝试获取锁，获取不到才会到等待队列的队尾等待。但如果此时锁刚好可用，那么这个线程可以无需阻塞直接获取到锁
- 非公平锁的优点是可以减少唤起线程的开销，整体的吞吐效率高，因为线程有几率不阻塞直接获得锁，CPU 不必唤醒所有线程。缺点是处于等待队列中的线程可

能会饿死，或者等很久才会获得锁

默认创建的对象lock()的时候：

- 如果锁当前没有被其它线程占用，并且当前线程之前没有获取过该锁，则当前线程会获取到该锁，然后设置当前锁的拥有者为当前线程，并设置 AQS 的状态值为1，然后直接返回。如果当前线程之前已经获取过该锁，则这次只是简单地把 AQS 的状态值加1后返回。
- 如果该锁已经被其他线程持有，非公平锁会尝试去获取锁，获取失败的话，则调用该方法线程会被放入 AQS 队列阻塞挂起。



31. ReentrantLock怎么实现公平锁的？

`new ReentrantLock()` 构造函数默认创建的是非公平锁 NonfairSync

```
1 | public ReentrantLock() {  
2 |     sync = new NonfairSync();  
3 | }
```

同时也可以在创建锁构造函数中传入具体参数创建公平锁 FairSync

```
1 | ReentrantLock lock = new ReentrantLock(true);
2 | --- ReentrantLock
3 | // true 代表公平锁, false 代表非公平锁
4 | public ReentrantLock(boolean fair) {
5 |     sync = fair ? new FairSync() : new NonfairSync();
6 | }
```

FairSync、NonfairSync 代表公平锁和非公平锁，两者都是 ReentrantLock 静态内部类，只不过实现不同锁语义。

非公平锁和公平锁的两处不同：

1. 非公平锁在调用 lock 后，首先就会调用 CAS 进行一次抢锁，如果这个时候恰巧锁没有被占用，那么直接就获取到锁返回了。
2. 非公平锁在 CAS 失败后，和公平锁一样都会进入到 tryAcquire 方法，在 tryAcquire 方法中，如果发现锁这个时候被释放了（state == 0），非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁，乖乖排到后面。

```
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}

final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    // State 等于0表示此时无锁
    if (c == 0) {
        // 再次使用CAS尝试获取锁，表现为非公平锁特性
        if (compareAndSetState(0, acquires)) {
            // 设置线程为独占锁线程
            setExclusiveOwnerThread(current);
            return true;
        }
        // 如果当前线程等于已获取锁线程，表现为可重入锁特性
    } else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        // 设置 State
        setState(nextc);
        return true;
    }
    // 如果state不等于0并且独占线程不是当前线程，返回 false
    return false;
}
```

相对来说，非公平锁会有更好的性能，因为它的吞吐量比较大。当然，非公平锁让获取锁的时间变得更加不确定，可能会导致在阻塞队列中的线程长期处于饥饿状态。

32.CAS呢？CAS了解多少？

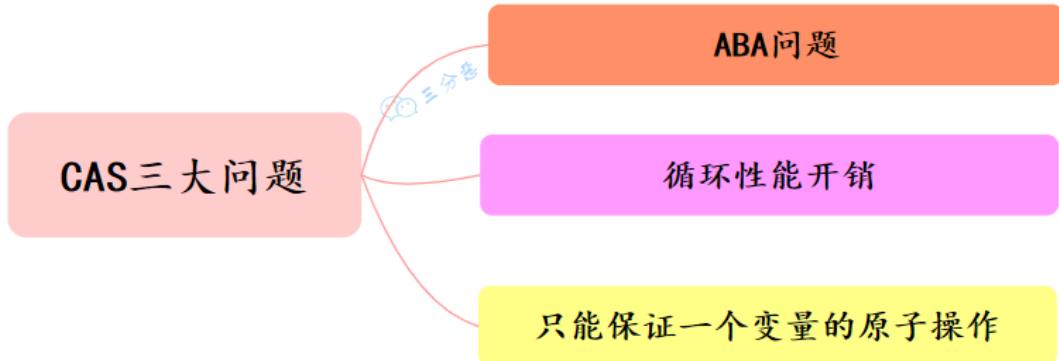
CAS叫做CompareAndSwap，比较并交换，主要是通过处理器的指令来保证操作的原子性的。

CAS 指令包含 3 个参数：共享变量的内存地址 A、预期的值 B 和共享变量的新值 C。

只有当内存中地址 A 处的值等于 B 时，才能将内存中地址 A 处的值更新为新值 C。作为一条 CPU 指令，CAS 指令本身是能够保证原子性的。

33.CAS有什么问题？如何解决？

CAS的经典三大问题：



H5 ABA问题

并发环境下，假设初始条件是A，去修改数据时，发现还是A就会执行修改。但是看到的虽然是A，中间可能发生了A变B，B又变回A的情况。此时A已经非彼A，数据即使成功修改，也可能有问题。

怎么解决ABA问题？

- 加版本号

每次修改变量，都在这个变量的版本号上加1，这样，刚刚A->B->A，虽然A的值没变，但是它的版本号已经变了，再判断版本号就会发现此时的A已经被改过了。参考乐观锁的版本号，这种做法可以给数据带上了一种实效性的检验。

Java提供了AtomicStampReference类，它的compareAndSet方法首先检查当前的对象引用值是否等于预期引用，并且当前印戳（Stamp）标志是否等于预期标志，如果全部相等，则以原子方式将引用值和印戳标志的值更新为给定的更新值。

H5 循环性能开销

自旋CAS，如果一直循环执行，一直不成功，会给CPU带来非常大的执行开销。

怎么解决循环性能开销问题？

在Java中，很多使用自旋CAS的地方，会有一个自旋次数的限制，超过一定次数，就停止自旋。

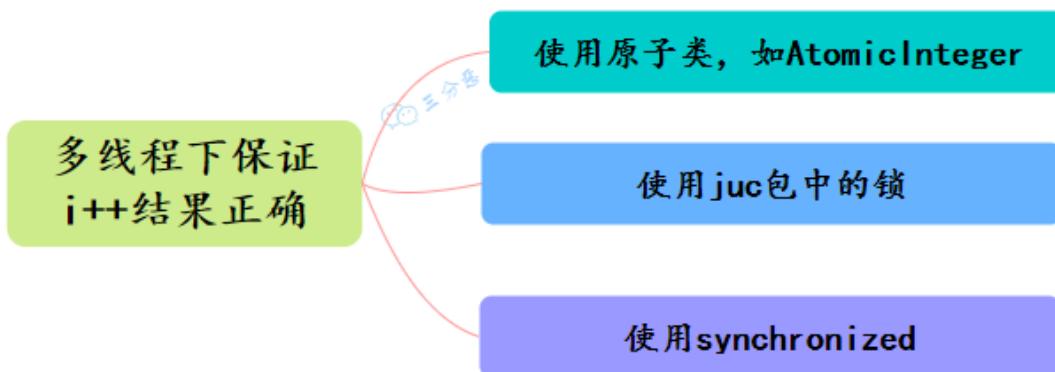
H5 只能保证一个变量的原子操作

CAS 保证的是对一个变量执行操作的原子性，如果对多个变量操作时，CAS 目前无法直接保证操作的原子性的。

怎么解决只能保证一个变量的原子操作问题？

- 可以考虑改用锁来保证操作的原子性
- 可以考虑合并多个变量，将多个变量封装成一个对象，通过AtomicReference来保证原子性。

34. Java有哪些保证原子性的方法？如何保证多线程下i++结果正确？



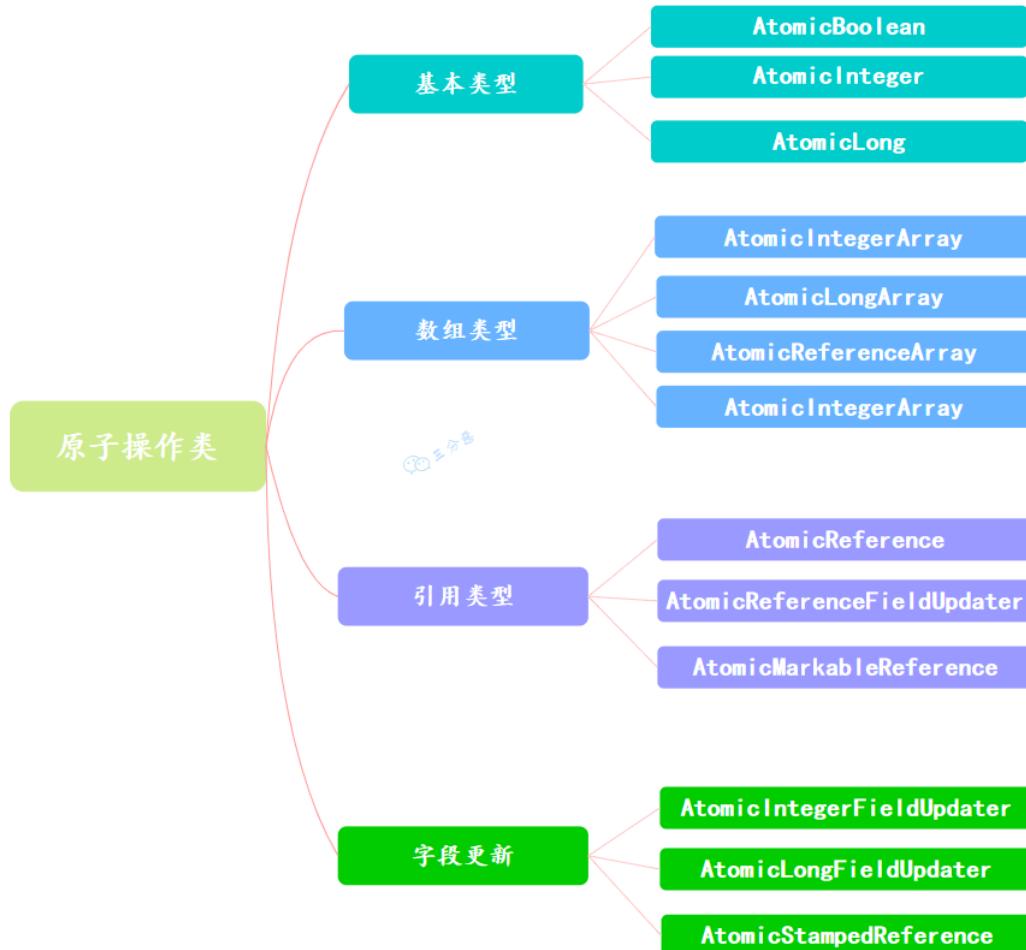
- 使用循环原子类，例如`AtomicInteger`，实现`i++`原子操作
- 使用juc包下的锁，如`ReentrantLock`，对`i++`操作加锁`lock.lock()`来实现原子性
- 使用`synchronized`，对`i++`操作加锁

35. 原子操作类了解多少？

当程序更新一个变量时，如果多线程同时更新这个变量，可能得到期望之外的值，比如变量`i=1`，A线程更新`i+1`，B线程也更新`i+1`，经过两个线程操作之后可能`i`不等于3，而是等于2。因为A和B线程在更新变量`i`的时候拿到的`i`都是1，这就是线程不安全的更新操作，一般我们会使用`synchronized`来解决这个问题，`synchronized`会保证多线程不会同时更新变量`i`。

其实除此之外，还有更轻量级的选择，Java从JDK 1.5开始提供了`java.util.concurrent.atomic`包，这个包中的原子操作类提供了一种用法简单、性能高效、线程安全地更新一个变量的方式。

因为变量的类型有很多种，所以在Atomic包里一共提供了13个类，属于4种类型的原子更新方式，分别是原子更新基本类型、原子更新数组、原子更新引用和原子更新属性（字段）。



Atomic包里的类基本都是使用Unsafe实现的包装类。

使用原子的方式更新基本类型，Atomic包提供了以下3个类：

- `AtomicBoolean`: 原子更新布尔类型。
- `AtomicInteger`: 原子更新整型。
- `AtomicLong`: 原子更新长整型。

通过原子的方式更新数组里的某个元素，Atomic包提供了以下4个类：

- `AtomicIntegerArray`: 原子更新整型数组里的元素。
- `AtomicLongArray`: 原子更新长整型数组里的元素。
- `AtomicReferenceArray`: 原子更新引用类型数组里的元素。
- `AtomicIntegerArrayList`: 提供原子的方式更新数组里的整型

原子更新基本类型的`AtomicInteger`，只能更新一个变量，如果要原子更新多个变量，就需要使用这个原子更新引用类型提供的类。Atomic包提供了以下3个类：

- `AtomicReference`: 原子更新引用类型。

- **AtomicReferenceFieldUpdater**: 原子更新引用类型里的字段。
- **AtomicMarkableReference**: 原子更新带有标记位的引用类型。可以原子更新一个布尔类型的标记位和引用类型。构造方法是**AtomicMarkableReference (V initialRef, boolean initialMark)**。

如果需原子地更新某个类里的某个字段时，就需要使用原子更新字段类，**Atomic**包提供了以下3个类进行原子字段更新：

- **AtomicIntegerFieldUpdater**: 原子更新整型的字段的更新器。
- **AtomicLongFieldUpdater**: 原子更新长整型字段的更新器。
- **AtomicStampedReference**: 原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于原子的更新数据和数据的版本号，可以解决使用**CAS**进行原子更新时可能出现的 ABA问题。

36.AtomicInteger 的原理？

一句话概括：使用**CAS**实现。

以**AtomicInteger**的添加方法为例：

```
1  public final int getAndIncrement() {
2      return unsafe.getAndAddInt(this, valueOffset, 1);
3 }
```

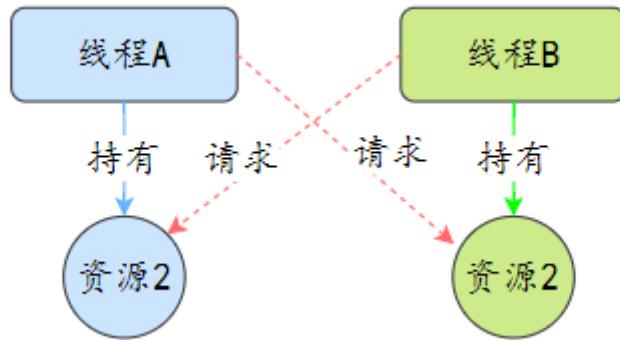
通过 **Unsafe** 类的实例来进行添加操作，来看看具体的**CAS**操作：

```
1  public final int getAndAddInt(Object var1, long var2, int
2      var4) {
3          int var5;
4          do {
5              var5 = this.getIntVolatile(var1, var2);
6              } while(!this.compareAndSwapInt(var1, var2, var5,
7                  var5 + var4));
8          return var5;
9      }
```

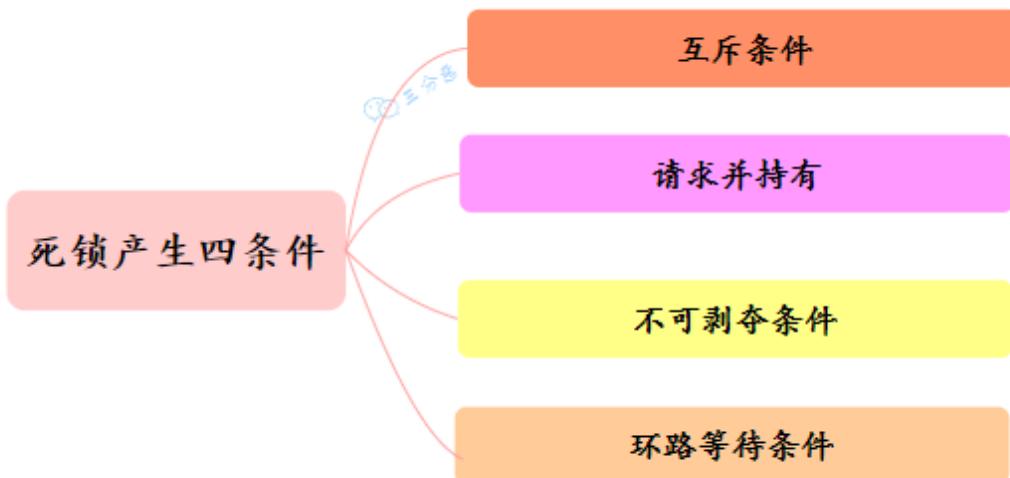
compareAndSwapInt 是一个native方法，基于**CAS**来操作int类型变量。其它的原子操作类基本都是大同小异。

37.线程死锁了解吗？该如何避免？

死锁是指两个或两个以上的线程在执行过程中，因争夺资源而造成的互相等待的现象，在无外力作用的情况下，这些线程会一直相互等待而无法继续运行下去。



那么为什么会产生死锁呢？死锁的产生必须具备以下四个条件：



- 互斥条件：指线程对已经获取到的资源进行独占使用，即该资源同时只由一个线程占用。如果此时还有其它线程请求获取该资源，则请求者只能等待，直至占有资源的线程释放该资源。
- 请求并持有条件：指一个线程已经持有了至少一个资源，但又提出了新的资源请求，而新资源已被其它线程占有，所以当前线程会被阻塞，但阻塞的同时并不释放自己已经获取的资源。
- 不可剥夺条件：指线程获取到的资源在自己使用完之前不能被其它线程抢占，只有在自己使用完毕后才由自己释放该资源。
- 环路等待条件：指在发生死锁时，必然存在一个线程——资源的环形链，即线程集合 {T0, T1, T2, ……, Tn} 中 T0 正在等待 T1 占用的资源，T1 正在等待 T2 占用的资源，…… Tn 在等待 T0 占用的资源。

该如何避免死锁呢？答案是至少破坏死锁发生的一个条件。

- 其中，互斥这个条件我们没有办法破坏，因为用锁为的就是互斥。不过其他三个条件都是有办法破坏掉的，到底如何做呢？
- 对于“请求并持有”这个条件，可以一次性请求所有的资源。
- 对于“不可剥夺”这个条件，占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源，这样不可抢占这个条件就破坏掉了。
- 对于“环路等待”这个条件，可以靠按序申请资源来预防。所谓按序申请，是指资源是有线性顺序的，申请的时候可以先申请资源序号小的，再申请资源序号大的，这样线性化后就不存在环路了。

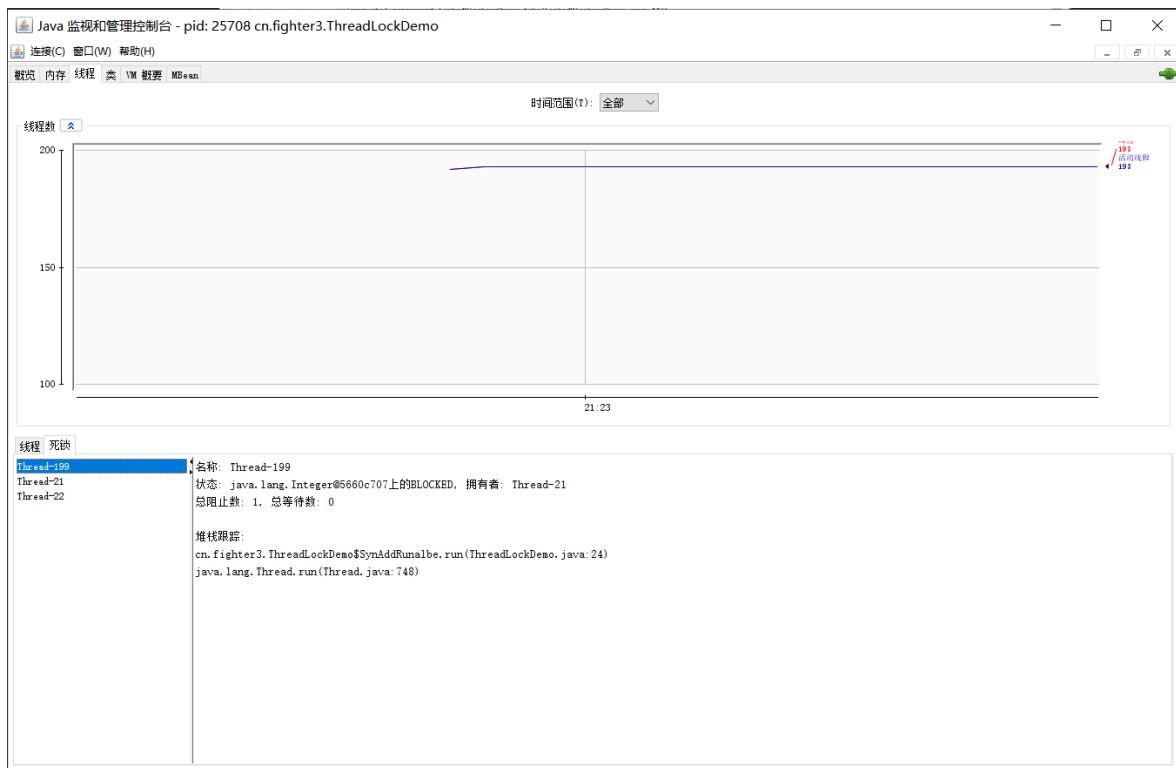
38.那死锁问题怎么排查呢？

可以使用jdk自带的命令行工具排查：

1. 使用jps查找运行的Java进程： jps -l
2. 使用jstack查看线程堆栈信息： jstack -l 进程id

基本就可以看到死锁的信息。

还可以利用图形化工具，比如JConsole。出现线程死锁以后，点击JConsole线程面板的 检测到死锁 按钮，将会看到线程的死锁信息。



并发工具类

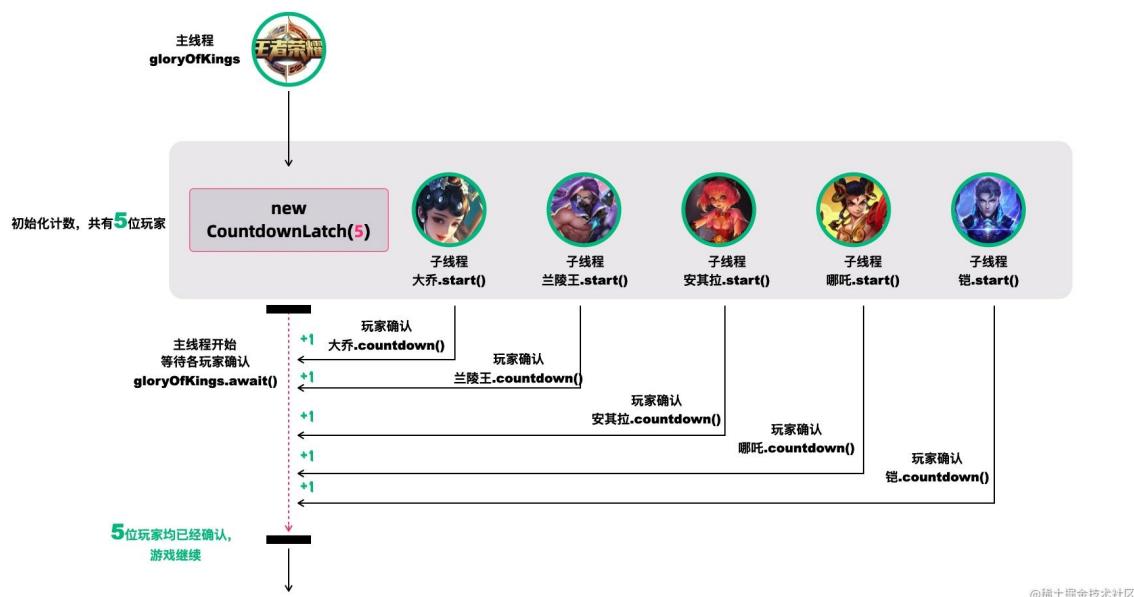
39.CountDownLatch（倒计数器）了解吗？

CountDownLatch，倒计数器，有两个常见的应用场景[18]:

场景1：协调子线程结束动作：等待所有子线程运行结束

CountDownLatch允许一个或多个线程等待其他线程完成操作。

例如，我们很多人喜欢玩的王者荣耀，开黑的时候，得等所有人都上线之后，才能开打。



CountDownLatch模仿这个场景(参考[18]):

创建大乔、兰陵王、安其拉、哪吒和铠等五个玩家，主线程必须在他们都完成确认后，才可以继续运行。

在这段代码中，`new CountDownLatch(5)` 用户创建初始的latch数量，各玩家通过`countDown()` 完成状态确认，主线程通过`await()` 等待。

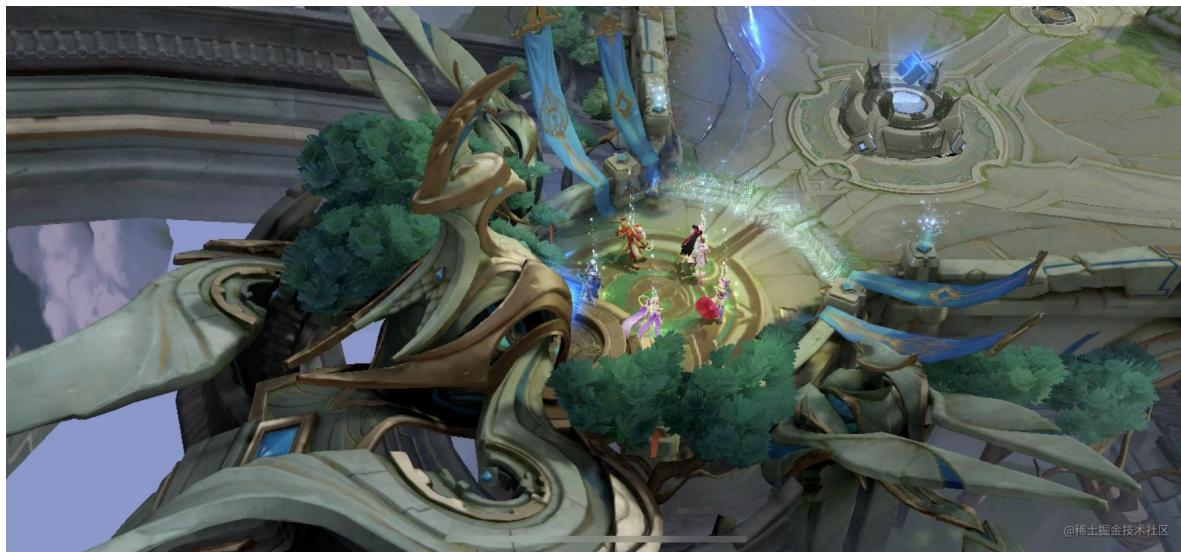
```
1 public static void main(String[] args) throws
2     InterruptedException {
3     CountDownLatch countDownLatch = new
4     CountDownLatch(5);
5
6     Thread 大乔 = new Thread(countDownLatch::countDown);
7     Thread 兰陵王 = new Thread(countDownLatch::countDown);
8     Thread 安其拉 = new Thread(countDownLatch::countDown);
```

```
7     Thread 哪吒 = new Thread(countDownLatch::countDown);
8     Thread 铠 = new Thread(() -> {
9         try {
10             // 稍等，上个卫生间，马上到...
11             Thread.sleep(1500);
12             countDownLatch.countDown();
13         } catch (InterruptedException ignored) {}
14     });
15
16     大乔.start();
17     兰陵王.start();
18     安其拉.start();
19     哪吒.start();
20     铠.start();
21     countDownLatch.await();
22     System.out.println("所有玩家已经就位! ");
23 }
```

场景2. 协调子线程开始动作：统一各线程动作开始的时机

王者游戏中也有类似的场景，游戏开始时，各玩家的初始状态必须一致。不能有的玩家都出完装了，有的才降生。

所以大家得一块出生，在



在这个场景中，仍然用五个线程代表大乔、兰陵王、安其拉、哪吒和铠等五个玩家。需要注意的是，各玩家虽然都调用了 `start()` 线程，但是它们在运行时都在等待 `countDownLatch` 的信号，在信号未收到前，它们不会往下执行。

```

1     public static void main(String[] args) throws
2         InterruptedException {
3             CountDownLatch countDownLatch = new
4                 CountDownLatch(1);
5
6             Thread 大乔 = new Thread(() ->
7                 waitToFight(countDownLatch));
8             Thread 兰陵王 = new Thread(() ->
9                 waitToFight(countDownLatch));
10            Thread 安其拉 = new Thread(() ->
11                waitToFight(countDownLatch));
12            Thread 哪吒 = new Thread(() ->
13                waitToFight(countDownLatch));
14            Thread 铠 = new Thread(() ->
15                waitToFight(countDownLatch));
16
17            大乔.start();
18            兰陵王.start();
19            安其拉.start();
20            哪吒.start();
21            铠.start();
22            Thread.sleep(1000);
23            countDownLatch.countDown();
24            System.out.println("敌方还有5秒达到战场，全军出击！");
25        }
26
27    }

28
29     private static void waitToFight(CountDownLatch
30         countDownLatch) {
31         try {
32             countDownLatch.await(); // 在此等待信号再继续
33             System.out.println("收到，发起进攻！");
34         } catch (InterruptedException e) {
35             e.printStackTrace();
36         }
37     }

```

CountDownLatch的**核心方法**也不多：

- `await()` : 等待latch降为0;
- `boolean await(long timeout, TimeUnit unit)` : 等待latch降为0，但是可以设置超时时间。比如有玩家超时未确认，那就重新匹配，总不能为了某个玩家等到天荒地老。

- `countDown()` : latch数量减1;
- `getCount()` : 获取当前的latch数量。

40.CyclicBarrier（同步屏障）了解吗？

CyclicBarrier的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续运行。

它和CountDownLatch类似，都可以协调多线程的结束动作，在它们结束后都可以执行特定动作，但是为什么要CyclicBarrier，自然是它有和CountDownLatch不同的地方。

不知道你听没听过一个新人UP主小约翰可汗，小约翰生平有两大恨——“想结衣结衣不依,迷爱理爱理不理。”我们来还原一下事情的经过：小约翰在亲政后认识了新垣结衣，于是决定第一次选妃，向结衣表白，等待回应。然而新垣结衣回应嫁给了星野源，小约翰伤心欲绝，发誓生平不娶，突然发现了铃木爱理，于是小约翰决定第二次选妃，求爱理搭理，等待回应。

想结衣结衣不依,迷爱理爱理不理。



我们拿代码模拟这一场景，发现CountDownLatch无能为力了，因为CountDownLatch的使用是一次性的，无法重复利用，而这里等待了两次。此时，我们用CyclicBarrier就可以实现，因为它可以重复利用。

```
● ● ●

public class CyclicBarrierTest {
    //第几次选妃
    private static String draftTime = "选妃#1";

    public static void main(String[] args) {
        //创建栅栏
        CyclicBarrier cyclicBarrier = new CyclicBarrier(2, () -> System.out.println("皇上通令不能没有妃子，正如西方不能没有耶路撒冷"+draftTime));

        Thread 小约翰可汗 = new Thread(() -> {
            say("我是新人王小约翰可汗。。。");
            try {
                //第一次调用await
                cyclicBarrier.await();
                say("结衣，你依不依。。。");
                Thread.sleep(2600); //等待结衣回应...
                draftTime = "选妃#2"; // 第二次选妃
                Thread.sleep(1500); //求爱理搭理中....
                note("可汗大点兵，仓鼠聚通汗。。。");
                cyclicBarrier.await(); // 可汗聚兵，仓鼠十万，选妃*2 ！！！注意这里是第二次调用await
                Thread.sleep(100);
                say("猛鼠落泪！！");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }, "小约翰可汗");

        Thread 新垣结衣 = new Thread(() -> {
            try {
                Thread.sleep(500); //铠打野中
                say("我是结衣啊，什么汗？");
                cyclicBarrier.await(); //等待中...
                Thread.sleep(1000); //忽略了...
                say("我已经和星野源结婚啦！"); //小约翰可汗突然看到了铃木爱理。
            } catch (Exception ignored) {}
        }, "新垣结衣");

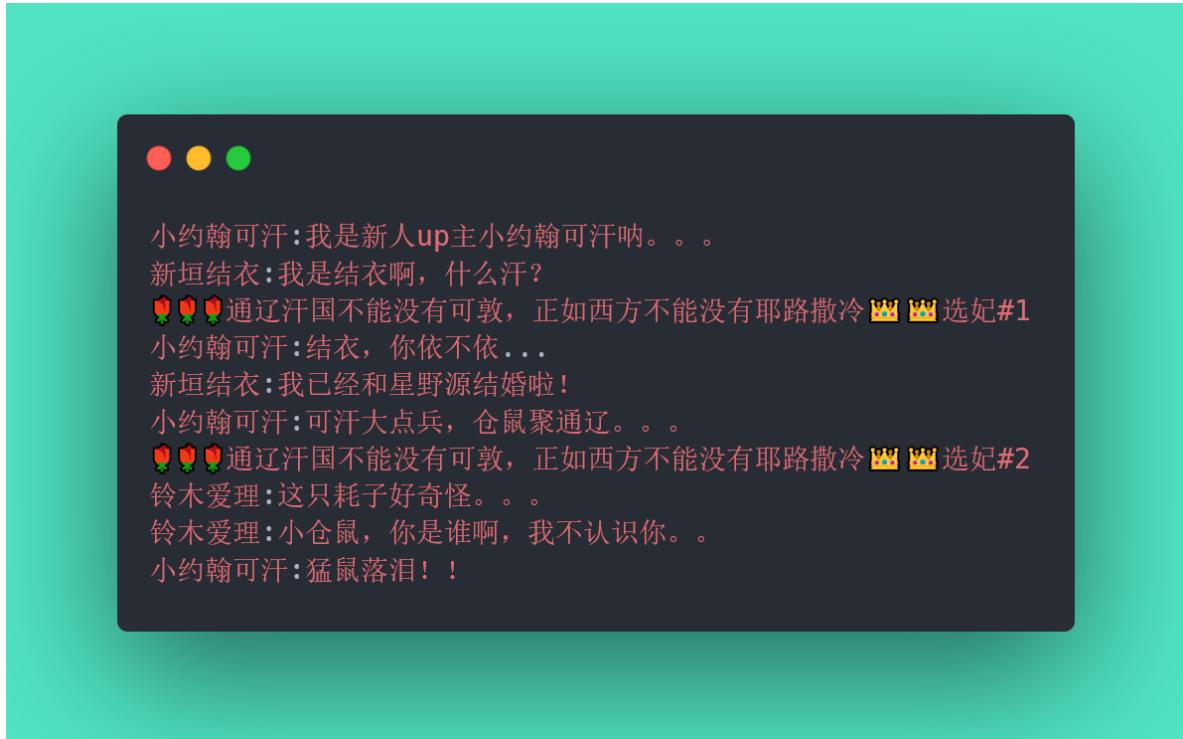
        Thread 铃木爱理 = new Thread(() -> {
            try {
                Thread.sleep(2500);
                cyclicBarrier.await();
                note("这只耗子好奇怪。。。");
                say("小仓鼠，你是谁啊，我不认识你。。。");
            } catch (Exception ignored) {}
        }, "铃木爱理");
    }

    小约翰可汗.start();
    新垣结衣.start();
    铃木爱理.start();
}

private static void note(String s) {
    System.out.println(Thread.currentThread().getName()+": "+s);
}

private static void say(String s) {
    System.out.println(Thread.currentThread().getName()+": "+s);
}
}
```

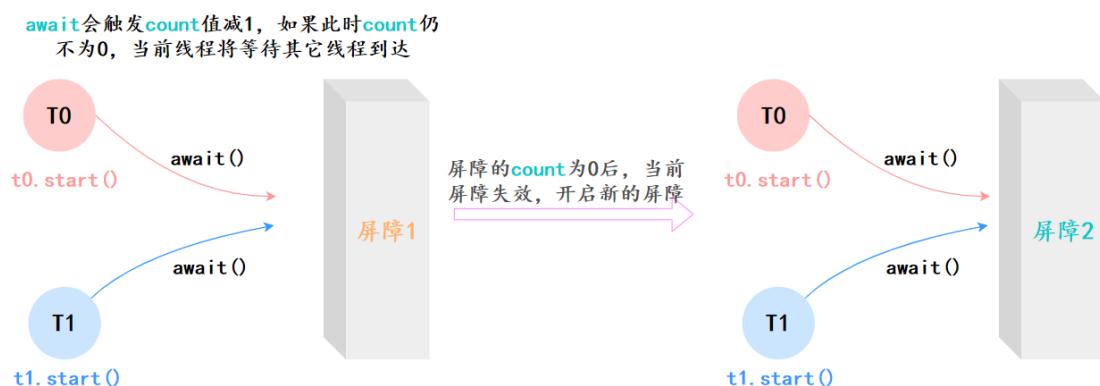
运行结果：



CyclicBarrier最核心的方法，仍然是await():

- 如果当前线程不是第一个到达屏障的话，它将会进入等待，直到其他线程都到达，除非发生被中断、屏障被拆除、屏障被重设等情况；

上面的例子抽象一下，本质上它的流程就是这样就是这样：



41.CyclicBarrier和CountDownLatch有什么区别？

两者最核心的区别[18]:

- CountDownLatch是一次性的，而CyclicBarrier则可以多次设置屏障，实现重复利用；
- CountDownLatch中的各个子线程不可以等待其他线程，只能完成自己的任务；而CyclicBarrier中的各个线程可以等待其他线程

它们区别用一个表格整理：

CyclicBarrier	CountDownLatch
CyclicBarrier是可重用的，其中的线程会等待所有的线程完成任务。届时，屏障将被拆除，并可以选择性地做一些特定的动作。	CountDownLatch是一次性的，不同的线程在同一个计数器上工作，直到计数器为0.
CyclicBarrier面向的是线程数	CountDownLatch面向的是任务数
在使用CyclicBarrier时，你必须在构造中指定参与协作的线程数，这些线程必须调用await()方法	使用CountDownLatch时，则必须要指定任务数，至于这些任务由哪些线程完成无关紧要
CyclicBarrier可以在所有的线程释放后重新使用	CountDownLatch在计数器为0时不能再使用
在CyclicBarrier中，如果某个线程遇到了中断、超时等问题时，则处于await的线程都会出现问题	在CountDownLatch中，如果某个线程出现问题，其他线程不受影响

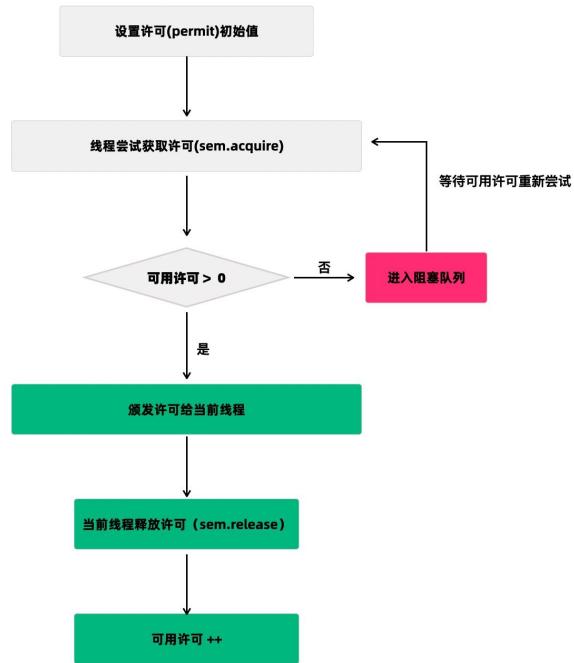
42.Semaphore（信号量）了解吗？

Semaphore（信号量）是用来控制同时访问特定资源的线程数量，它通过协调各个线程，以保证合理的使用公共资源。

听起来似乎很抽象，现在汽车多了，开车出门在外的一个老大难问题就是停车。停车场的车位是有限的，只能允许若干车辆停泊，如果停车场还有空位，那么显示屏显示的就是绿灯和剩余的车位，车辆就可以驶入；如果停车场没位了，那么显示屏显示的就是绿灯和数字0，车辆就得等待。如果满了的停车场有车离开，那么显示屏就又变绿，显示空车位数量，等待的车辆就能进停车场。



我们把这个例子类比一下，车辆就是线程，进入停车场就是线程在执行，离开停车场就是线程执行完毕，看见红灯就表示线程被阻塞，不能执行，Semaphore的本质就是**协调多个线程对共享资源的获取**。



©博士锐金技术社区

我们再来看一个Semaphore的用途：它可以用于做流量控制，特别是公用资源有限的应用场景，比如数据库连接。

假如有一个需求，要读取几万个文件的数据，因为都是IO密集型任务，我们可以启动几十个线程并发地读取，但是如果读到内存后，还需要存储到数据库中，而数据库的连接数只有10个，这时我们必须控制只有10个线程同时获取数据库连接保存数据，否则会报错无法获取数据库连接。这个时候，就可以使用Semaphore来做流量控制，如下：

```
1 public class SemaphoreTest {
2     private static final int THREAD_COUNT = 30;
3     private static ExecutorService threadPool =
4         Executors.newFixedThreadPool(THREAD_COUNT);
5     private static Semaphore s = new Semaphore(10);
6
7     public static void main(String[] args) {
8         for (int i = 0; i < THREAD_COUNT; i++) {
9             threadPool.execute(new Runnable() {
10                 @Override
11                 public void run() {
12                     try {
13                         s.acquire();
14                         System.out.println("save data");
15                         s.release();
16                     } catch (InterruptedException e) {
17                         }
18                     }
19                 });
20             threadPool.shutdown();
21         }
22     }
}
```

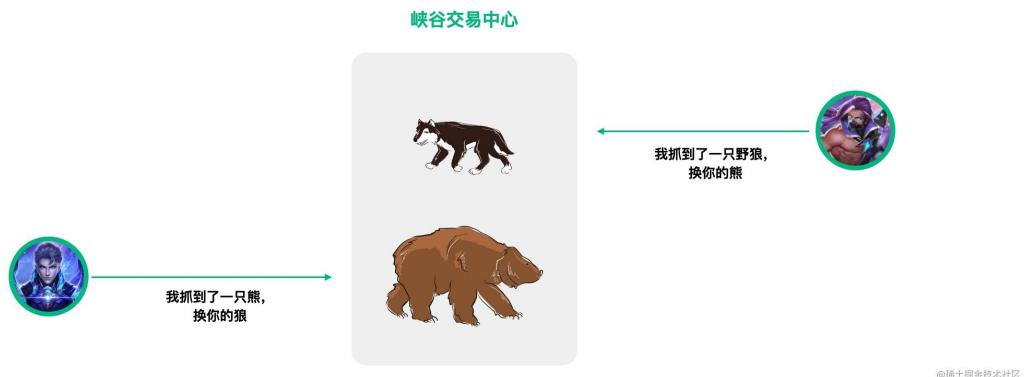
在代码中，虽然有30个线程在执行，但是只允许10个并发执行。Semaphore的构造方法 `Semaphore (int permits)` 接受一个整型的数字，表示可用的许可证数量。

`Semaphore (10)` 表示允许10个线程获取许可证，也就是最大并发数是10。

Semaphore的用法也很简单，首先线程使用 Semaphore的acquire()方法获取一个许可证，使用完之后调用release()方法归还许可证。还可以用tryAcquire()方法尝试获取许可证。

43.Exchanger 了解吗？

Exchanger（交换者）是一个用于线程间协作的工具类。Exchanger用于进行线程间的数据交换。它提供一个同步点，在这个同步点，两个线程可以交换彼此的数据。



©稀土掘金技术社区

这两个线程通过 `exchange`方法交换数据，如果第一个线程先执行`exchange()`方法，它会一直等待第二个线程也执行`exchange`方法，当两个线程都到达同步点时，这两个线程就可以交换数据，将本线程生产出来的数据传递给对方。

Exchanger可以用于遗传算法，遗传算法里需要选出两个人作为交配对象，这时候会交换两人的数据，并使用交叉规则得出2个交配结果。Exchanger也可以用于校对工作，比如我们需要将纸制银行流水通过人工的方式录入成电子银行流水，为了避免错误，采用AB岗两人进行录入，录入到Excel之后，系统需要加载这两个Excel，并对两个Excel数据进行校对，看看是否录入一致。

```
1 public class ExchangerTest {
2     private static final Exchanger<String> exgr = new
3     Exchanger<String>();
4     private static ExecutorService threadPool =
5     Executors.newFixedThreadPool(2);
6
7     public static void main(String[] args) {
8         threadPool.execute(new Runnable() {
9             @Override
10            public void run() {
11                try {
12                    String A = "银行流水A"; // A录入银行流水数据
13                    exgr.exchange(A);
14                } catch (InterruptedException e) {
15                }
16            }
17        });
18        threadPool.execute(new Runnable() {
19            @Override
20            public void run() {
```

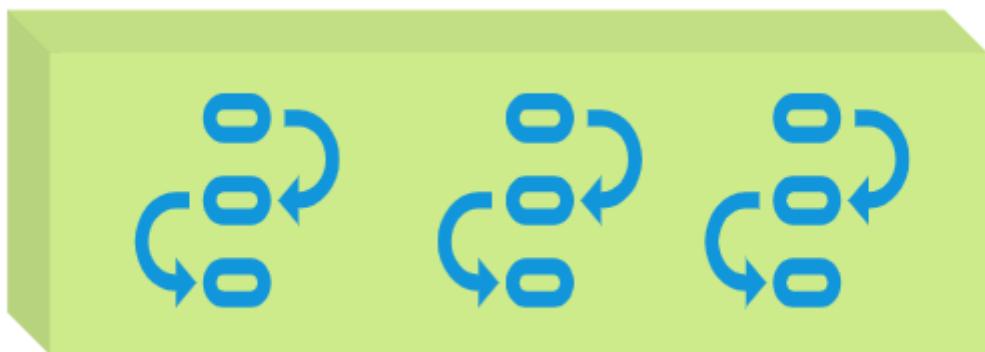
```
19     try {
20         String B = "银行流水B"; // B录入银行流水数据
21         String A = exgr.exchange("B");
22         System.out.println("A和B数据是否一致: " +
23             A.equals(B) + ", A录入的是: " +
24             + A + ", B录入是: " + B);
25     } catch (InterruptedException e) {
26     }
27 });
28 threadPool.shutdown();
29 }
30 }
```

假如两个线程有一个没有执行exchange()方法，则会一直等待，如果担心有特殊情况发生，避免一直等待，可以使用 `exchange(V x, long timeOut, TimeUnit unit)` 设置最大等待时长。

线程池

44.什么是线程池？

线程池： 简单理解，它就是一个管理线程的池子。

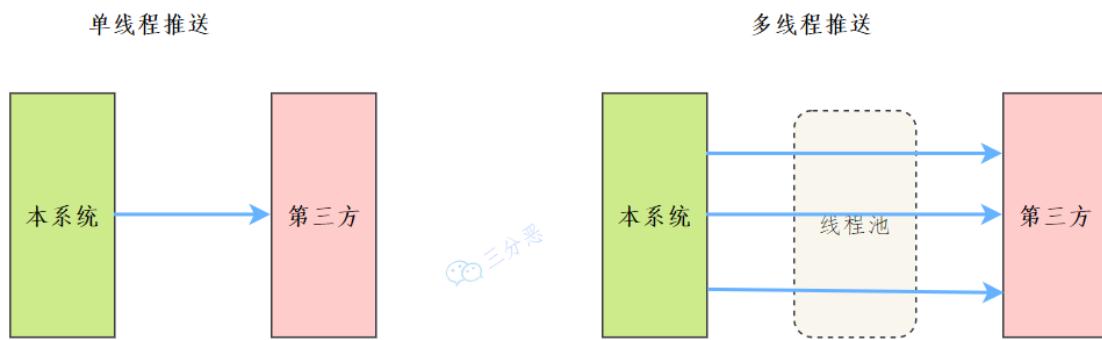


- 它帮我们管理线程，避免增加创建线程和销毁线程的资源损耗。因为线程其实也是一个对象，创建一个对象，需要经过类加载过程，销毁一个对象，需要走GC垃圾回收流程，都是需要资源开销的。
- 提高响应速度。如果任务到达了，相对于从线程池拿线程，重新去创建一条线程执行，速度肯定慢很多。

- 重复利用。 线程用完，再放回池子，可以达到重复利用的效果，节省资源。

45.能说说工作中线程池的应用吗？

之前我们有一个和第三方对接的需求，需要向第三方推送数据，引入了多线程来提升数据推送的效率，其中用到了线程池来管理线程。



主要代码如下：

```
● ● ●

@Service
public class PushProcessServiceImpl implements PushProcessService {
    @Autowired
    private PushUtil pushUtil;
    @Autowired
    private PushProcessMapper pushProcessMapper;

    private final static Logger logger = LoggerFactory.getLogger(PushProcessServiceImpl.class);

    //每个线程每次查询的条数
    private static final Integer LIMIT = 5000;
    //核心线程数:设置为操作系统CPU数乘以2
    private static final Integer CORE_POOL_SIZE = Runtime.getRuntime().availableProcessors() * 2;
    //最大线程数:设置为和核心线程数相同
    private static final Integer MAXIMUM_POOL_SIZE = CORE_POOL_SIZE;
    //创建线程池
    ThreadPoolExecutor pool = new ThreadPoolExecutor(CORE_POOL_SIZE, MAXIMUM_POOL_SIZE * 2, 0,
    TimeUnit.SECONDS, new LinkedBlockingQueue<>(100));

    @Override
    public void pushData() throws ExecutionException, InterruptedException {
        //计数器, 需要保证线程安全
        int count = 0;
        //未推送数据总数
        Integer total = pushProcessMapper.countPushRecordsByState(0);
        logger.info("未推送数据条数: {}", total);
        //计算需要多少轮
        int num = total / (LIMIT * CORE_POOL_SIZE) + 1;
        logger.info("要经过的轮数:{}", num);
        //统计总共推送成功的数据条数
        int totalSuccessCount = 0;
        for (int i = 0; i < num; i++) {
            //接收线程返回结果
            List<Future<Integer>> futureList = new ArrayList<>(32);
            //起CORE_POOL_SIZE个线程并行查询更新库, 加锁
            for (int j = 0; j < CORE_POOL_SIZE; j++) {
                synchronized (PushProcessServiceImpl.class) {
                    int start = count * LIMIT;
                    count++;
                    //提交线程, 用数据起始位置标识线程
                    Future<Integer> future = pool.submit(new PushDataTask(start, LIMIT, start));
                    //先不取值, 防止阻塞, 放进集合
                    futureList.add(future);
                }
            }
            //统计本轮推送成功数据
            for (Future f : futureList) {
                totalSuccessCount = totalSuccessCount + (int) f.get();
            }
        }
        //关闭线程池
        //pool.shutdown();
        //更新推送标志
        pushProcessMapper.updateAllState(1);
        logger.info("推送数据完成, 需推送数据:{}条, 推送成功: {}, total, totalSuccessCount");
    }

    /**
     * 推送数据线程类
     */
    class PushDataTask implements Callable<Integer> {
        int start;
        int limit;

        PushDataTask(int start, int limit, int threadNo) {
            this.start = start;
            this.limit = limit;
        }

        @Override
        public Integer call() throws Exception {
            //设置线程名字
            Thread.currentThread().setName("Thread" + start);
            int count = 0;
            //推送的数据
            List<PushProcess> pushProcessList = pushProcessMapper.findPushRecordsByStateLimit(0, start,
            limit);
            if (CollectionUtils.isEmpty(pushProcessList)) {
                return count;
            }
            logger.info("推送区间: [{},{}]数据", start, start + limit);
            for (PushProcess process : pushProcessList) {
                boolean isSuccess = pushUtil.sendRecord(process);
                if (isSuccess) { //推送成功
                    //更新推送标识
                    pushProcessMapper.updateFlagById(process.getId(), 1);
                    count++;
                } else { //推送失败
                    pushProcessMapper.updateFlagById(process.getId(), 2);
                }
            }
            logger.info("推送区间[{},{}]数据,推送成功{}条! ", start, start + limit, count);
        }
    }
}
```

```
    }  
}  
}  
}  
return count;  
}
```

完整可运行代码地址: <https://gitee.com/fighter3/thread-demo.git>

线程池的参数如下:

- corePoolSize: 线程核心参数选择了CPU数×2
- maximumPoolSize: 最大线程数选择了和核心线程数相同
- keepAliveTime: 非核心闲置线程存活时间直接置为0
- unit: 非核心线程保持存活的时间选择了 TimeUnit.SECONDS 秒
- workQueue: 线程池等待队列, 使用 LinkedBlockingQueue阻塞队列

同时还用了synchronized 来加锁, 保证数据不会被重复推送:

```
1 | synchronized (PushProcessServiceImpl.class) {}
```

ps:这个例子只是简单地进行了数据推送, 实际上还可以结合其他的业务, 像什么数据清洗啊、数据统计啊, 都可以套用。

46.能简单说一下线程池的工作流程吗?

用一个通俗的比喻:

有一个营业厅, 总共有六个窗口, 现在开放了三个窗口, 现在有三个窗口坐着三个营业员小姐姐在营业。

老三去办业务, 可能会遇到什么情况呢?

1. 老三发现有空间的在营业的窗口, 直接去找小姐姐办理业务。



2. 老三发现没有空闲的窗口, 就在排队区排队等。



3. 老三发现没有空闲的窗口，等待区也满了，蚌埠住了，经理一看，就让休息的小姐姐赶紧回来上班，等待区号靠前的赶紧去新窗口办，老三去排队区排队。小姐姐比较辛苦，假如一段时间发现他们可以不用接着营业，经理就让她们接着休息。



4. 老三一看，六个窗口都满了，等待区也没位置了。老三急了，要闹，经理赶紧出来了，经理该怎么办呢？



- 1. 我们银行系统已经瘫痪
- 2. 谁叫你来办的你找谁去
- 3. 看你比较急，去队里加个塞

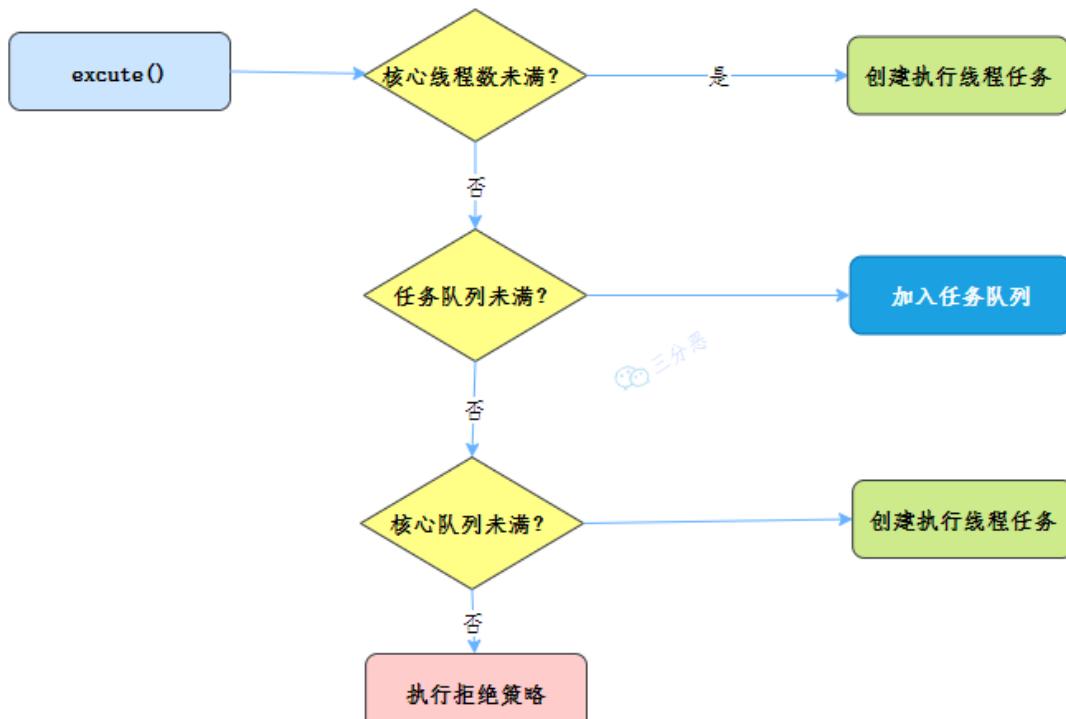
4. 今天没办法，不行你看改一天

上面的这个流程几乎就跟 JDK 线程池的大致流程类似，

1. 营业中的 3 个窗口对应核心线程池数: corePoolSize
2. 总的营业窗口数 6 对应: maximumPoolSize
3. 打开的临时窗口在多少时间内无人办理则关闭对应: unit
4. 排队区就是等待队列: workQueue
5. 无法办理的时候银行给出的解决方法对应: RejectedExecutionHandler
6. threadFactory 该参数在 JDK 中是线程工厂，用来创建线程对象，一般不会动。

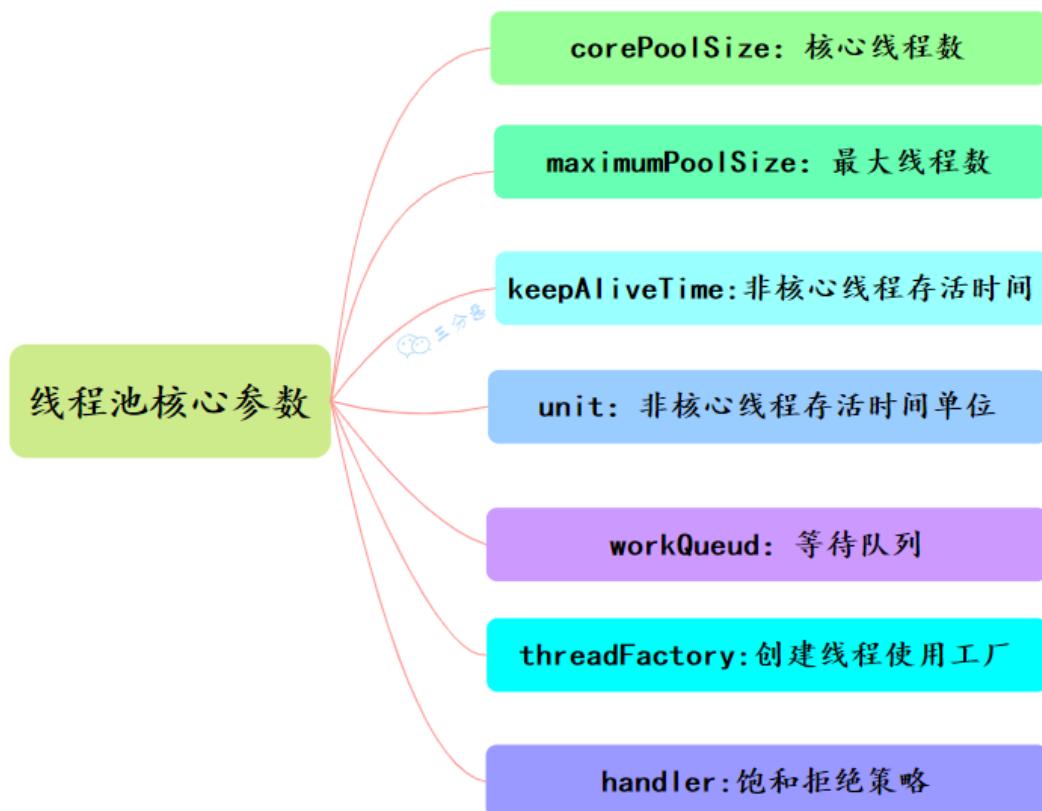
所以我们线程池的工作流程也比较好理解了：

1. 线程池刚创建时，里面没有一个线程。任务队列是作为参数传进来的。不过，就算队列里面有任务，线程池也不会马上执行它们。
2. 当调用 `execute()` 方法添加一个任务时，线程池会做如下判断：
 - 如果正在运行的线程数量小于 `corePoolSize`，那么马上创建线程运行这个任务；
 - 如果正在运行的线程数量大于或等于 `corePoolSize`，那么将这个任务放入队列；
 - 如果这时候队列满了，而且正在运行的线程数量小于 `maximumPoolSize`，那么还是要创建非核心线程立刻运行这个任务；
 - 如果队列满了，而且正在运行的线程数量大于或等于 `maximumPoolSize`，那么线程池会根据拒绝策略来对应处理。



3. 当一个线程完成任务时，它会从队列中取下一个任务来执行。
4. 当一个线程无事可做，超过一定的时间（keepAliveTime）时，线程池会判断，如果当前运行的线程数大于 corePoolSize，那么这个线程就被停掉。所以线程池的所有任务完成后，它最终会收缩到 corePoolSize 的大小。

47. 线程池主要参数有哪些？



线程池有七大参数，需要重点关注 `corePoolSize`、`maximumPoolSize`、`workQueue`、`handler` 这四个。

1. corePoolSize

此值是用来初始化线程池中核心线程数，当线程池中线程数 < `corePoolSize` 时，系统默认是添加一个任务才创建一个线程池。当线程数 = `corePoolSize` 时，新任务会追加到 `workQueue` 中。

2. maximumPoolSize

`maximumPoolSize` 表示允许的最大线程数 = (非核心线程数+核心线程数)，当 `BlockingQueue` 也满了，但线程池中总线程数 < `maximumPoolSize` 时候就会再次创建新的线程。

3. keepAliveTime

非核心线程 = $(\text{maximumPoolSize} - \text{corePoolSize})$, 非核心线程闲置下来不干活最多存活时间。

4. unit

线程池中非核心线程保持存活的时间的单位

- TimeUnit.DAYS; 天
- TimeUnit.HOURS; 小时
- TimeUnit.MINUTES; 分钟
- TimeUnit.SECONDS; 秒
- TimeUnit.MILLISECONDS; 毫秒
- TimeUnit.MICROSECONDS; 微秒
- TimeUnit.NANOSECONDS; 纳秒

5. workQueue

线程池等待队列，维护着等待执行的 `Runnable` 对象。当运行当线程数=`corePoolSize`时，新的任务会被添加到 `workQueue` 中，如果 `workQueue` 也满了则尝试用非核心线程执行任务，等待队列应该尽量用有界的。

6. threadFactory

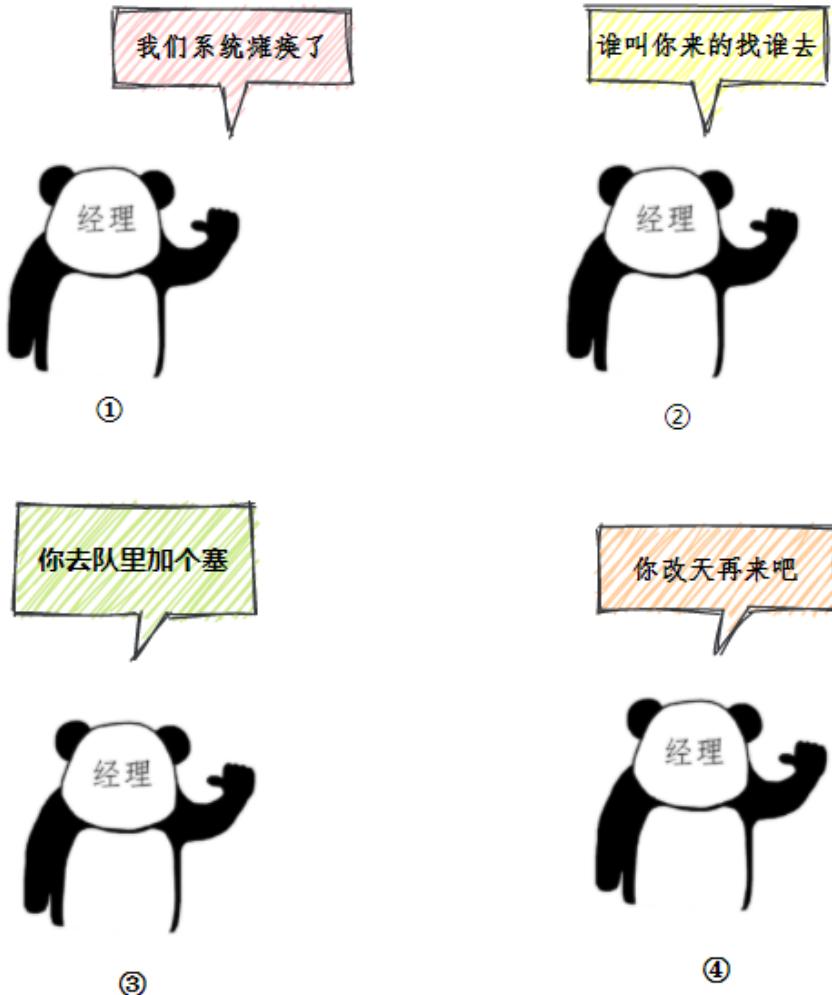
创建一个新线程时使用的工厂，可以用来设定线程名、是否为daemon线程等等。

7. handler

`corePoolSize`、`workQueue`、`maximumPoolSize` 都不可用的时候执行的饱和策略。

48.线程池的拒绝策略有哪些？

类比前面的例子，无法办理业务时的处理方式，帮助记忆：

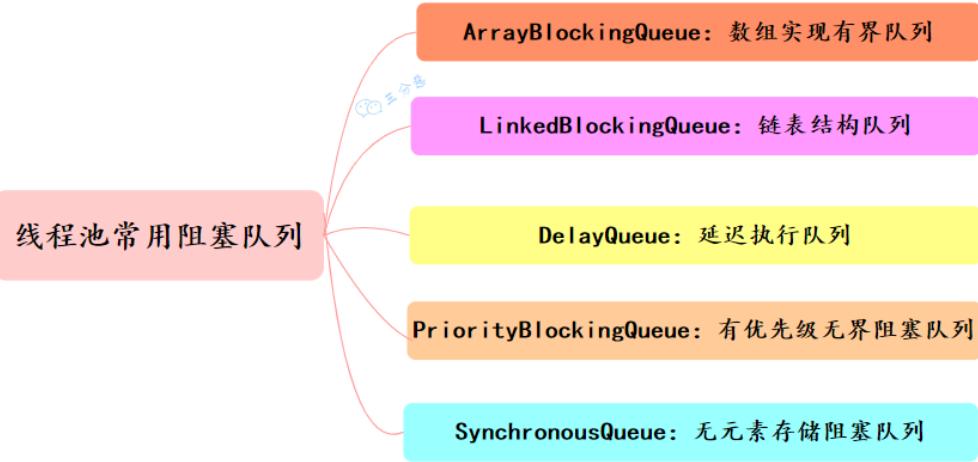


- AbortPolicy : 直接抛出异常， 默认使用此策略
- CallerRunsPolicy: 用调用者所在的线程来执行任务
- DiscardOldestPolicy: 丢弃阻塞队列里最老的任务， 也就是队列里靠前的任务
- DiscardPolicy : 当前任务直接丢弃

想实现自己的拒绝策略， 实现RejectedExecutionHandler接口即可。

49.线程池有哪几种工作队列？

常用的阻塞队列主要有以下几种：



- **ArrayBlockingQueue:** ArrayBlockingQueue（有界队列）是一个用数组实现的有界阻塞队列，按FIFO排序量。
- **LinkedBlockingQueue:** LinkedBlockingQueue（可设置容量队列）是基于链表结构的阻塞队列，按FIFO排序任务，容量可以选择进行设置，不设置的话，将是一个无边界的阻塞队列，最大长度为Integer.MAX_VALUE，吞吐量通常要高于ArrayBlockingQuene；newFixedThreadPool线程池使用了这个队列
- **DelayQueue:** DelayQueue（延迟队列）是一个任务定时周期的延迟执行的队列。根据指定的执行时间从小到大排序，否则根据插入到队列的先后排序。newScheduledThreadPool线程池使用了这个队列。
- **PriorityBlockingQueue:** PriorityBlockingQueue（优先级队列）是具有优先级的无界阻塞队列
- **SynchronousQueue:** SynchronousQueue（同步队列）是一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQuene，newCachedThreadPool线程池使用了这个队列。

50. 线程池提交execute和submit有什么区别？

1. execute 用于提交不需要返回值的任务

```

1 threadsPool.execute(new Runnable() {
2     @Override public void run() {
3         // TODO Auto-generated method stub
4     });

```

2. submit()方法用于提交需要返回值的任务。线程池会返回一个future类型的对象，通过这个 future对象可以判断任务是否执行成功，并且可以通过future的get()方法来获取返回值

```
1 Future<Object> future = executor.submit(harReturnValuetask);
2 try { Object s = future.get(); } catch (InterruptedException
e) {
3     // 处理中断异常
4 } catch (ExecutionException e) {
5     // 处理无法执行任务异常
6 } finally {
7     // 关闭线程池 executor.shutdown();
8 }
```

51.线程池怎么关闭知道吗？

可以通过调用线程池的 `shutdown` 或 `shutdownNow` 方法来关闭线程池。它们的原理是遍历线程池中的工作线程，然后逐个调用线程的`interrupt`方法来中断线程，所以无法响应中断的任务可能永远无法终止。

shutdown() 将线程池状态置为**shutdown**,并不会立即停止：

1. 停止接收外部submit的任务
2. 内部正在跑的任务和队列里等待的任务，会执行完
3. 等到第二步完成后，才真正停止

shutdownNow() 将线程池状态置为**stop**。一般会立即停止，事实上不一定：

1. 和shutdown()一样，先停止接收外部提交的任务
2. 忽略队列里等待的任务
3. 尝试将正在跑的任务interrupt中断
4. 返回未执行的任务列表

shutdown 和shutdownnow简单来说区别如下：

- shutdownNow()能立即停止线程池，正在跑的和正在等待的任务都停下了。这样做立即生效，但是风险也比较大。
- shutdown()只是关闭了提交通道，用submit()是无效的；而内部的任务该怎么跑还是怎么跑，跑完再彻底停止线程池。

52.线程池的线程数应该怎么配置？

线程在Java中属于稀缺资源，线程池不是越大越好也不是越小越好。任务分为计算密集型、IO密集型、混合型。

1. 计算密集型：大部分都在用CPU跟内存，加密，逻辑操作业务处理等。
2. IO密集型：数据库链接，网络通讯传输等。

方案	问题
$N_{cpu} = \text{number of CPUs}$ $U_{cpu} = \text{target CPU utilization}, 0 \leq U_{cpu} \leq 1$ $\frac{W}{C} = \text{ratio of wait time to compute time}$ The optimal pool size for keeping the processors at the desired utilization is : $N_{threads} = N_{cpu} * U_{cpu} * (1 + \frac{W}{C})$	出自《Java并发编程实践》 该方案偏理论化。首先，线程计算的时间和等待的时间要如何确定呢？这个在实际开发中很难得到确切的值。另外计算出来的线程个数逼近线程实体的个数，Java线程池可以利用线程切换的方式最大程度利用CPU核数，这样计算出来的结果是非常偏离业务场景的。
$coreSize = 2 * N_{cpu}$ $maxSize = 25 * N_{cpu}$	没有考虑应用中往往使用多个线程池的情况，统一的配置明显不符合多样的业务场景。
$coreSize = tps * time$ $maxSize = tps * time * (1.7 - 2)$	这种计算方式，考虑到了业务场景，但是该模型是在假定流量平均分布得出的。业务场景的流量往往是随机的，这样不符合真实情况。

一般的经验，不同类型线程池的参数配置：

1. 计算密集型一般推荐线程池不要过大，一般是CPU数 + 1，+1是因为可能存在页缺失（就是可能存在有些数据在硬盘中需要多来一个线程将数据读入内存）。如果线程池数太大，可能会频繁的进行线程上下文切换跟任务调度。获得当前CPU核心数代码如下：

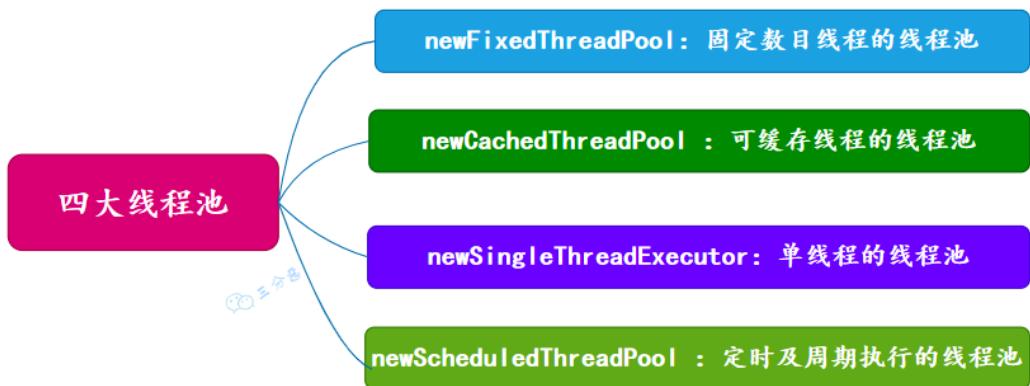
```
1 | Runtime.getRuntime().availableProcessors();
```

2. IO密集型：线程数适当大一点，机器的Cpu核心数*2。
3. 混合型：可以考虑根据情况将它拆分成CPU密集型和IO密集型任务，如果执行时间相差不大，拆分可以提升吞吐量，反之没有必要。

当然，实际应用中没有固定的公式，需要结合测试和监控来进行调整。

53.有哪几种常见的线程池？

面试常问，主要有四种，都是通过工具类Executors创建出来的，需要注意，阿里巴巴《Java开发手册》里禁止使用这种方式来创建线程池。



- `newFixedThreadPool` (固定数目线程的线程池)
- `newCachedThreadPool` (可缓存线程的线程池)
- `newSingleThreadExecutor` (单线程的线程池)
- `newScheduledThreadPool` (定时及周期执行的线程池)

54.能说一下四种常见线程池的原理吗？

前三种线程池的构造直接调用ThreadPoolExecutor的构造方法。

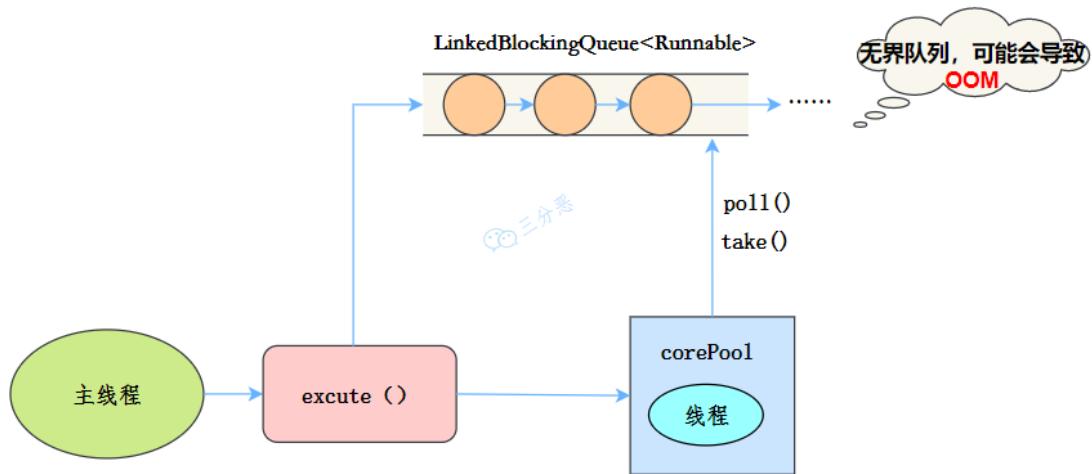
H5 `newSingleThreadExecutor`

```
1  public static ExecutorService
2      newSingleThreadExecutor(ThreadFactory threadFactory) {
3          return new FinalizableDelegatedExecutorService(
4              new ThreadPoolExecutor(1, 1,
5                  0L,
6                  TimeUnit.MILLISECONDS,
7                  new LinkedBlockingQueue<Runnable>(),
8                  threadFactory));
9  }
```

线程池特点

- 核心线程数为1
- 最大线程数也为1
- 阻塞队列是无界队列LinkedBlockingQueue，可能会导致OOM

- keepAliveTime为0



工作流程：

- 提交任务
- 线程池是否有一条线程在，如果没有，新建线程执行任务
- 如果有，将任务加到阻塞队列
- 当前的唯一线程，从队列取任务，执行完一个，再继续取，一个线程执行任务。

适用场景

适用于串行执行任务的场景，一个任务一个任务地执行。

H5 newFixedThreadPool

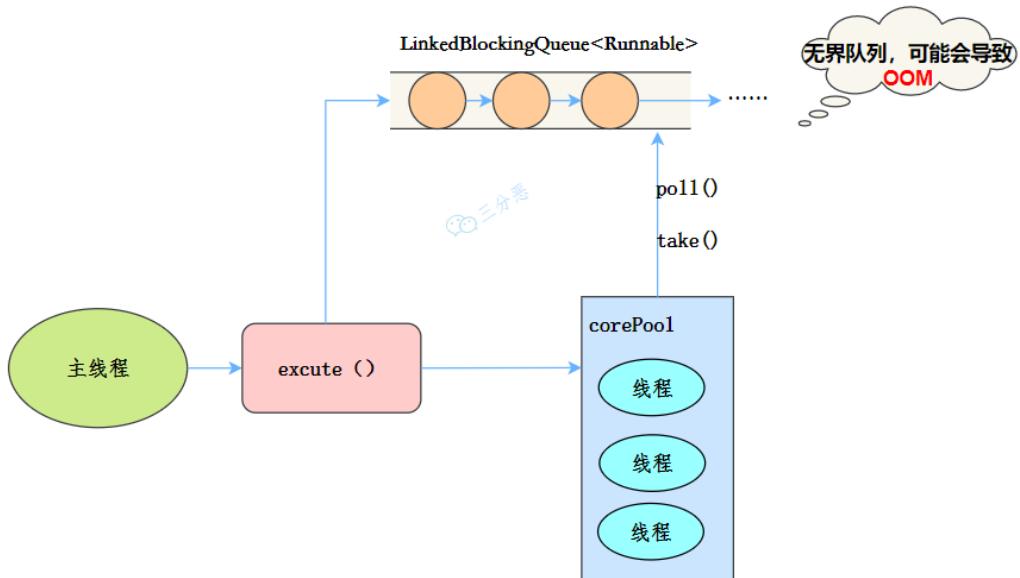
```

1  public static ExecutorService newFixedThreadPool(int
2      nThreads, ThreadFactory threadFactory) {
3          return new ThreadPoolExecutor(nThreads, nThreads,
4                                          0L,
5                                          TimeUnit.MILLISECONDS,
6                                          new
7                                          LinkedBlockingQueue<Runnable>(),
8                                          threadFactory);
9      }

```

线程池特点：

- 核心线程数和最大线程数大小一样
- 没有所谓的非空闲时间，即keepAliveTime为0
- 阻塞队列为无界队列LinkedBlockingQueue，可能会导致OOM



工作流程：

- 提交任务
- 如果线程数少于核心线程，创建核心线程执行任务
- 如果线程数等于核心线程，把任务添加到LinkedBlockingQueue阻塞队列
- 如果线程执行完任务，去阻塞队列取任务，继续执行。

使用场景

FixedThreadPool 适用于处理CPU密集型的任务，确保CPU在长期被工作线程使用的情况下，尽可能的少的分配线程，即适用执行长期的任务。

H5 newCachedThreadPool

```

1  public static ExecutorService
2      newCachedThreadPool(ThreadFactory threadFactory) {
3          return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
4                                         60L, TimeUnit.SECONDS,
5                                         new
6                                         SynchronousQueue<Runnable>(),
7                                         threadFactory);
8      }

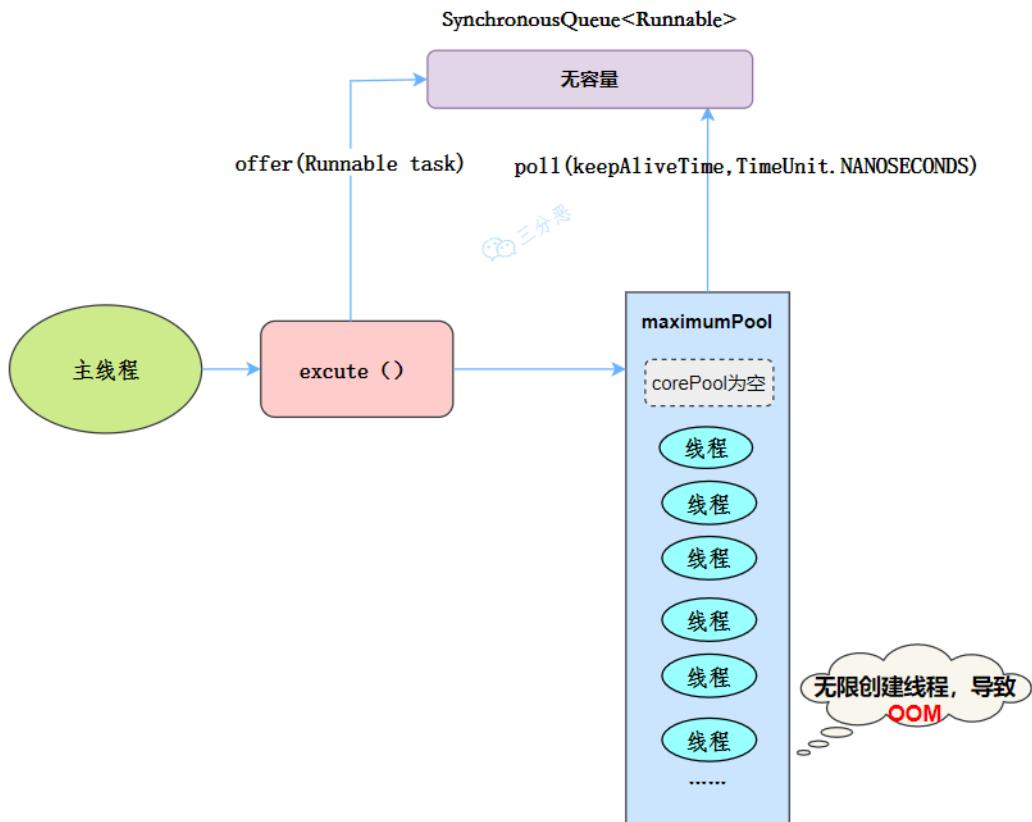
```

线程池特点：

- 核心线程数为0
- 最大线程数为Integer.MAX_VALUE，即无限大，可能会因为无限创建线程，导致OOM
- 阻塞队列是SynchronousQueue

- 非核心线程空闲存活时间为60秒

当提交任务的速度大于处理任务的速度时，每次提交一个任务，就必然会创建一个线程。极端情况下会创建过多的线程，耗尽 CPU 和内存资源。由于空闲 60 秒的线程会被终止，长时间保持空闲的 CachedThreadPool 不会占用任何资源。



工作流程：

- 提交任务
- 因为没有核心线程，所以任务直接加到SynchronousQueue队列。
- 判断是否有空闲线程，如果有，就去取出任务执行。
- 如果没有空闲线程，就新建一个线程执行。
- 执行完任务的线程，还可以存活60秒，如果在这期间，接到任务，可以继续活下去；否则，被销毁。

适用场景

用于并发执行大量短期的小任务。

H5 newScheduledThreadPool

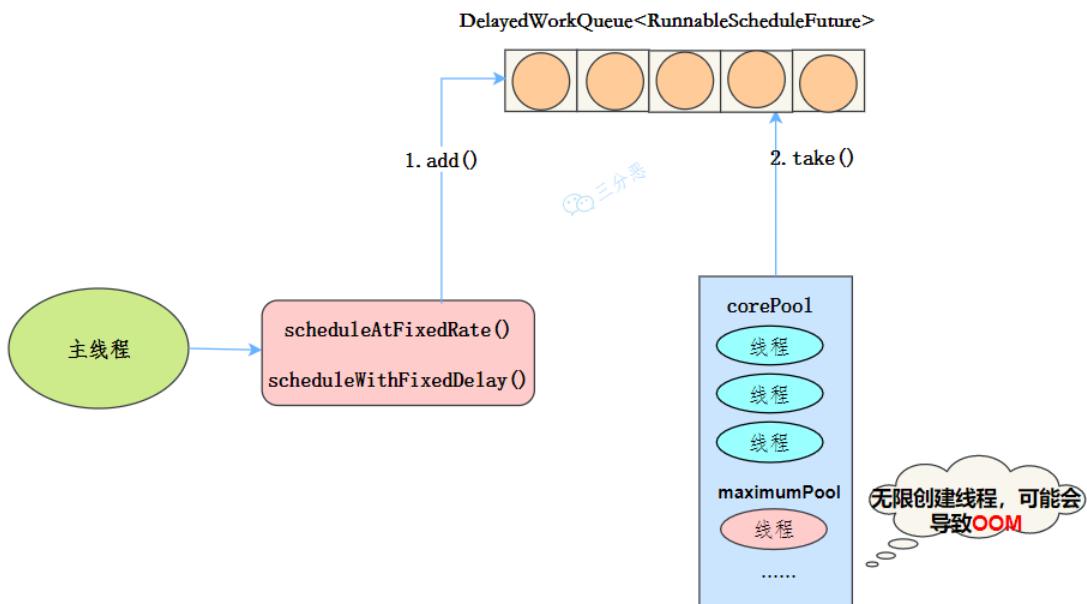
```

1     public ScheduledThreadPoolExecutor(int corePoolSize) {
2         super(corePoolSize, Integer.MAX_VALUE, 0,
3             NANOSECONDS,
4             new DelayedWorkQueue());

```

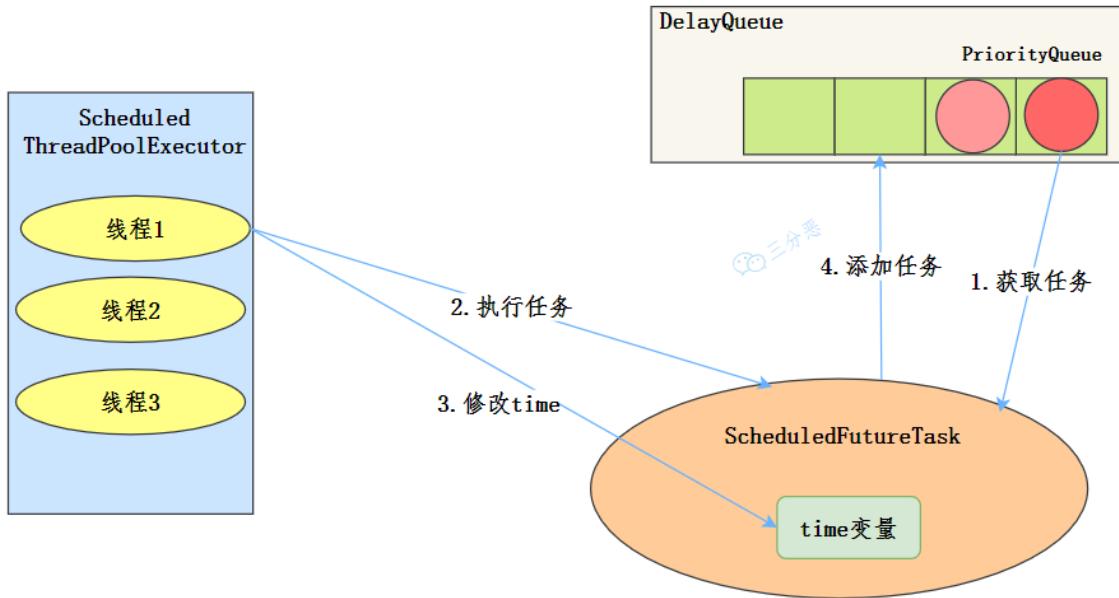
线程池特点

- 最大线程数为Integer.MAX_VALUE，也有OOM的风险
- 阻塞队列是DelayedWorkQueue
- keepAliveTime为0
- scheduleAtFixedRate()：按某种速率周期执行
- scheduleWithFixedDelay(): 在某个延迟后执行



工作机制

- 线程从DelayQueue中获取已到期的ScheduledFutureTask（DelayQueue.take()）。
到期任务是指ScheduledFutureTask的time大于等于当前时间。
- 线程执行这个ScheduledFutureTask。
- 线程修改ScheduledFutureTask的time变量为下次将要被执行的时间。
- 线程把这个修改time之后的ScheduledFutureTask放回DelayQueue中
(DelayQueue.add())。



使用场景

周期性执行任务的场景，需要限制线程数量的场景

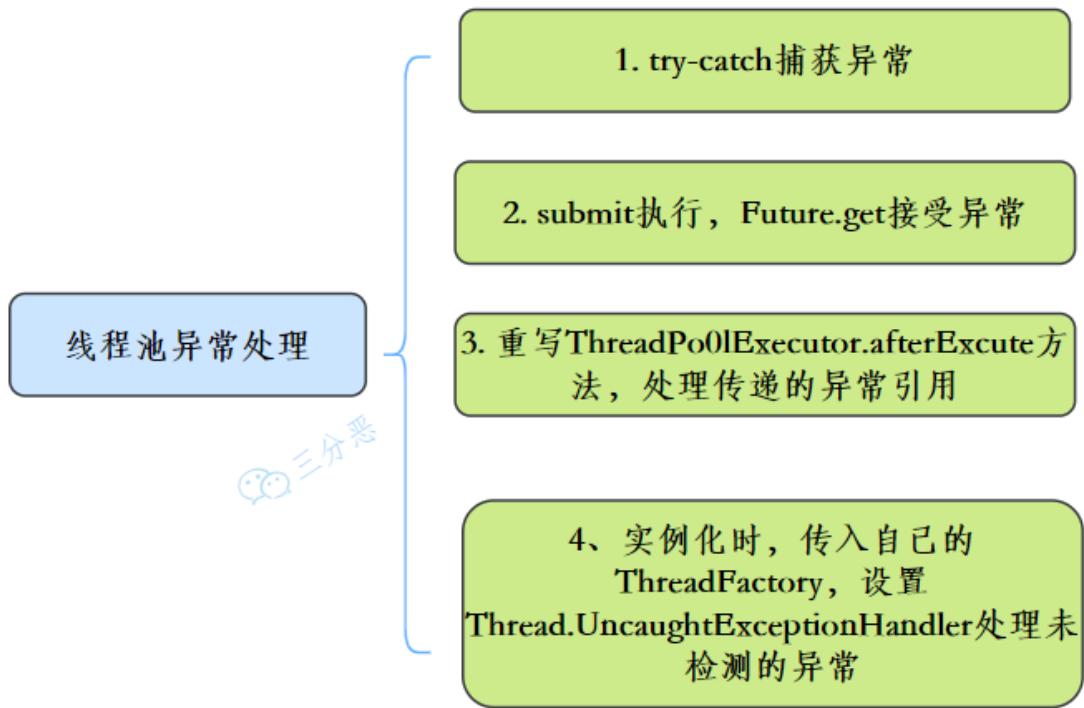
使用无界队列的线程池会导致什么问题吗？

例如`newFixedThreadPool`使用了无界的阻塞队列`LinkedBlockingQueue`，如果线程获取一个任务后，任务的执行时间比较长，会导致队列的任务越积越多，导致机器内存使用不停飙升，最终导致OOM。

55.线程池异常怎么处理知道吗？

在使用线程池处理任务的时候，任务代码可能抛出`RuntimeException`，抛出异常后，线程池可能捕获它，也可能创建一个新的线程来代替异常的线程，我们可能无法感知任务出现了异常，因此我们需要考虑线程池异常情况。

常见的异常处理方式：



56.能说一下线程池有几种状态吗？

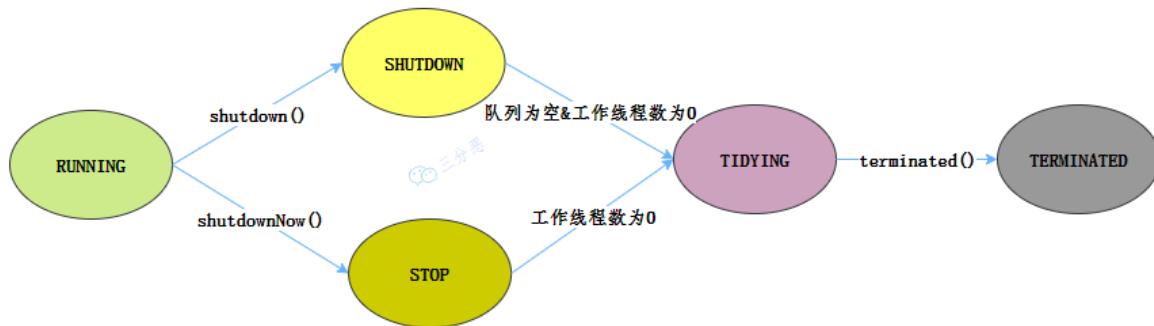
线程池有这几个状态：RUNNING,SHUTDOWN,STOP,TIDYING,TERMINATED。

```

1 //线程池状态
2 private static final int RUNNING      = -1 << COUNT_BITS;
3 private static final int SHUTDOWN     =  0 << COUNT_BITS;
4 private static final int STOP         =  1 << COUNT_BITS;
5 private static final int TIDYING      =  2 << COUNT_BITS;
6 private static final int TERMINATED   =  3 << COUNT_BITS;

```

线程池各个状态切换图：



RUNNING

- 该状态的线程池会接收新任务，并处理阻塞队列中的任务；
- 调用线程池的shutdown()方法，可以切换到SHUTDOWN状态；
- 调用线程池的shutdownNow()方法，可以切换到STOP状态；

SHUTDOWN

- 该状态的线程池不会接收新任务，但会处理阻塞队列中的任务；
- 队列为空，并且线程池中执行的任务也为空，进入TIDYING状态；

STOP

- 该状态的线程不会接收新任务，也不会处理阻塞队列中的任务，而且会中断正在运行的任务；
- 线程池中执行的任务为空，进入TIDYING状态；

TIDYING

- 该状态表明所有的任务已经运行终止，记录的任务数量为0。
- terminated()执行完毕，进入TERMINATED状态

TERMINATED

- 该状态表示线程池彻底终止

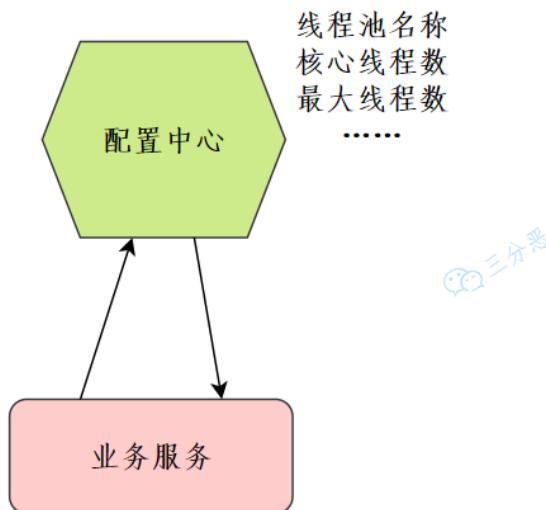
57.线程池如何实现参数的动态修改？

线程池提供了几个 setter方法来设置线程池的参数。

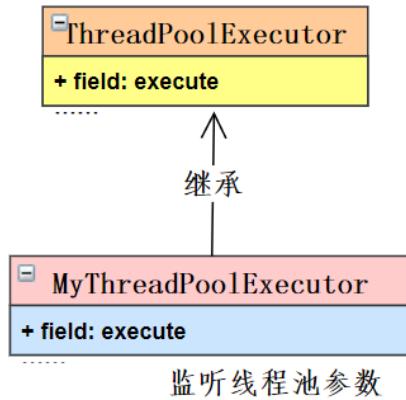
```
▼ CThreadPoolExecutor
  m setCorePoolSize(int): void
  m setKeepAliveTime(long, TimeUnit): void
  m setMaximumPoolSize(int): void
  m setRejectedExecutionHandler(RejectedExecutionHandler): void
  m setThreadFactory(ThreadFactory): void
  f workers: HashSet<Worker> = new HashSet<Worker>()
```

这里主要有两个思路：

①利用配置中心配置线程池参数



②自己实现线程池，监听参数变化



- 在我们微服务的架构下，可以利用配置中心如Nacos、Apollo等等，也可以自己开发配置中心。业务服务读取线程池配置，获取相应的线程池实例来修改线程池的参数。
- 如果限制了配置中心的使用，也可以自己去扩展 ThreadPoolExecutor，重写方法，监听线程池参数变化，来动态修改线程池参数。

58.线程池调优了解吗？

线程池配置没有固定的公式，通常事前会对线程池进行一定评估，常见的评估方案如下：

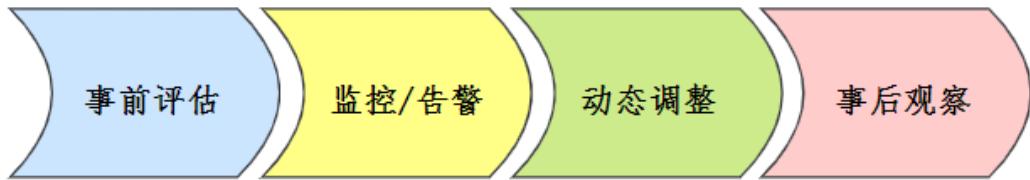
方案	问题
$N_{cpu} = \text{number of CPUs}$ $U_{cpu} = \text{target CPU utilization}, 0 \leq U_{cpu} \leq 1$ $\frac{W}{C} = \text{ratio of wait time to compute time}$ <i>The optimal pool size for keeping the processors at the desired utilization is :</i> $N_{threads} = N_{cpu} * U_{cpu} * (1 + \frac{W}{C})$	出自《Java并发编程实践》 该方案偏理论化。首先，线程计算的时间和等待的时间要如何确定呢？这个在实际开发中很难得到确切的值。另外计算出来的线程个数逼近线程实体的个数，Java线程池可以利用线程切换的方式最大程度利用CPU核数，这样计算出来的结果是非常偏离业务场景的。
$coreSize = 2 * N_{cpu}$ $maxSize = 25 * N_{cpu}$	没有考虑应用中往往使用多个线程池的情况，统一的配置明显不符合多样的业务场景。
$coreSize = tps * time$ $maxSize = tps * time * (1.7 - 2)$	这种计算方式，考虑到了业务场景，但是该模型是在假定流量平均分布得出的。业务场景的流量往往是随机的，这样不符合真实情况。

上线之前也要进行充分的测试，上线之后要建立完善的线程池监控机制。

事中结合监控告警机制，分析线程池的问题，或者可优化点，结合线程池动态参数配置机制来调整配置。

事后要注意仔细观察，随时调整。

线程池调优

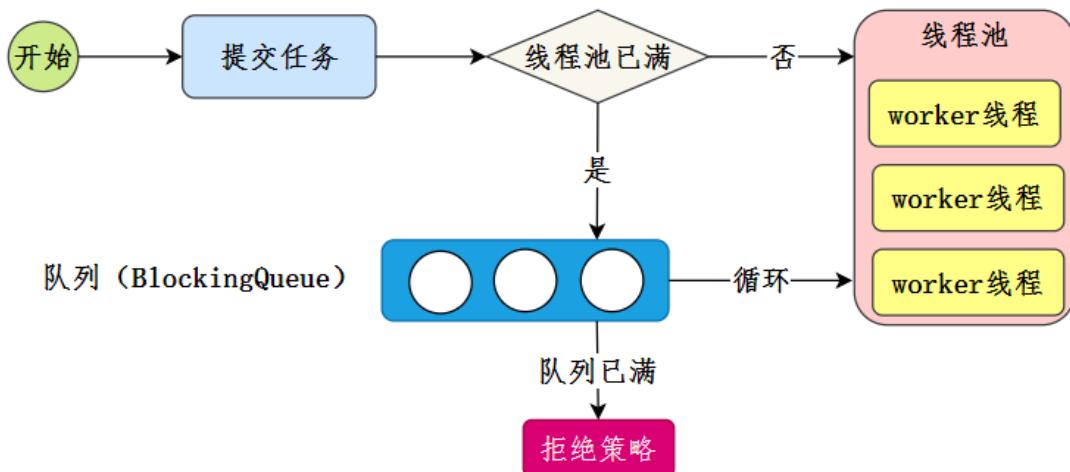


具体的调优案例可以查看参考[7]美团技术博客。

59. 你能设计实现一个线程池吗？

☆这道题在阿里的面试中出现频率比较高

线程池实现原理可以查看 [要是以前有人这么讲线程池，我早就该明白了！](#)，当然，我们自己实现，只需要抓住线程池的核心流程-参考[6]:



我们自己的实现就是完成这个核心流程：

- 线程池中有N个工作线程
- 把任务提交给线程池运行
- 如果线程池已满，把任务放入队列
- 最后当有空闲时，获取队列中任务来执行

实现代码[6]:


```
        break;
    }
    task = null;
}
} finally {
    //工作线程数增加
    ctl.decrementAndGet();
}
}

/**
 * 从队列中获取任务
 *
 * @return
 */
private Runnable getTask() {
    for ( ; ; ) {
        try {
            System.out.println("workQueue size:" + workQueue.size());
            return workQueue.take();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

//测试
public static void main(String[] args) {
    MyThreadPoolExecutor myThreadPoolExecutor = new MyThreadPoolExecutor(2, 2,
        new ArrayBlockingQueue<Runnable>(10));
    for (int i = 0; i < 10; i++) {
        int taskNum = i;
        myThreadPoolExecutor.execute(() -> {
            try {
                Thread.sleep(1500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("任务编号: " + taskNum);
        });
    }
}
```

这样，一个实现了线程池主要流程的类就完成了。

60.单机线程池执行断电了应该怎么处理？

我们可以对正在处理和阻塞队列的任务做事务管理或者对阻塞队列中的任务持久化处理，并且当断电或者系统崩溃，操作无法继续下去的时候，可以通过回溯日志的方式来撤销 **正在处理** 的已经执行成功的操作。然后重新执行整个阻塞队列。

也就是说，对阻塞队列持久化；正在处理任务事务控制；断电之后正在处理任务的回滚，通过日志恢复该次操作；服务器重启后阻塞队列中的数据再加载。

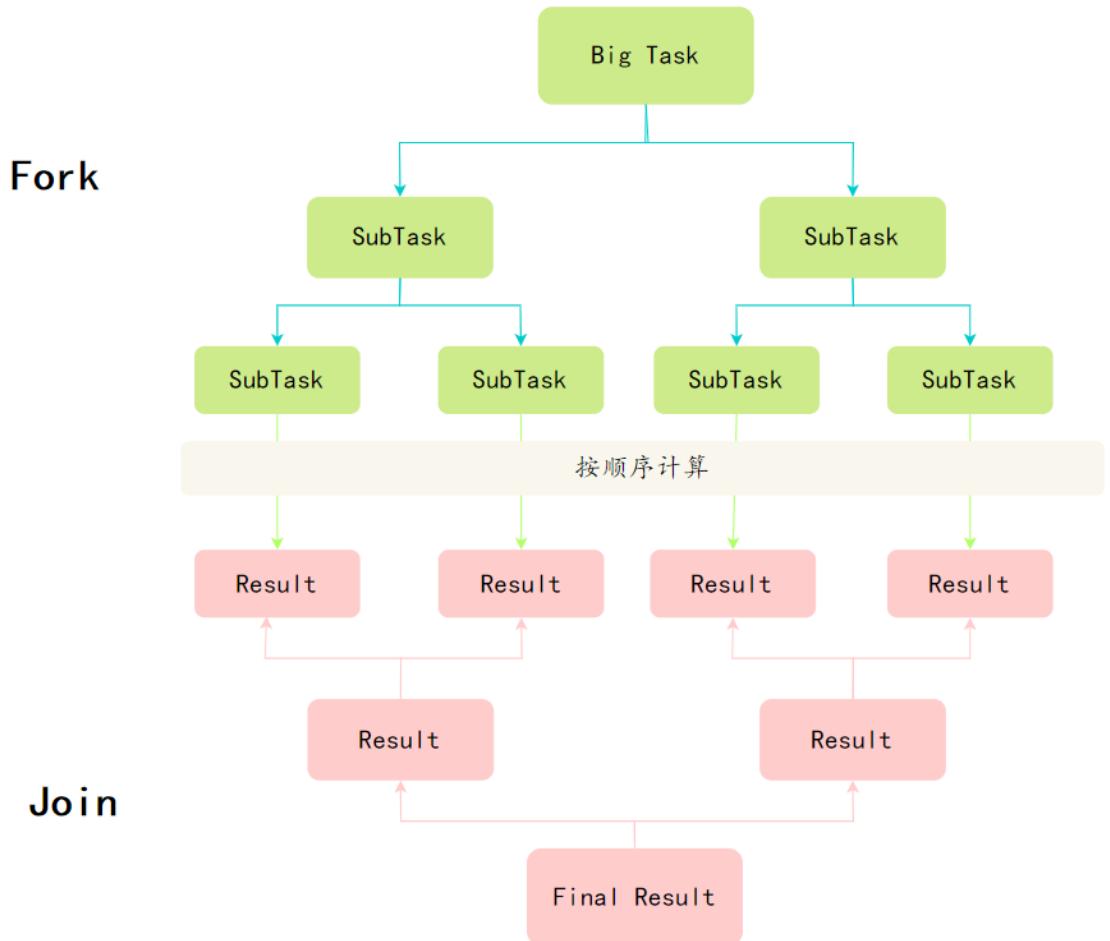
61.Fork/Join框架了解吗？

Fork/Join框架是Java7提供的一个用于并行执行任务的框架，是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

要想掌握Fork/Join框架，首先需要理解两个点，**分而治之** 和**工作窃取算法**。

分而治之

Fork/Join框架的定义，其实就体现了分治思想：将一个规模为N的问题分解为K个规模较小的子问题，这些子问题相互独立且与原问题性质相同。求出子问题的解，就可得到原问题的解。

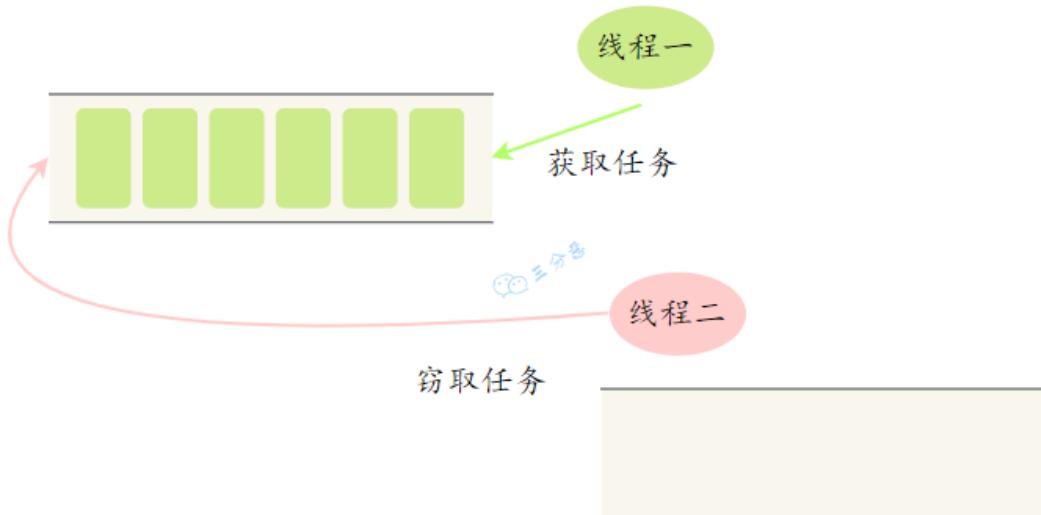


工作窃取算法

大任务拆成了若干个小任务，把这些小任务放到不同的队列里，各自创建单独线程来执行队列里的任务。

那么问题来了，有的线程干活快，有的线程干活慢。干完活的线程不能让它空下来，得让它去帮没干完活的线程干活。它去其它线程的队列里窃取一个任务来执行，这就是所谓的**工作窃取**。

工作窃取发生的时候，它们会访问同一个队列，为了减少窃取任务线程和被窃取任务线程之间的竞争，通常任务会使用双端队列，被窃取任务线程永远从双端队列的头部拿，而窃取任务的线程永远从双端队列的尾部拿任务执行。



看一个Fork/Join框架应用的例子，计算1~n之间的和： $1+2+3+\dots+n$

- 设置一个分割阈值，任务大于阈值就拆分任务
- 任务有结果，所以需要继承RecursiveTask

```

1  public class CountTask extends RecursiveTask<Integer> {
2      private static final int THRESHOLD = 16; // 阈值
3      private int start;
4      private int end;
5
6      public CountTask(int start, int end) {
7          this.start = start;
8          this.end = end;
9      }
10
11     @Override
12     protected Integer compute() {
13         int sum = 0;
14         // 如果任务足够小就计算任务
15         boolean canCompute = (end - start) <= THRESHOLD;
16         if (canCompute) {
17             for (int i = start; i <= end; i++) {
18                 sum += i;
19             }
20         } else {
21             // 如果任务大于阈值，就分裂成两个子任务计算
22             int middle = (start + end) / 2;
23             CountTask leftTask = new CountTask(start,
middle);

```

```

24     CountTask rightTask = new CountTask(middle + 1,
25     end);
26         // 执行子任务
27         leftTask.fork();
28         rightTask.fork(); // 等待子任务执行完，并得到其结果
29         int leftResult = leftTask.join();
30         int rightResult = rightTask.join(); // 合并子任务
31         sum = leftResult + rightResult;
32     }
33 }
34
35 public static void main(String[] args) {
36     ForkJoinPool forkJoinPool = new ForkJoinPool(); // 生
成一个计算任务，负责计算1+2+3+4
37     CountTask task = new CountTask(1, 100); // 执行一个任务
38     Future<Integer> result = forkJoinPool.submit(task);
39     try {
40         System.out.println(result.get());
41     } catch (InterruptedException e) {
42     } catch (ExecutionException e) {
43     }
44 }
45
46 }

```

ForkJoinTask与一般Task的主要区别在于它需要实现compute方法，在这个方法里，首先需要判断任务是否足够小，如果足够小就直接执行任务。如果比较大，就必须分割成两个子任务，每个子任务在调用fork方法时，又会进compute方法，看看当前子任务是否需要继续分割成子任务，如果不需要继续分割，则执行当前子任务并返回结果。使用join方法会等待子任务执行完并得到其结果。

参考：

- [1]. 《Java并发编程的艺术》
- [2]. 《Java并发编程实战》
- [3]. 个人珍藏的80道多线程并发面试题（1-10答案解析）：<https://juejin.cn/post/6854573221258199048>

[4]. 艾小仙 《我想进大厂》

[5]. Java并发基础知识，我用思维导图整理好了：<https://fighter3.blog.csdn.net/article/details/113612422>

[6] . 极客时间 《Java并发编程实战》

[7]. 《Java并发编程之美》

[8]. 万字图文 | 聊一聊 ReentrantLock 和 AQS 那点事（看完不会你找我）：<https://juejin.cn/post/6896278031317663751>

[9]. 《深入理解Java虚拟机》

[10]. 如何实现阻塞队列：<https://juejin.cn/post/6977948393272246285>

[11]. 讲真 这次绝对让你轻松学习线程池：<https://mp.weixin.qq.com/s/dTMH1TdxjCKy5yotQ7u7cA>

[12]. 面试必备：Java线程池解析：<https://juejin.cn/post/6844903889678893063>

[13]. 面试官问：“在项目中用过多线程吗？”你就把这个案例讲给他听！：<https://juejin.cn/post/6936457087505399821>

[14]. 小傅哥 《Java面经手册》

[15]. Java线程池实现原理及其在美团业务中的实践：<https://tech.meituan.com/2020/04/02/java-pooling-pratice-in-meituan.html>

[16]. 面试官：小伙子，听说你看过ThreadLocal源码？（万字图文深度解析 ThreadLocal）：<https://juejin.cn/post/6844904151567040519>

[17]. 面试官问我什么是JMM：<https://zhuanlan.zhihu.com/p/258393139>

[18]. 《王者并发课》：<https://juejin.cn/column/6963590682602635294>

[19]. synchronized锁升级详细过程：<https://www.cnblogs.com/suixing123/p/13996479.html>

关注公众号：三分恶

手册更新动态
即刻送达



添加个人微信：ThirdFighter

技术交流
加大佬云集微信群



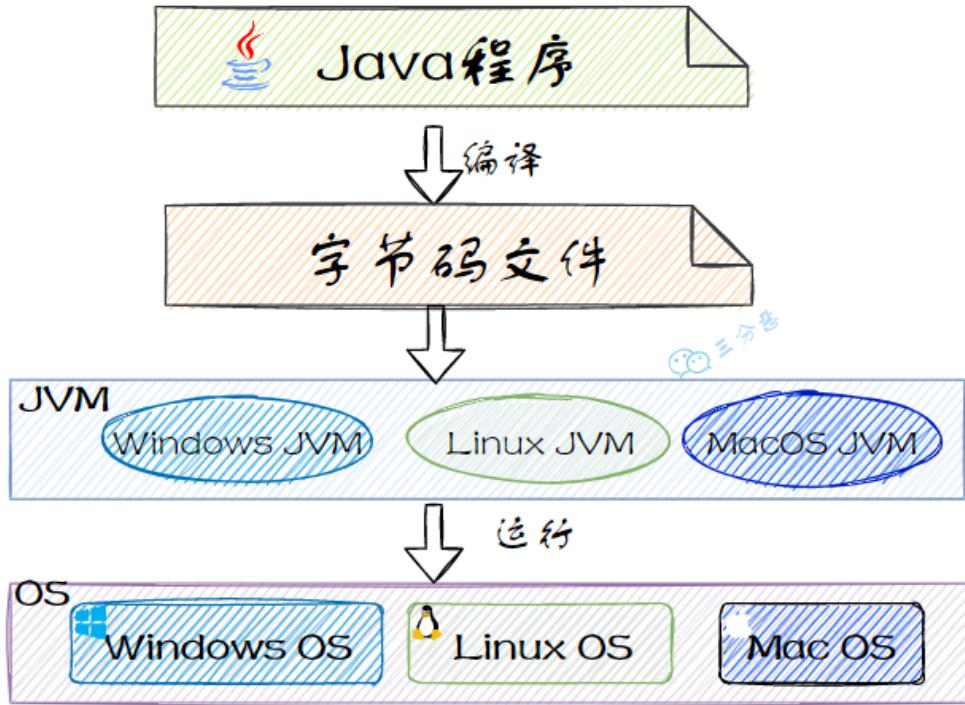
四、JVM

引言

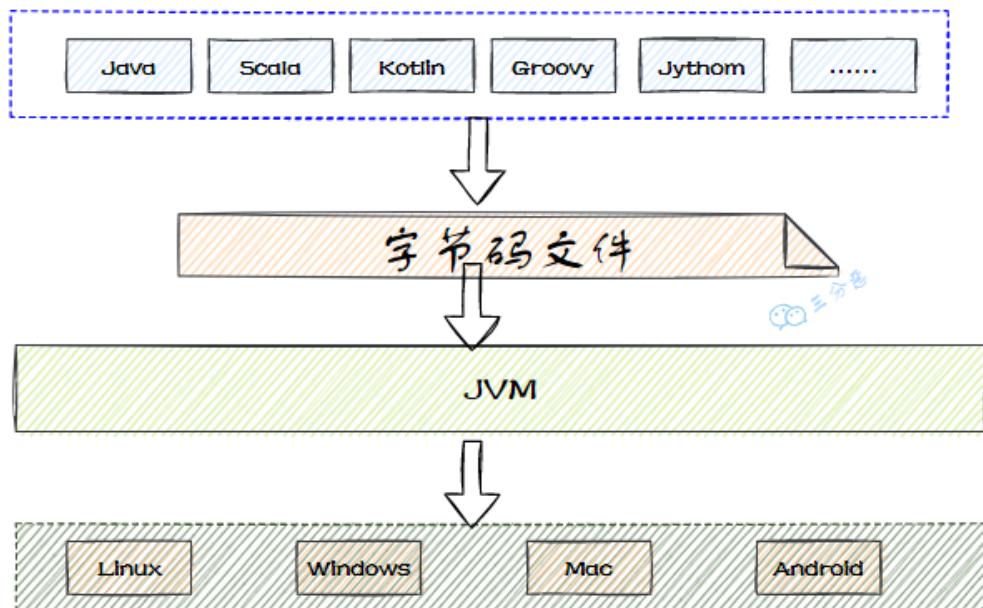
1.什么是JVM?

JVM——Java虚拟机，它是Java实现平台无关性的基石。

Java程序运行的时候，编译器将Java文件编译成平台无关的Java字节码文件（.class），接下来对应平台JVM对字节码文件进行解释，翻译成对应平台匹配的机器指令并运行。



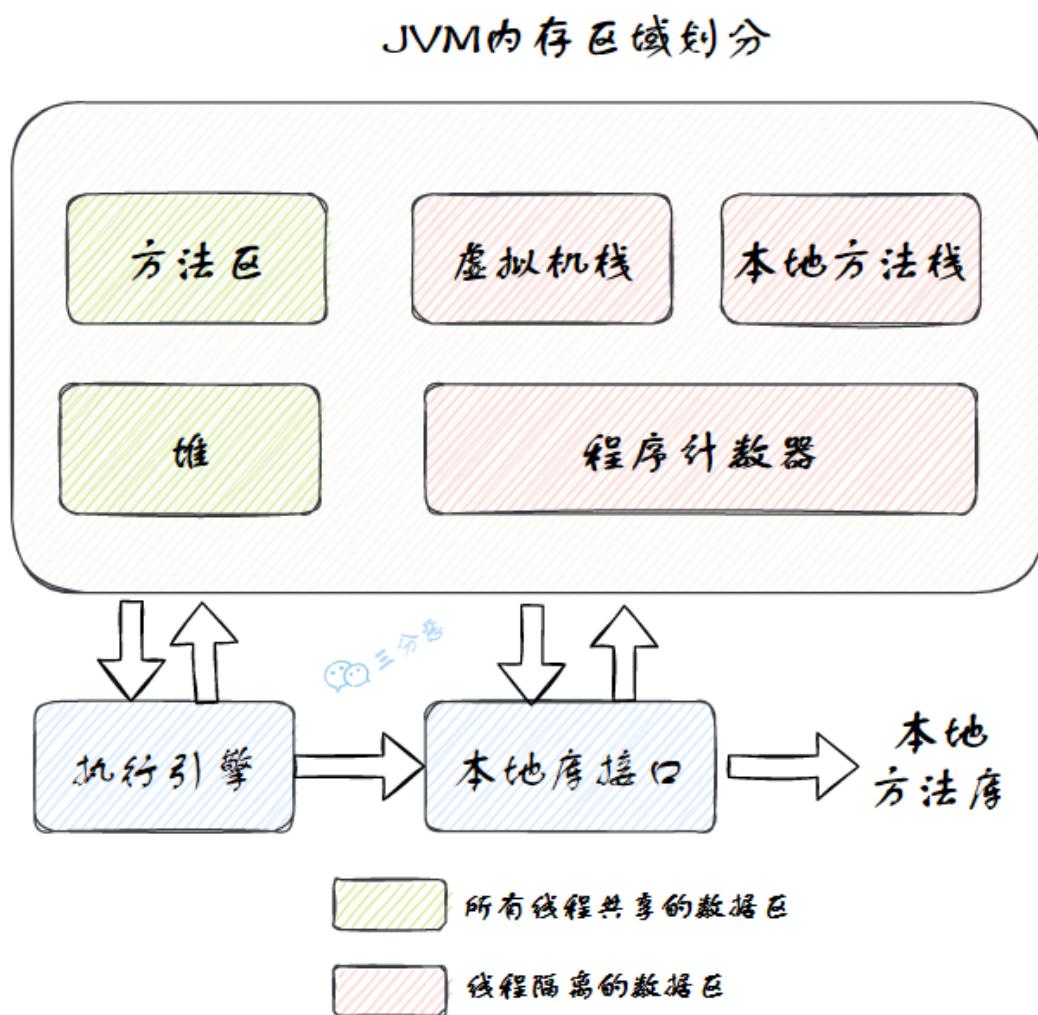
同时JVM也是一个跨语言的平台，和语言无关，只和class的文件格式关联，任何语言，只要能翻译成符合规范的字节码文件，都能被JVM运行。



内存管理

2.能说一下JVM的内存区域吗？

JVM内存区域最粗略的划分可以分为 **堆** 和 **栈**，当然，按照虚拟机规范，可以划分以下几个区域：



JVM内存分为线程私有区和线程共享区，其中 **方法区** 和 **堆** 是线程共享区， **虚拟机栈**、**本地方法栈** 和 **程序计数器** 是线程隔离的数据区。

1、程序计数器

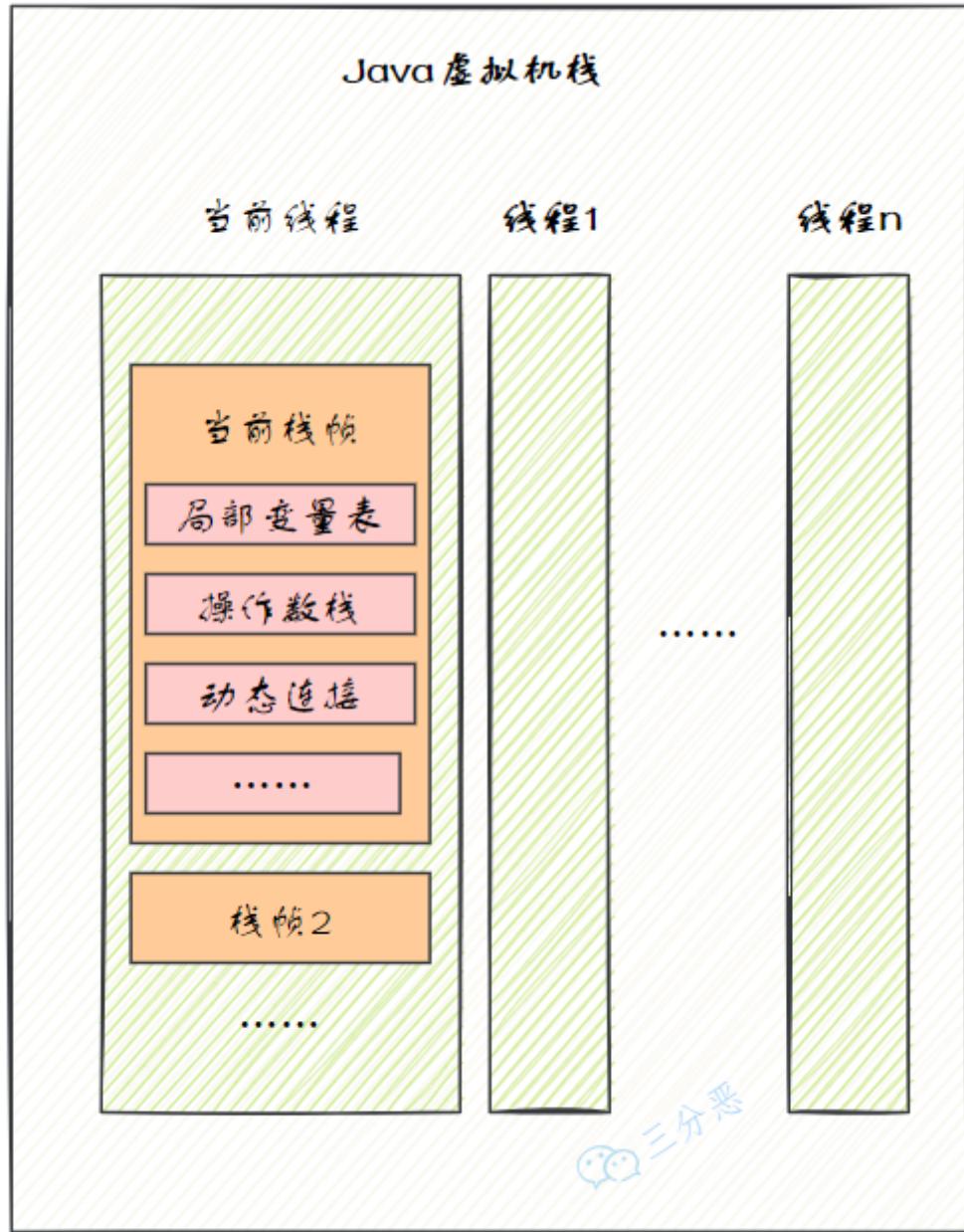
程序计数器（Program Counter Register）也被称为PC寄存器，是一块较小的内存空间。

它可以看作是当前线程所执行的字节码的行号指示器。

2、Java虚拟机栈

Java虚拟机栈（Java Virtual Machine Stack）也是线程私有的，它的生命周期与线程相同。

Java虚拟机栈描述的是Java方法执行的线程内存模型：方法执行时，JVM会同步创建一个栈帧，用来存储局部变量表、操作数栈、动态连接等。



3、本地方法栈

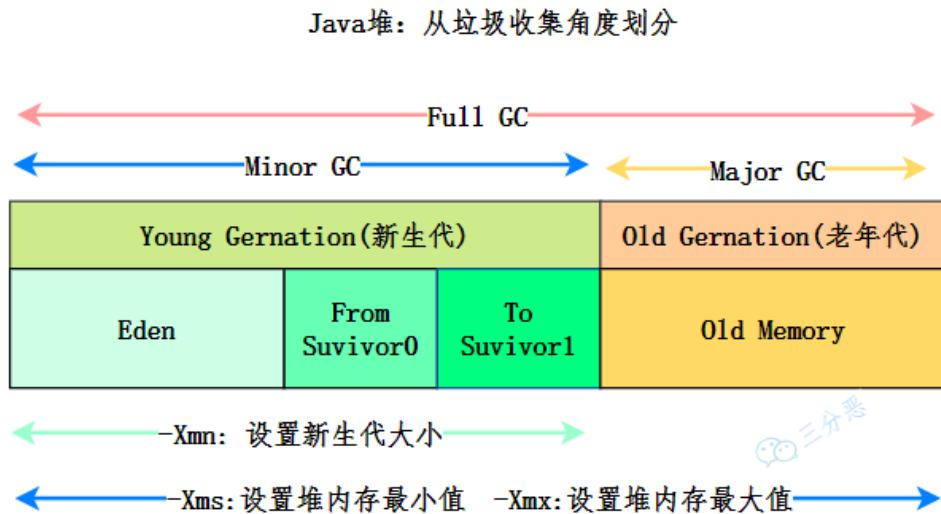
本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别只是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的本地（Native）方法服务。

Java 虚拟机规范允许本地方法栈被实现成固定大小的或者是根据计算动态扩展和收缩的。

4、Java堆

对于Java应用程序来说，Java堆（Java Heap）是虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，Java里“几乎”所有的对象实例都在这里分配内存。

Java堆是垃圾收集器管理的内存区域，因此一些资料中它也被称作“GC堆”（Garbage Collected Heap，）。从回收内存的角度看，由于现代垃圾收集器大部分都是基于分代收集理论设计的，所以Java堆中经常会出现 **新生代**、**老年年代**、**Eden空间**、**From Survivor空间**、**To Survivor空间** 等名词，需要注意的是这种划分只是根据垃圾回收机制来进行的划分，不是Java虚拟机规范本身制定的。



5.方法区

方法区是比较特别的一块区域，和堆类似，它也是各个线程共享的内存区域，用于存储已被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存等数据。

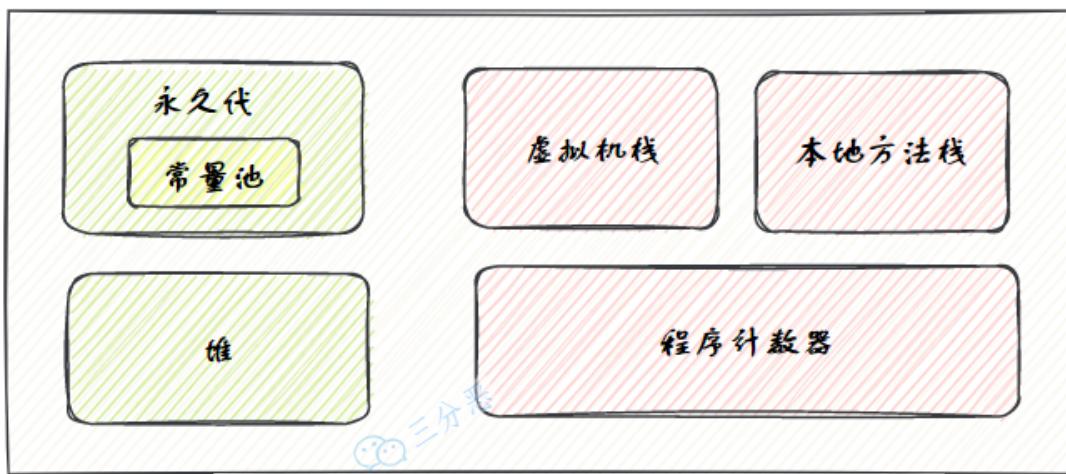
它特别在Java虚拟机规范对它的约束非常宽松，所以方法区的具体实现历经了许多变迁，例如jdk1.7之前使用永久代作为方法区的实现。

3.说一下JDK1.6、1.7、1.8内存区域的变化？

JDK1.6、1.7/1.8内存区域发生了变化，主要体现在方法区的实现：

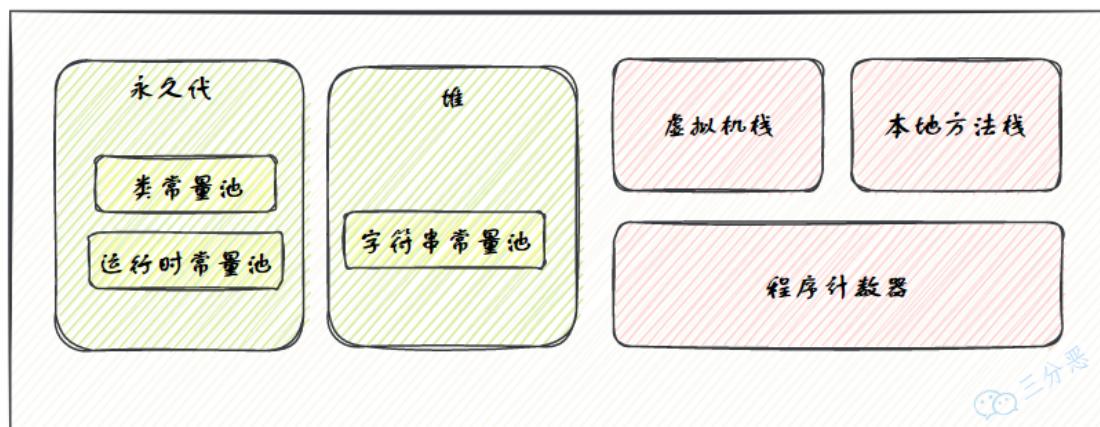
- JDK1.6使用永久代实现方法区：

JDK1.6

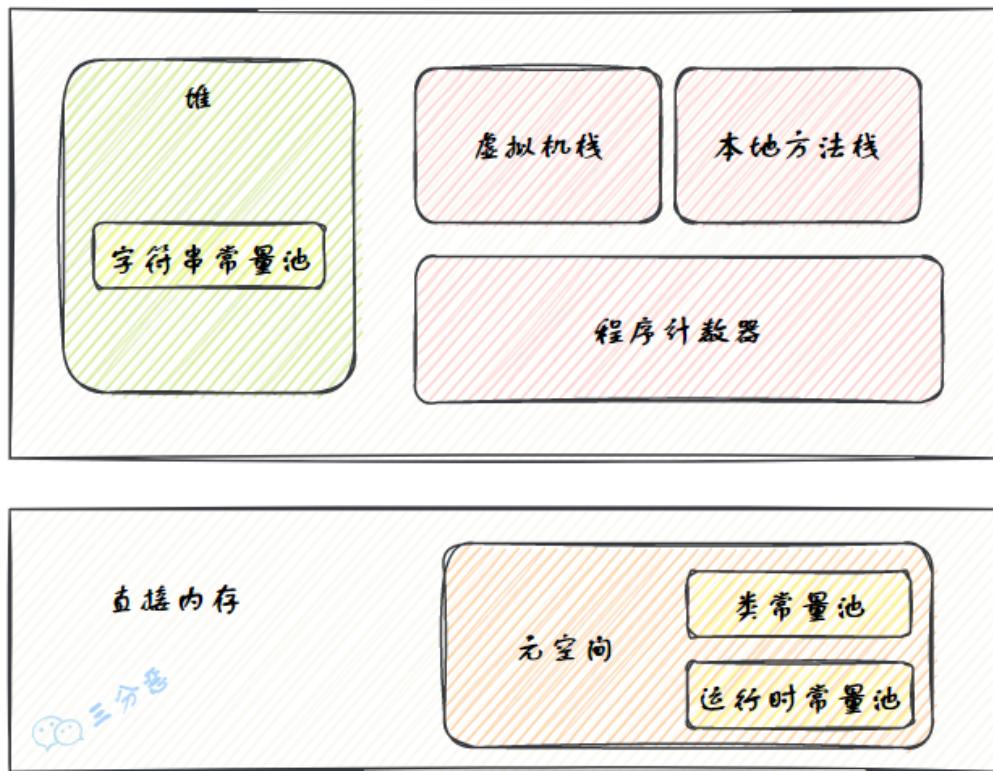


- JDK1.7时发生了一些变化，将字符串常量池、静态变量，存放在堆上

JDK1.7



- 在JDK1.8时彻底干掉了永久代，而在直接内存中划出一块区域作为**元空间**，运行时常量池、类常量池都移动到元空间。



4.为什么使用元空间替代永久代作为方法区的实现？

Java虚拟机规范规定的方法区只是换种方式实现。有客观和主观两个原因。

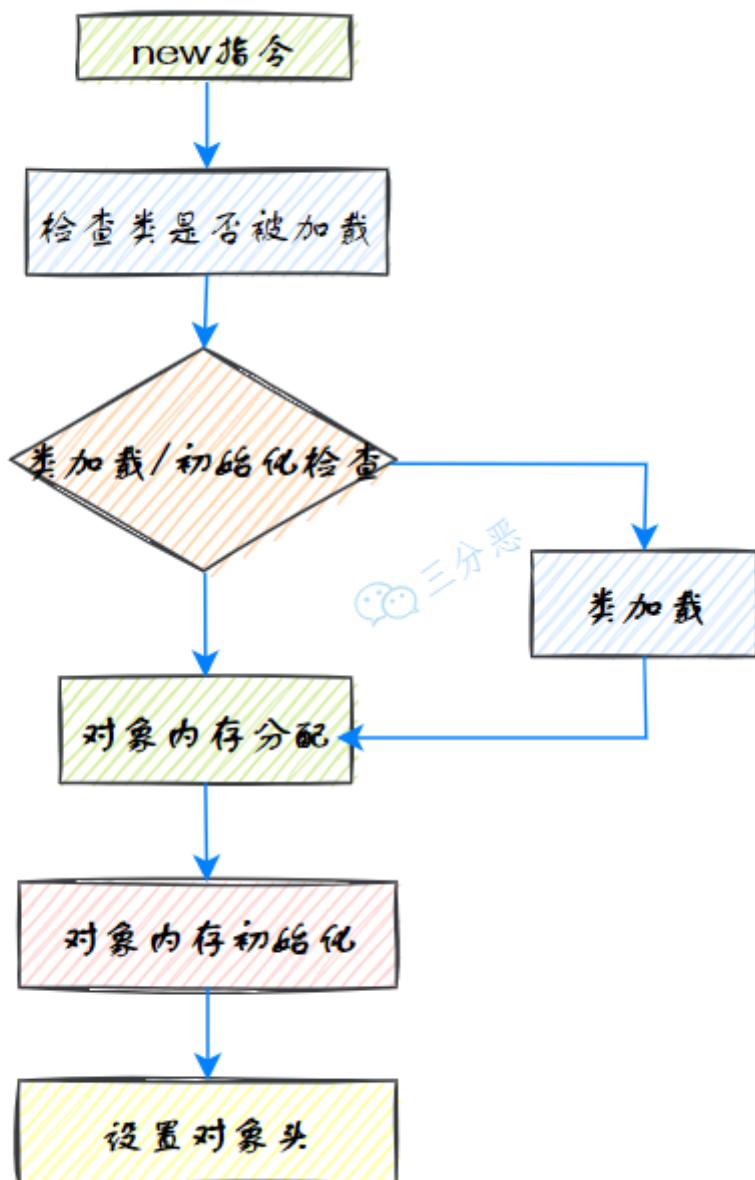
- 客观上使用永久代来实现方法区的决定的设计导致了Java应用更容易遇到内存溢出的问题（永久代有-XX: MaxPermSize的上限，即使不设置也有默认大小，而J9和JRockit只要没有触碰到进程可用内存的上限，例如32位系统中的4GB限制，就不会出问题），而且有极少数方法（例如String::intern()）会因永久代的原因而导致不同虚拟机下有不同的表现。
- 主观上当Oracle收购BEA获得了JRockit的所有权后，准备把JRockit中的优秀功能，譬如Java Mission Control管理工具，移植到HotSpot虚拟机时，但因为两者对方法区实现的差异而面临诸多困难。考虑到HotSpot未来的发展，在JDK 6的时候HotSpot开发团队就有放弃永久代，逐步改为采用本地内存（Native Memory）来实现方法区的计划了，到了JDK 7的HotSpot，已经把原本放在永久代的字符串常量池、静态变量等移出，而到了JDK 8，终于完全废弃了永久代的概念，改用与JRockit、J9一样在本地内存中实现的元空间（Meta-space）来代替，把JDK 7中永久代还剩余的内容（主要是类型信息）全部移到元空间中。

5.对象创建的过程了解吗？

在JVM中对象的创建，我们从一个new指令开始：

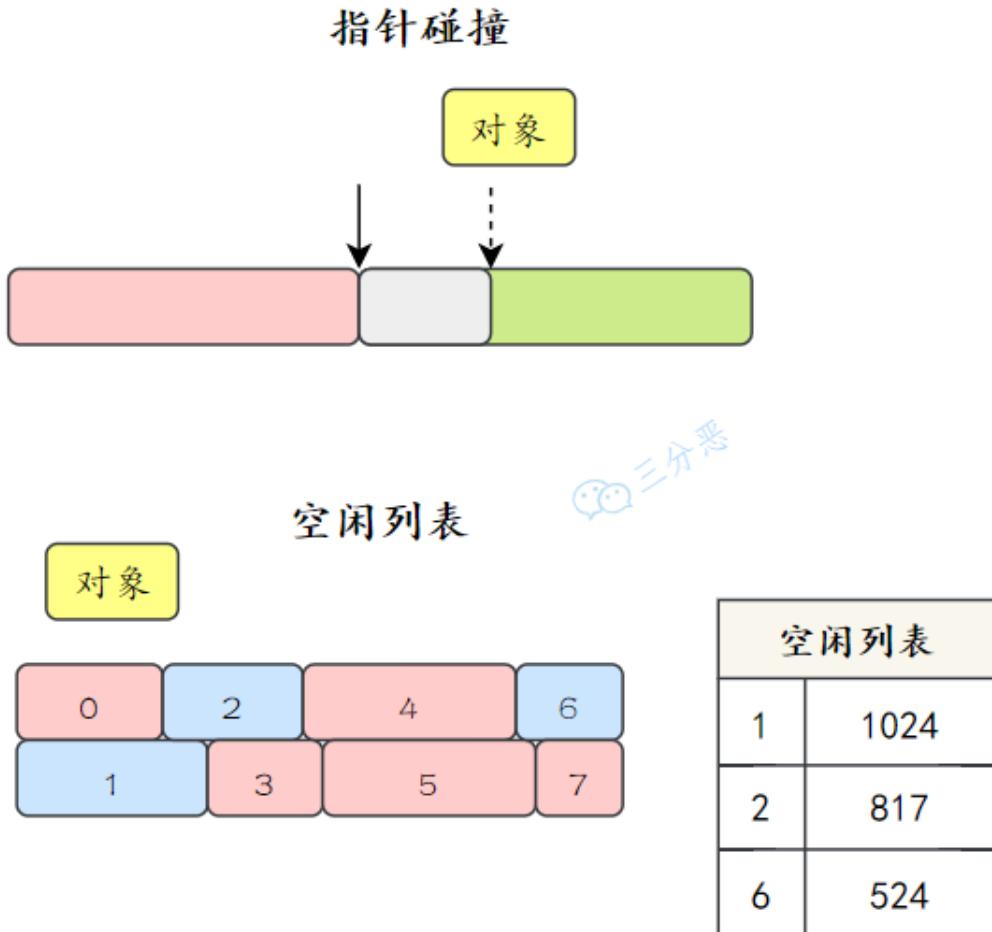
- 首先检查这个指令的参数是否能在常量池中定位到一个类的符号引用
- 检查这个符号引用代表的类是否已被加载、解析和初始化过。如果没有，就先执行相应的类加载过程
- 类加载检查通过后，接下来虚拟机将为新生对象分配内存。
- 内存分配完成之后，虚拟机将分配到的内存空间（但不包括对象头）都初始化为零值。
- 接下来设置对象头，请求头里包含了对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的GC分代年龄等信息。

这个过程大概图示如下：



6.什么是指针碰撞？什么是空闲列表？

内存分配有两种方式，**指针碰撞**（Bump The Pointer）、**空闲列表**（Free List）。



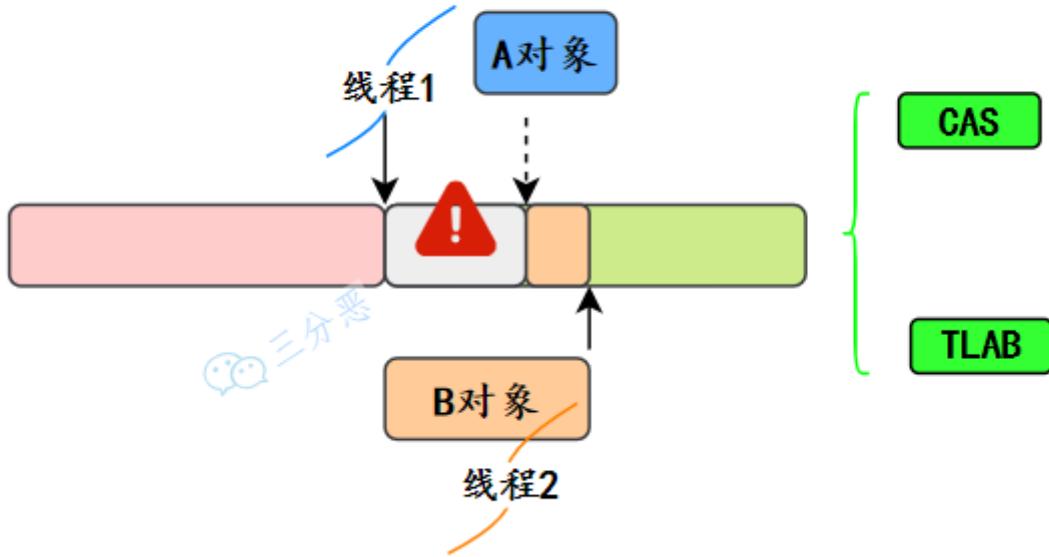
- 指针碰撞：假设Java堆中内存是绝对规整的，所有被使用过的内存都被放在一边，空闲的内存被放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间方向挪动一段与对象大小相等的距离，这种分配方式称为“指针碰撞”。
- 空闲列表：如果Java堆中的内存并不是规整的，已被使用的内存和空闲的内存相互交错在一起，那就没有办法简单地进行指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式称为“空闲列表”。
- 两种方式的选择由Java堆是否规整决定，Java堆是否规整是由选择的垃圾收集器是否具有压缩整理能力决定的。

7.JVM 里 new 对象时，堆会发生抢占吗？JVM 是怎么设计来保证线程安全的？

会，假设JVM虚拟机上，每一次new 对象时，指针就会向右移动一个对象size的距离，一个线程正在给A对象分配内存，指针还没有来的及修改，另一个为B对象分配内存的线程，又引用了这个指针来分配内存，这就发生了抢占。

有两种可选方案来解决这个问题：

堆抢占

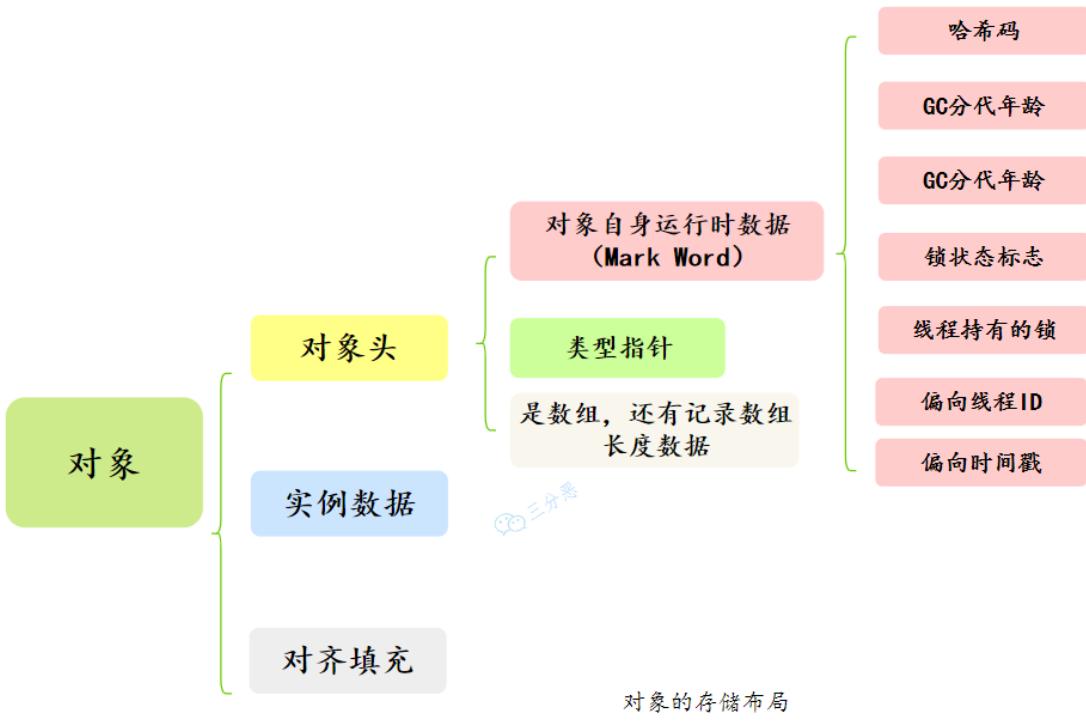


- 采用CAS分配重试的方式来保证更新操作的原子性
- 每个线程在Java堆中预先分配一小块内存，也就是本地线程分配缓冲（Thread Local Allocation

Buffer, TLAB），要分配内存的线程，先在本地缓冲区中分配，只有本地缓冲区用完了，分配新的缓存区时才需要同步锁定。

8.能说一下对象的内存布局吗？

在HotSpot虚拟机里，对象在堆内存中的存储布局可以划分为三个部分：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。



对象头主要由两部分组成：

- 第一部分存储对象自身的运行时数据：哈希码、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等，官方称它为Mark Word，它是个动态的结构，随着对象状态变化。
- 第二部分是类型指针，指向对象的类元数据类型（即对象代表哪个类）。
- 此外，如果对象是一个Java数组，那还应该有一块用于记录数组长度的数据

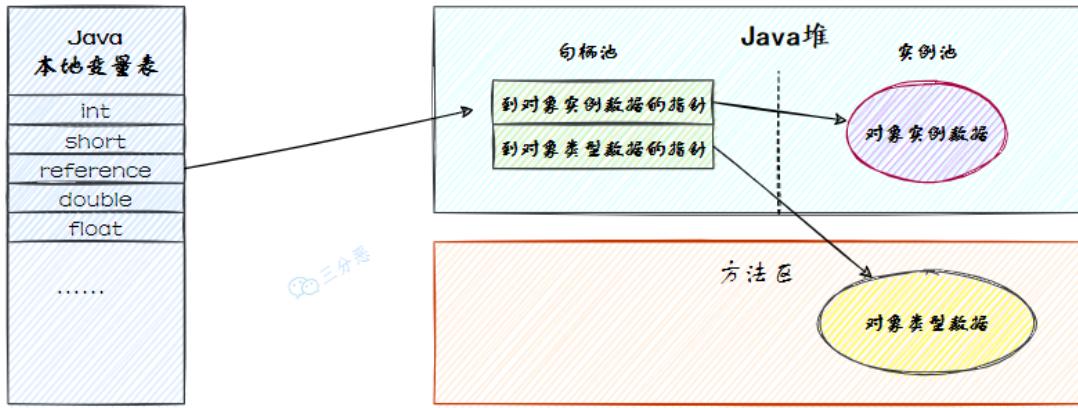
实例数据用来存储对象真正的有效信息，也就是我们在程序代码里所定义的各种类型的字段内容，无论是从父类继承的，还是自己定义的。

对齐填充不是必须的，没有特别含义，仅仅起着占位符的作用。

9. 对象怎么访问定位？

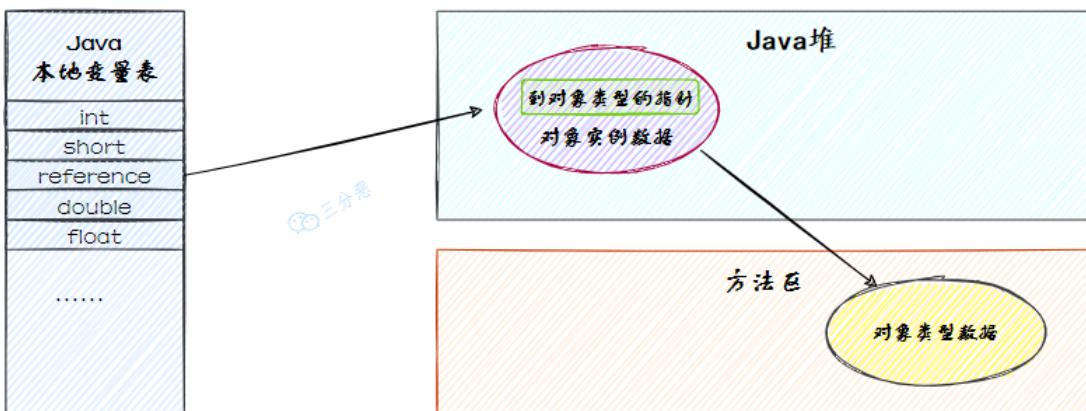
Java程序会通过栈上的reference数据来操作堆上的具体对象。由于reference类型在《Java虚拟机规范》里面只规定了它是一个指向对象的引用，并没有定义这个引用应该通过什么方式去定位、访问到堆中对象的具体位置，所以对象访问方式也是由虚拟机实现而定的，主流的访问方式主要有使用句柄和直接指针两种：

- 如果使用句柄访问的话，Java堆中将可能会划分出一块内存来作为句柄池，reference中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自具体的地址信息，其结构如图所示：



通过句柄访问对象

- 如果使用直接指针访问的话，Java堆中对象的内存布局就必须考虑如何放置访问类型数据的相关信息，reference中存储的直接就是对象地址，如果只是访问对象本身的话，就不需要多一次间接访问的开销，如图所示：



通过直接指针访问对象

这两种对象访问方式各有优势，使用句柄来访问的最大好处就是reference中存储的是稳定句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而reference本身不需要被修改。

使用直接指针来访问最大的好处就是速度更快，它节省了一次指针定位的时间开销，由于对象访问在Java中非常频繁，因此这类开销积少成多也是一项极为可观的执行成本。

HotSpot虚拟机主要使用直接指针来进行对象访问。

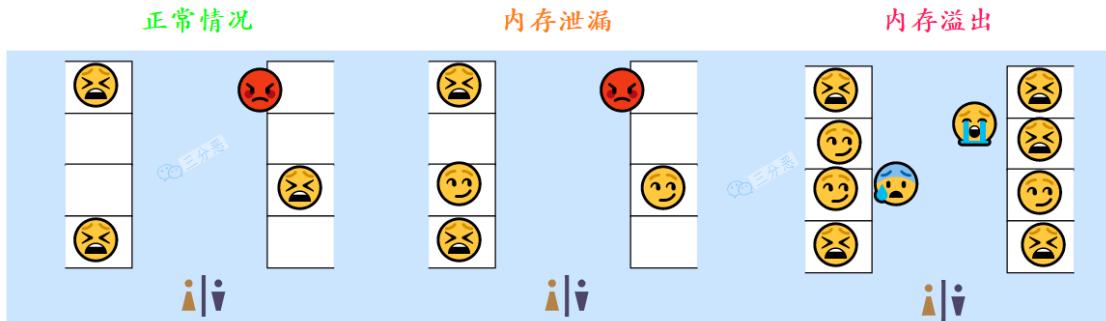
10. 内存溢出和内存泄漏是什么意思？

内存泄露就是申请的内存空间没有被正确释放，导致内存被白白占用。

内存溢出就是申请的内存超过了可用内存，内存不够了。

两者关系：内存泄露可能会导致内存溢出。

用一个有味道的比喻，内存溢出就是排队去蹲坑，发现没坑位了，内存泄漏，就是有人占着茅坑不拉屎，占着茅坑不拉屎的多了可能导致坑位不够用。



11.能手写内存溢出的例子吗？

在JVM的几个内存区域中，除了程序计数器外，其他几个运行时区域都有可能发生内存溢出（OOM）异常的可能，重点关注堆和栈。

- Java堆溢出

Java堆用于储存对象实例，只要不断创建不可被回收的对象，比如静态对象，那么随着对象数量的增加，总容量触及最大堆的容量限制后就会产生内存溢出异常（`OutOfMemoryError`）。

这就相当于一个房子里，不断堆积不能被收走的杂物，那么房子很快就会被堆满了。

```
1 /**
2  * VM参数: -Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError
3 */
4 public class HeapOOM {
5     static class OOMObject {
6     }
7
8     public static void main(String[] args) {
9         List<OOMObject> list = new ArrayList<OOMObject>();
10        while (true) {
11            list.add(new OOMObject());
12        }
13    }
14}
```

```
13     }
14 }
15 }
```

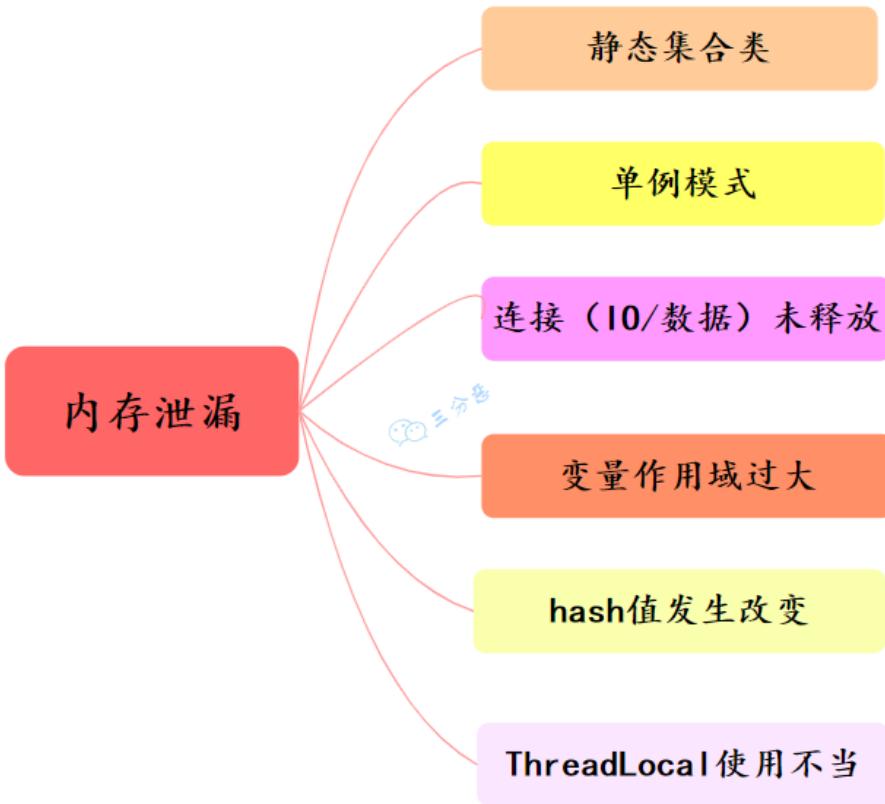
- 虚拟机栈.OutOfMemoryError

JDK使用的HotSpot虚拟机的栈内存大小是固定的，我们可以把栈的内存设大一点，然后不断地去创建线程，因为操作系统给每个进程分配的内存是有限的，所以到最后，也会发生OutOfMemoryError异常。

```
1  /**
2  * vm参数: -Xss2M
3  */
4  public class JavaVMStackOOM {
5      private void dontStop() {
6          while (true) {
7              }
8          }
9
10     public void stackLeakByThread() {
11         while (true) {
12             Thread thread = new Thread(new Runnable() {
13                 public void run() {
14                     dontStop();
15                 }
16             });
17             thread.start();
18         }
19     }
20
21     public static void main(String[] args) throws Throwable {
22         JavaVMStackOOM oom = new JavaVMStackOOM();
23         oom.stackLeakByThread();
24     }
25 }
26 }
```

12. 内存泄漏可能由哪些原因导致呢？

内存泄漏可能的原因有很多种：



静态集合类引起内存泄漏

静态集合的生命周期和 JVM 一致，所以静态集合引用的对象不能被释放。

```

1 | public class OOM {
2 |     static List list = new ArrayList();
3 |
4 |     public void oomTests(){
5 |         Object obj = new Object();
6 |
7 |         list.add(obj);
8 |     }
9 |
10}

```

单例模式

和上面的例子原理类似，单例对象在初始化后会以静态变量的方式在 JVM 的整个生命周期中存在。如果单例对象持有外部的引用，那么这个外部对象将不能被 GC 回收，导致内存泄漏。

数据连接、IO、Socket等连接

创建的连接不再使用时，需要调用 `close` 方法关闭连接，只有连接被关闭后，GC 才会回收对应的对象（Connection，Statement，ResultSet，Session）。忘记关闭这些资源会导致持续占有内存，无法被 GC 回收。

```
1   try {
2       Connection conn = null;
3       Class.forName("com.mysql.jdbc.Driver");
4       conn = DriverManager.getConnection("url", "", "");
5   }
6   Statement stmt = conn.createStatement();
7   ResultSet rs = stmt.executeQuery("....");
8
9 } finally {
10    //不关闭连接
11 }
12 }
```

变量不合理的作用域

一个变量的定义作用域大于其使用范围，很可能存在内存泄漏；或不再使用对象没有及时将对象设置为 `null`，很可能导致内存泄漏的发生。

```
1 public class Simple {
2     Object object;
3     public void method1(){
4         object = new Object();
5         //...其他代码
6         //由于作用域原因，method1执行完成之后，object 对象所分配的内
7         存不会马上释放
8         object = null;
9     }
10 }
```

hash值发生变化

对象Hash值改变，使用HashMap、HashSet等容器中时候，由于对象修改之后的Hash值和存储进容器时的Hash值不同，所以无法找到存入的对象，自然也无法单独删除了，这也会造成内存泄漏。说句题外话，这也是为什么String类型被设置成了不可变类型。

ThreadLocal使用不当

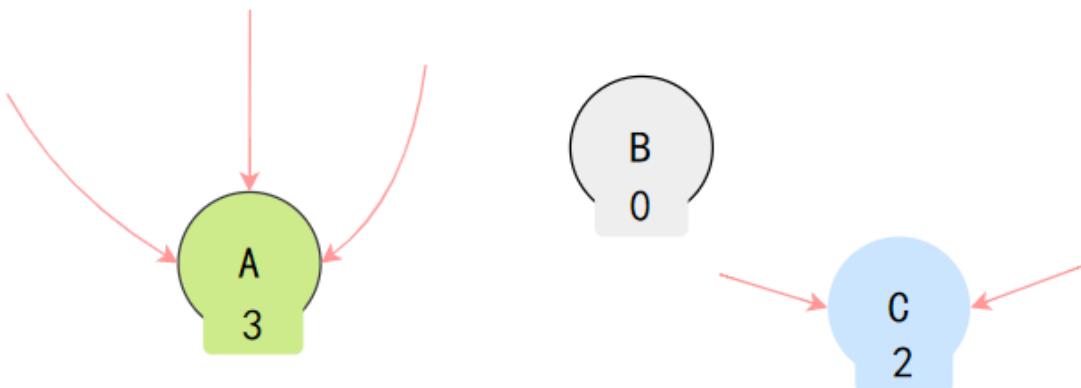
ThreadLocal的弱引用导致内存泄漏也是个老生常谈的话题了，使用完ThreadLocal一定要记得使用remove方法来进行清除。

13. 如何判断对象仍然存活？

有两种方式，**引用计数算法（reference counting）** 和可达性分析算法。

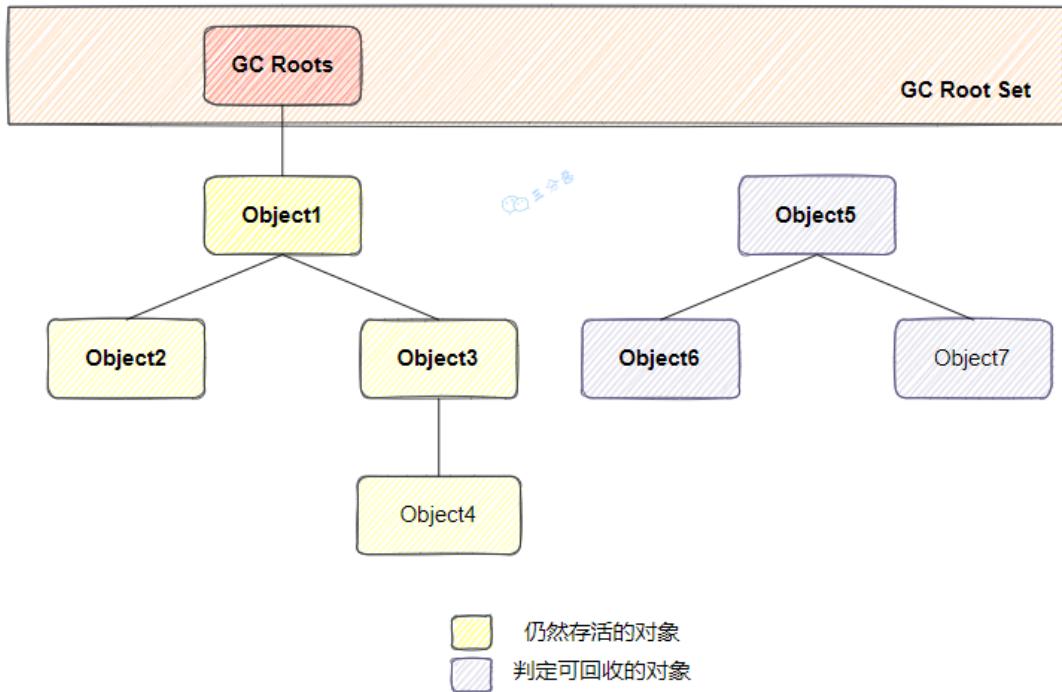
- 引用计数算法

引用计数器的算法是这样的：在对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加一；当引用失效时，计数器值就减一；任何时刻计数器为零的对象就是不可能再被使用的。



- 可达性分析算法

目前 Java 虚拟机的主流垃圾回收器采取的是可达性分析算法。这个算法的实质在于将一系列 GC Roots 作为初始的存活对象合集（Gc Root Set），然后从该合集出发，探索所有能够被该集合引用到的对象，并将其加入到该集合中，这个过程我们也称之为标记（mark）。最终，未被探索到的对象便是死亡的，是可以回收的。



14. Java中可作为GC Roots的对象有哪几种？

可以作为GC Roots的主要有四种对象：

- 虚拟机栈(栈帧中的本地变量表)中引用的对象
- 方法区中类静态属性引用的对象
- 方法区中常量引用的对象
- 本地方法栈中JNI引用的对象

15. 说一下对象有哪几种引用？

Java中的引用有四种，分为强引用（Strongly Reference）、软引用（Soft Reference）、弱引用（Weak Reference）和虚引用（Phantom Reference）4种，这4种引用强度依次逐渐减弱。

- 强引用是最传统的引用的定义，是指在程序代码之中普遍存在的引用赋值，无论任何情况下，只要强引用关系还存在，垃圾收集器就永远不会回收掉被引用的对象。

```
1 | Object obj =new Object();
```

- 软引用是用来描述一些还有用，但非必须的对象。只被软引用关联着的对象，在系统将要发生内存溢出异常前，会把这些对象列进回收范围之中进行第二次回收，如果这次回收还没有足够的内存，才会抛出内存溢出异常。在JDK 1.2版之后提供了SoftReference类来实现软引用。

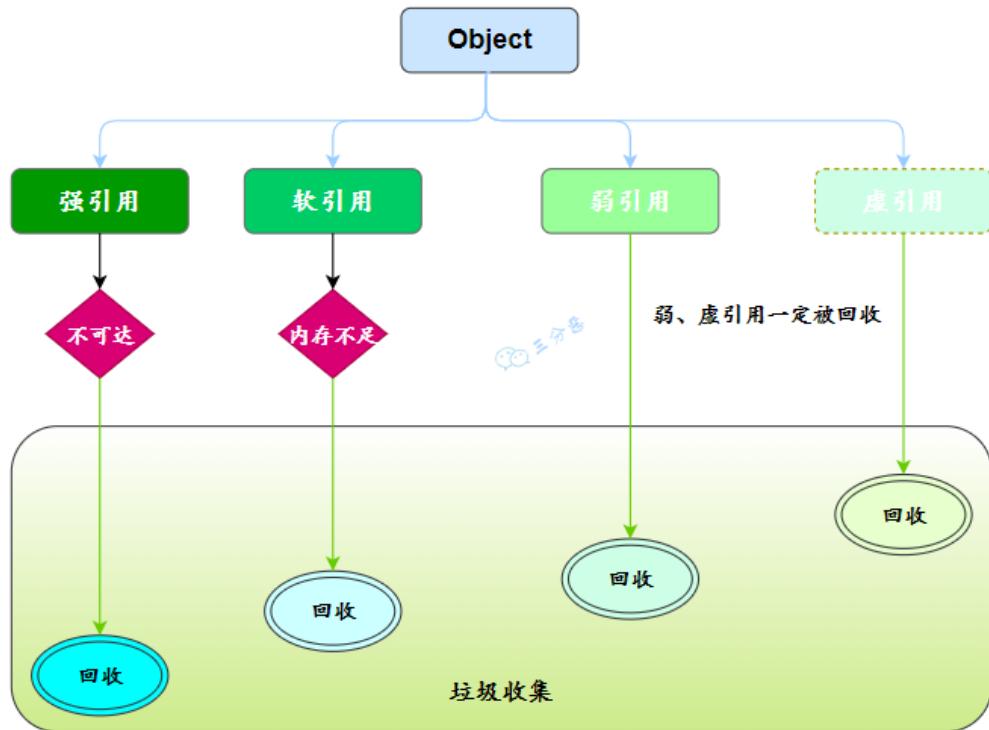
```
1 |     Object obj = new Object();
2 |     ReferenceQueue queue = new ReferenceQueue();
3 |     SoftReference reference = new SoftReference(obj,
4 |           queue);
5 |     //强引用对象滞空，保留软引用
|     obj = null;
```

- 弱引用也是用来描述那些非必须对象，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生为止。当垃圾收集器开始工作，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。在JDK 1.2版之后提供了WeakReference类来实现弱引用。

```
1 |     Object obj = new Object();
2 |     ReferenceQueue queue = new ReferenceQueue();
3 |     WeakReference reference = new WeakReference(obj,
4 |           queue);
5 |     //强引用对象滞空，保留软引用
|     obj = null;
```

- 虚引用也称为“幽灵引用”或者“幻影引用”，它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用取得一个对象实例。为一个对象设置虚引用关联的唯一目的只是为了能在这个对象被收集器回收时收到一个系统通知。在JDK 1.2版之后提供了PhantomReference类来实现虚引用。

```
1 |     Object obj = new Object();
2 |     ReferenceQueue queue = new ReferenceQueue();
3 |     PhantomReference reference = new
4 |       PhantomReference(obj, queue);
5 |     //强引用对象滞空，保留软引用
|     obj = null;
```



16.finalize()方法了解吗？有什么作用？

用一个不太贴切的比喻，垃圾回收就是古代的秋后问斩，`finalize()`就是刀下留人，在人犯被处决之前，还要做最后一次审计，青天大老爷看看有没有什么冤情，需不需要刀下留人。

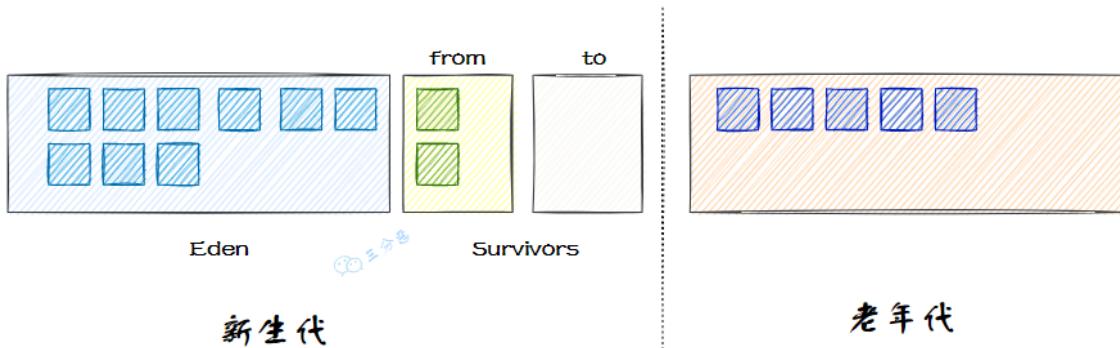


如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链，那它将会被第一次标记，随后进行一次筛选，筛选的条件是此对象是否有必要执行`finalize()`方法。如果对象在`finalize()`中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可，譬如把自己（`this`关键字）赋值给某个类变量或者对象的成员变量，那在第二次标记时它就“逃过一劫”；但是如果没有抓住这个机会，那么对象就真的要被回收了。

17. Java堆的内存分区了解吗？

按照垃圾收集，将Java堆划分为**新生代（Young Generation）** 和**老年代（Old Generation）** 两个区域，新生代存放存活时间短的对象，而每次回收后存活的少量对象，将会逐步晋升到老年代中存放。

而新生代又可以分为三个区域，eden、from、to，比例是8: 1: 1，而新生代的内存分区同样是从垃圾收集的角度来分配的。



18. 垃圾收集算法了解吗？

垃圾收集算法主要有三种：

1. 标记-清除算法

见名知义，**标记-清除**（Mark-Sweep）算法分为两个阶段：

- 标记：标记出所有需要回收的对象
- 清除：回收所有被标记的对象

标记-清除算法



标记-清除算法比较基础，但是主要存在两个缺点：

- 执行效率不稳定，如果Java堆中包含大量对象，而且其中大部分是需要被回收的，这时必须进行大量标记和清除的动作，导致标记和清除两个过程的执行效率都随对象数量增长而降低。
- 内存空间的碎片化问题，标记、清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致当以后在程序运行过程中需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

2. 标记-复制算法

标记-复制算法解决了标记-清除算法面对大量可回收对象时执行效率低的问题。

过程也比较简单：将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

标记-复制算法



这种算法存在一个明显的缺点：一部分空间没有使用，存在空间的浪费。

新生代垃圾收集主要采用这种算法，因为新生代的存活对象比较少，每次复制的只是少量的存活对象。当然，实际新生代的收集不是按照这个比例。

3. 标记-整理算法

为了降低内存的消耗，引入一种针对性的算法：**标记-整理**（Mark-Compact）算法。

其中的标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向内存空间一端移动，然后直接清理掉边界以外的内存。

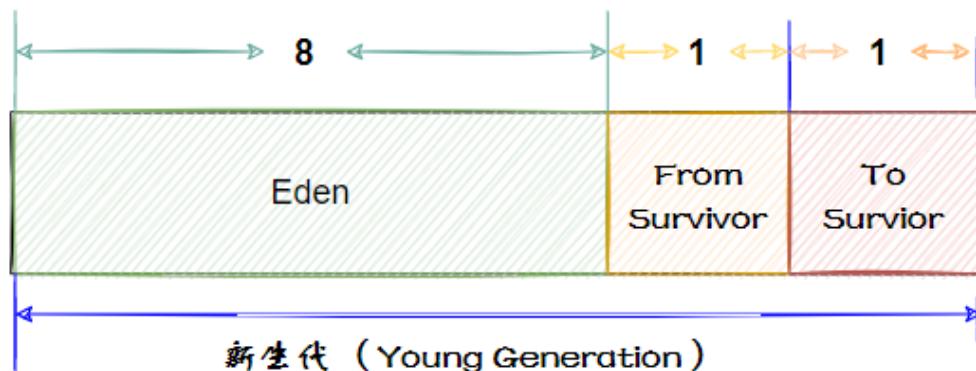


标记-整理算法主要用于老年代，移动存活对象是个极为负重的操作，而且这种操作需要Stop The World才能进行，只是从整体的吞吐量来考量，老年代使用标记-整理算法更加合适。

19.说一下新生代的区域划分？

新生代的垃圾收集主要采用标记-复制算法，因为新生代的存活对象比较少，每次复制少量的存活对象效率比较高。

基于这种算法，虚拟机将内存分为一块较大的Eden空间和两块较小的 Survivor空间，每次分配内存只使用Eden和其中一块Survivor。发生垃圾收集时，将Eden和 Survivor中仍然存活的对象一次性复制到另外一块Survivor空间上，然后直接清理掉 Eden和已用过的那块Survivor空间。默认Eden和Survivor的大小比例是8：1。



20.Minor GC/Young GC、Major GC/Old GC、Mixed GC、Full GC都是什么意思？

部分收集（Partial GC）：指目标不是完整收集整个Java堆的垃圾收集，其中又分为：

- 新生代收集（Minor GC/Young GC）：指目标只是新生代的垃圾收集。
- 老年代收集（Major GC/Old GC）：指目标只是老年代的垃圾收集。目前只有 CMS收集器会有单独收集老年代的行为。
- 混合收集（Mixed GC）：指目标是收集整个新生代以及部分老年代的垃圾收集。目前只有G1收集器会有这种行为。

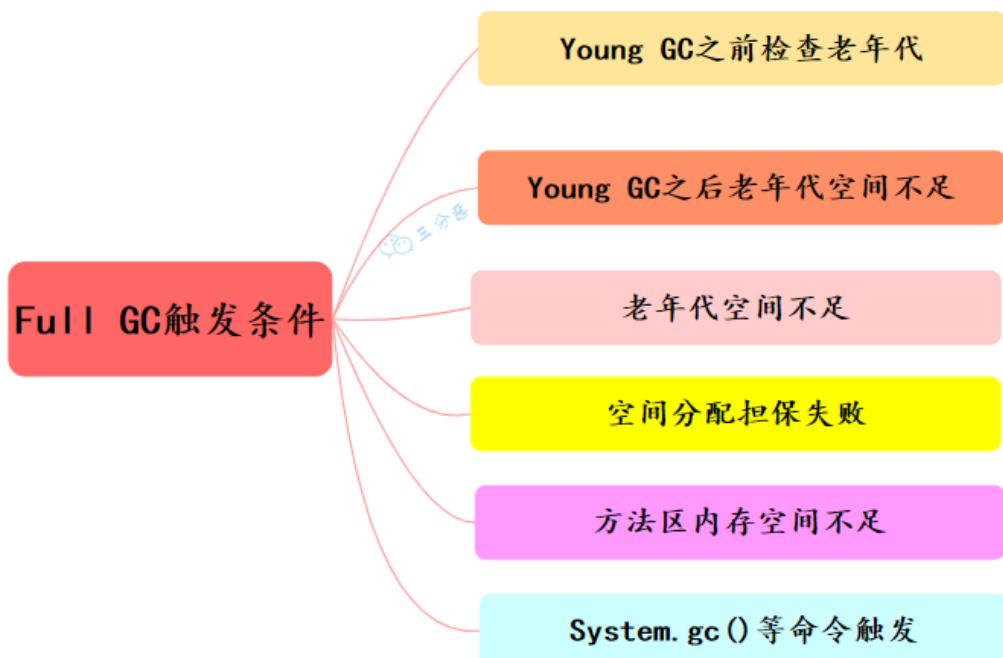
整堆收集（Full GC）：收集整个Java堆和方法区的垃圾收集。

21. Minor GC/Young GC什么时候触发?

新创建的对象优先在新生代Eden区进行分配，如果Eden区没有足够的空间时，就会触发Young GC来清理新生代。

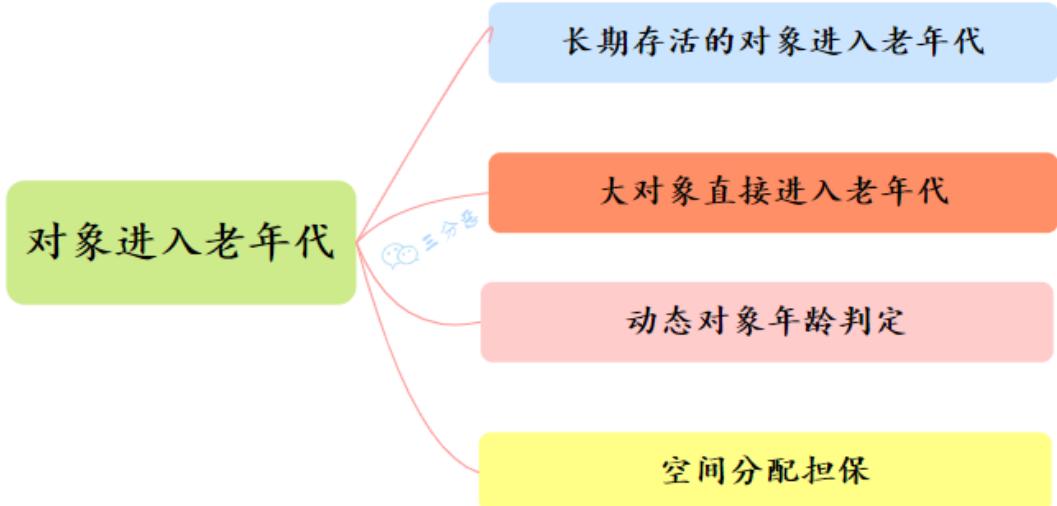
22. 什么时候会触发Full GC?

这个触发条件稍微有点多，往下看：



- Young GC之前检查老年代：在要进行 Young GC 的时候，发现 老年代可用的连续 内存空间 < 新生代历次 Young GC 后升入老年代的对象总和的平均大小，说明本次 Young GC 后可能升入老年代的对象大小，可能超过了老年代当前可用内存空间，那就会触发 Full GC。
- Young GC之后老年代空间不足：执行 Young GC 之后有一批对象需要放入老年代，此时老年代就是没有足够的内存空间存放这些对象了，此时必须立即触发一次Full GC
- 老年代空间不足，老年代内存使用率过高，达到一定比例，也会触发Full GC。
- 空间分配担保失败（Promotion Failure），新生代的 To 区放不下从 Eden 和 From 拷贝过来对象，或者新生代对象 GC 年龄到达阈值需要晋升这两种情况，老年代如果放不下的话都会触发 Full GC。
- 方法区内存空间不足：如果方法区由永久代实现，永久代空间不足 Full GC。
- System.gc()等命令触发：System.gc()、jmap -dump 等命令会触发 full gc。

23. 对象什么时候会进入老年代？



长期存活的对象将进入老年代

在对象的对头信息中存储着对象的迭代年龄,迭代年龄会在每次YoungGC之后对象的移区操作中增加,每一次移区年龄加一.当这个年龄达到15(默认)之后,这个对象将会被移入老年代。

可以通过这个参数设置这个年龄值。

```
1 | -XX:MaxTenuringThreshold
```

大对象直接进入老年代

有一些占用大量连续内存空间的对象在被加载就会直接进入老年代.这样的大对象一般是一些数组,长字符串之类的对。

HotSpot虚拟机提供了这个参数来设置。

```
1 | -XX:PretenureSizeThreshold
```

动态对象年龄判定

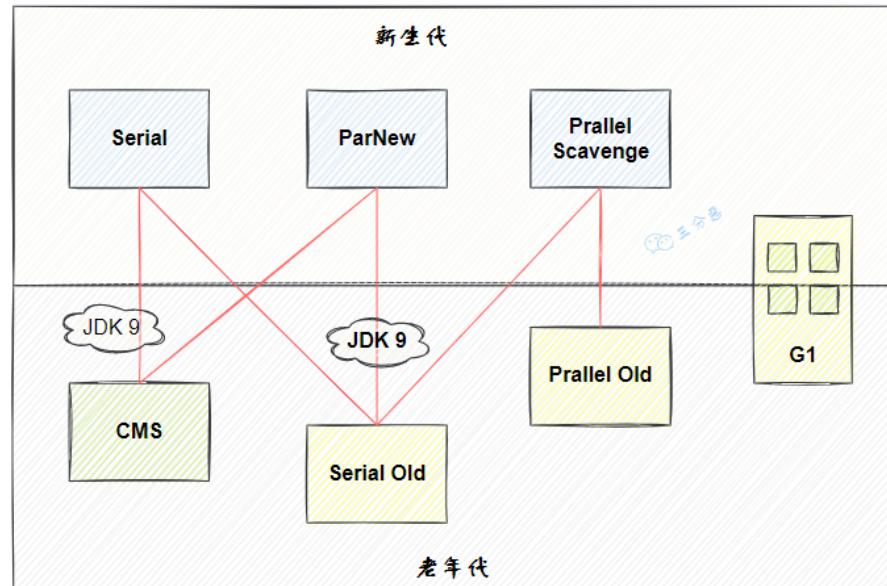
为了能更好地适应不同程序的内存状况，HotSpot虚拟机并不是永远要求对象的年龄必须达到`-XX:MaxTenuringThreshold`才能晋升老年代，如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代。

空间分配担保

假如在Young GC之后，新生代仍然有大量对象存活，就需要老年代进行分配担保，把Survivor无法容纳的对象直接送入老年代。

24. 知道有哪些垃圾收集器吗？

主要垃圾收集器如下，图中标出了它们的工作区域、垃圾收集算法，以及配合关系。



这些收集器里，面试的重点是两个——**CMS** 和 **G1**。

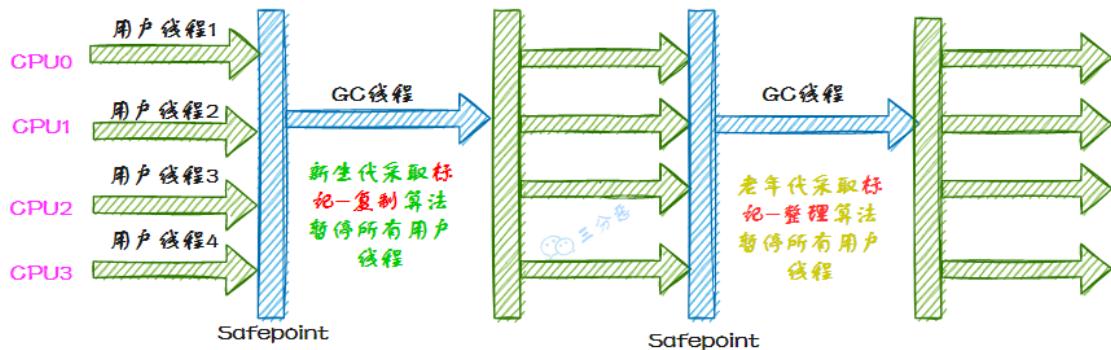
- Serial收集器

Serial收集器是最基础、历史最悠久的收集器。

如同它的名字（串行），它是一个单线程工作的收集器，使用一个处理器或一条收集线程去完成垃圾收集工作。并且进行垃圾收集时，必须暂停其他所有工作线程，直到垃圾收集结束——这就是所谓的“Stop The World”。

Serial/Serial Old收集器的运行过程如图：

Serial/Serial Old收集器运行示意图

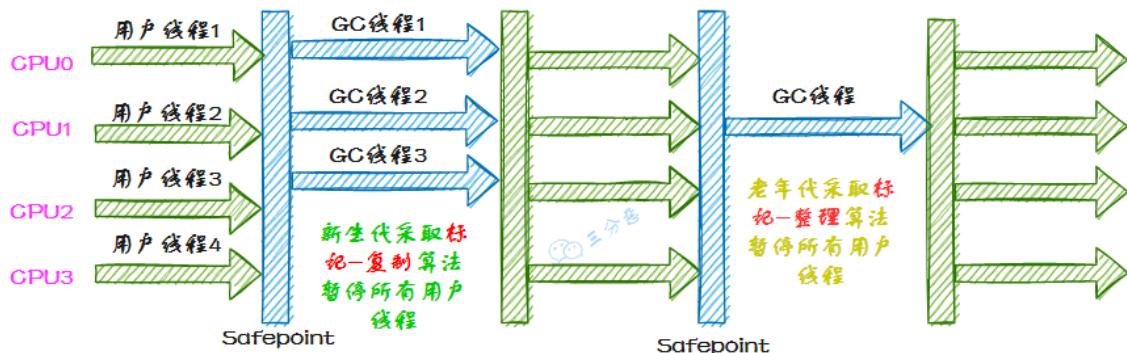


- ParNew

ParNew收集器实质上是Serial收集器的多线程并行版本，使用多条线程进行垃圾收集。

ParNew/Serial Old收集器运行示意图如下：

ParNew/Serial Old收集器运行示意图



- Parallel Scavenge

Parallel Scavenger收集器是一款新生代收集器，基于标记-复制算法实现，也能够并行收集。和ParNew有些类似，但Parallel Scavenger主要关注的是垃圾收集的吞吐量——所谓吞吐量，就是CPU用于运行用户代码的时间和总消耗时间的比值，比值越大，说明垃圾收集的占比越小。

$$\text{吞吐量} = \frac{\text{运行用户代码的时间}}{\text{运行垃圾收集时间} + \text{运行用户代码的时间}}$$

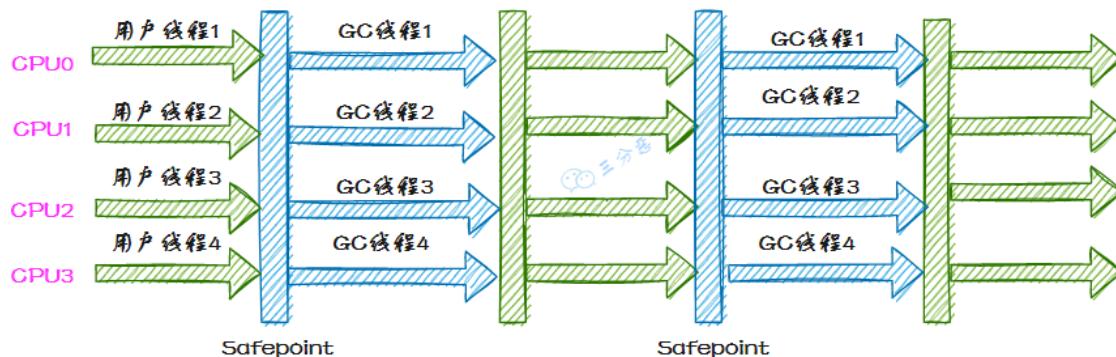
- Serial Old

Serial Old是Serial收集器的老年代版本，它同样是一个单线程收集器，使用标记-整理算法。

- Parallel Old

Parallel Old是Parallel Scavenge收集器的老年代版本，支持多线程并发收集，基于标记-整理算法实现。

Parallel Scavenge/Parallel Old收集器运行示意图



- CMS收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为为目标的收集器，同样是老年代的收集器，采用标记-清除算法。

- Garbage First收集器

Garbage First（简称G1）收集器是垃圾收集器的一个颠覆性的产物，它开创了局部收集的设计思路和基于Region的内存布局形式。

25.什么是Stop The World ? 什么是 OopMap ? 什么是安全点?

进行垃圾回收的过程中，会涉及对象的移动。为了保证对象引用更新的正确性，必须暂停所有的用户线程，像这样的停顿，虚拟机设计者形象描述为 **Stop The World**。也简称为STW。

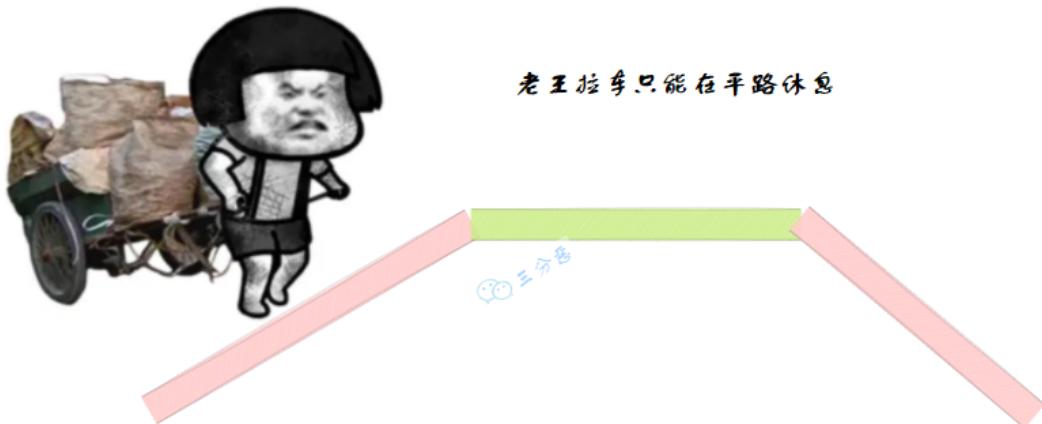
在HotSpot中，有个数据结构（映射表）称为 **OopMap**。一旦类加载动作完成的时候，HotSpot就会把对象内什么偏移量上是什么类型的数据计算出来，记录到 OopMap。在即时编译过程中，也会在 **特定的位置** 生成 OopMap，记录下栈上和寄存器里哪些位置是引用。

这些特定的位置主要在：

- 1. 循环的末尾（非 counted 循环）
- 2. 方法临返回前 / 调用方法的call指令后
- 3. 可能抛异常的位置

这些位置就叫作**安全点(safepoint)**。 用户程序执行时并非在代码指令流的任意位置都能够停顿下来开始垃圾收集，而是必须是执行到安全点才能够暂停。

用通俗的比喻，假如老王去拉车，车上东西很重，老王累的汗流浃背，但是老王不能在上坡或者下坡休息，只能在平地上停下来擦擦汗，喝口水。



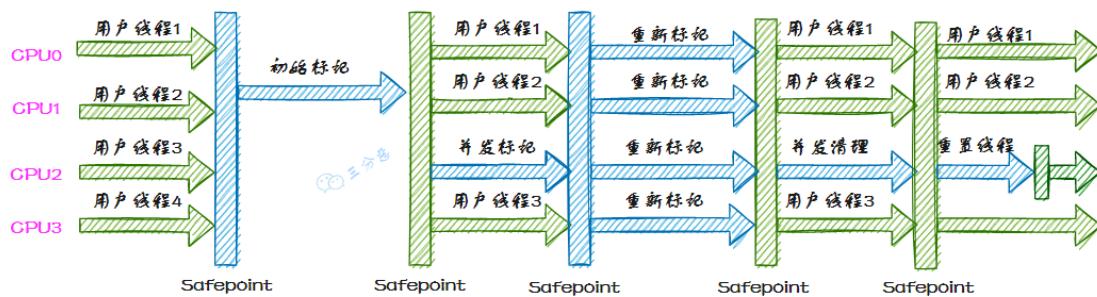
26. 能详细说一下CMS收集器的垃圾收集过程吗？

CMS收集器的垃圾收集分为四步：

- 初始标记（CMS initial mark）：单线程运行，需要Stop The World，标记GC Roots能直达的对象。
- 并发标记（CMS concurrent mark）：无停顿，和用户线程同时运行，从GC Roots直达对象开始遍历整个对象图。
- 重新标记（CMS remark）：多线程运行，需要Stop The World，标记并发标记阶段产生对象。
- 并发清除（CMS concurrent sweep）：无停顿，和用户线程同时运行，清理掉标记阶段标记的死亡的对象。

Concurrent Mark Sweep收集器运行示意图如下：

CMS收集器运行示意图

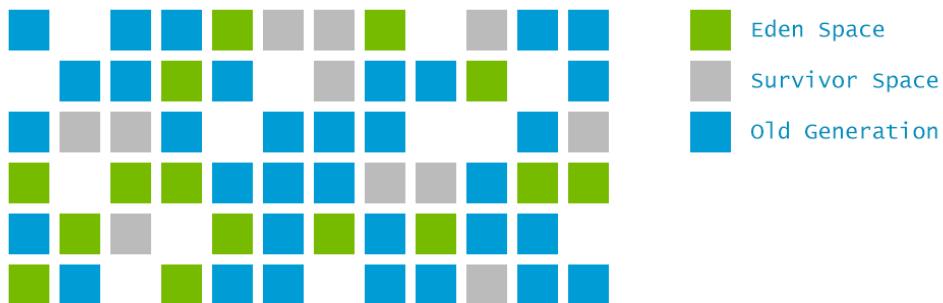


27.G1垃圾收集器了解吗？

Garbage First（简称G1）收集器是垃圾收集器的一个颠覆性的产物，它开创了局部收集的设计思路和基于Region的内存布局形式。

虽然G1也仍是遵循分代收集理论设计的，但其堆内存的布局与其他收集器有非常明显的差异。以前的收集器分代是划分新生代、老年代、持久代等。

G1把连续的Java堆划分为多个大小相等的独立区域（Region），每一个Region都可以根据需要，扮演新生代的Eden空间、Survivor空间，或者老年代空间。收集器能够对扮演不同角色的Region采用不同的策略去处理。

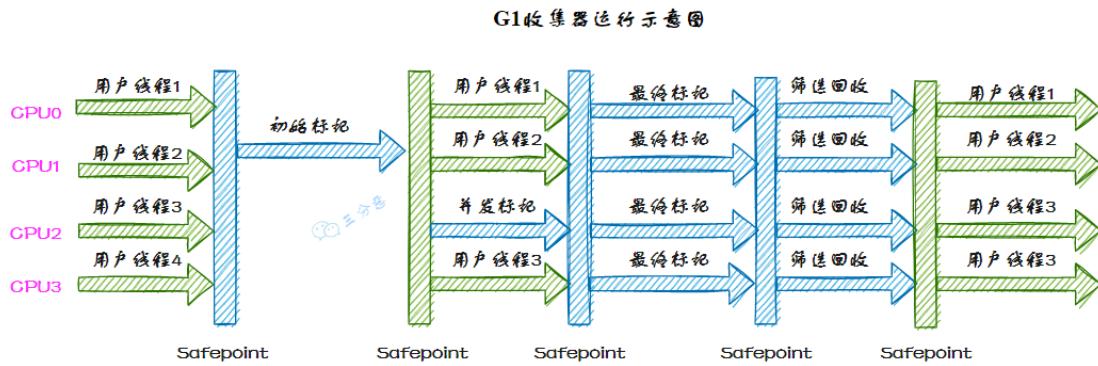


这样就避免了收集整个堆，而是按照若干个Region集进行收集，同时维护一个优先级列表，跟踪各个Region回收的“价值”，优先收集价值高的Region。

G1收集器的运行过程大致可划分为以下四个步骤：

- 初始标记（initial mark），标记了从GC Root开始直接关联可达的对象。
- STW（Stop the World）执行。
- 并发标记（concurrent marking），和用户线程并发执行，从GC Root开始对堆中对象进行可达性分析，递归扫描整个堆里的对象图，找出要回收的对象。
- 最终标记（Remark），STW，标记再并发标记过程中产生的垃圾。
- 筛选回收（Live Data Counting And Evacuation），制定回收计划，选择多个Region构成回收集，把回收集中Region的存活对象复制到空的Region中，再清理

掉整个旧 Region 的全部空间。需要STW。



28.有了CMS，为什么还要引入G1？

优点：CMS最主要的优点在名字上已经体现出来——并发收集、低停顿。

缺点：CMS同样有三个明显的缺点。

- Mark Sweep算法会导致内存碎片比较多
- CMS的并发能力比较依赖于CPU资源，并发回收时垃圾收集线程可能会抢占用用户线程的资源，导致用户程序性能下降。
- 并发清除阶段，用户线程依然在运行，会产生所谓的“浮动垃圾”（Floating Garbage），本次垃圾收集无法处理浮动垃圾，必须到下一次垃圾收集才能处理。如果浮动垃圾太多，会触发新的垃圾回收，导致性能降低。

G1主要解决了内存碎片过多的问题。

29.你们线上用的什么垃圾收集器？为什么要用它？

怎么说呢，虽然调优说的震天响，但是我们一般都是用默认。管你Java怎么升，我用8，那么JDK1.8默认用的是什么呢？

可以使用命令：

```
1 | java -XX:+PrintCommandLineFlags -version
```

可以看到有这么一行：

```
1 | -XX:+UseParallelGC
```

`UseParallelGC = Parallel Scavenge + Parallel Old`，表示的是新生代用的 `Parallel Scavenge` 收集器，老年代用的是 `Parallel Old` 收集器。

那为什么要用这个呢？默认的呗。

当然面试肯定不能这么答。

`Parallel Scavenge` 的特点是什么？

高吞吐，我们可以回答：因为我们系统是业务相对复杂，但并发并不是非常高，所以希望尽可能的利用处理器资源，出于提高吞吐量的考虑采用 `Parallel Scavenge + Parallel Old` 的组合。

当然，这个默认虽然也有说法，但不太讨喜。

还可以说：

采用 `Parallel New + CMS` 的组合，我们比较关注服务的响应速度，所以采用了 CMS 来降低停顿时间。

或者一步到位：

我们线上采用了设计比较优秀的 G1 垃圾收集器，因为它不仅满足我们低停顿的要求，而且解决了 CMS 的浮动垃圾问题、内存碎片问题。

30. 垃圾收集器应该如何选择？

垃圾收集器的选择需要权衡的点还是比较多的——例如运行应用的基础设施如何？使用 JDK 的发行商是什么？等等……

这里简单地列一下上面提到的一些收集器的适用场景：

- `Serial`：如果应用程序有一个很小的内存空间（大约 100 MB）亦或它在没有停顿时间要求的单线程处理器上运行。
- `Parallel`：如果优先考虑应用程序的峰值性能，并且没有时间要求要求，或者可以接受 1 秒或更长的停顿时间。
- `CMS/G1`：如果响应时间比吞吐量优先级高，或者垃圾收集暂停必须保持在大约 1 秒以内。
- `ZGC`：如果响应时间是高优先级的，或者堆空间比较大。

31. 对象一定分配在堆中吗？有没有了解逃逸分析技术？

对象一定分配在堆中吗？ 不一定的。

随着JIT编译期的发展与逃逸分析技术逐渐成熟，所有的对象都分配到堆上也渐渐变得不那么“绝对”了。其实，在编译期间，JIT会对代码做很多优化。其中有一部分优化的目的就是减少内存堆分配压力，其中一种重要的技术叫做逃逸分析。

什么是逃逸分析？

逃逸分析是指分析指针动态范围的方法，它同编译器优化原理的指针分析和外形分析相关联。当变量（或者对象）在方法中分配后，其指针有可能被返回或者被全局引用，这样就会被其他方法或者线程所引用，这种现象称作指针（或者引用）的逃逸(Escape)。

通俗点讲，当一个对象被new出来之后，它可能被外部所调用，如果是作为参数传递到外部了，就称之为方法逃逸。

未逃逸

```
public static void returnStr() {
    User user = new User();
    user.setId(1);
    user.setName("张三");
    user.setAge(18);
}

public static String returnStr() {
    User user = new User();
    user.setId(1);
    user.setName("张三");
    user.setAge(18);
    return user.toString(); //这里User要实现get, set方法，还要实现toString方法
}
```

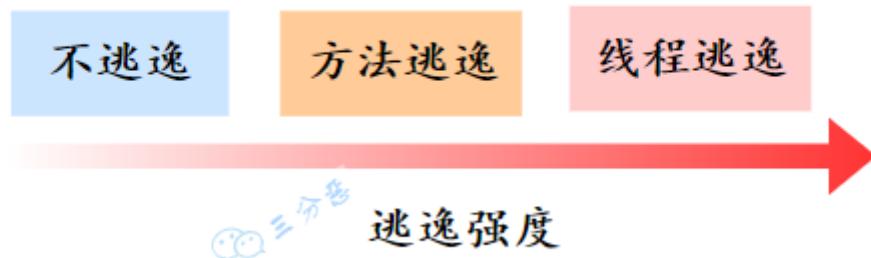
局部对象user未被外部调用

方法逃逸

```
public static User returnStr() {
    User user = new User();
    user.setId(1);
    user.setName("张三");
    user.setAge(18);
    return user; //这里User要实现get, set方法
}
```

局部对象user可能会被外部调用

除此之外，如果对象还有可能被外部线程访问到，例如赋值给可以在其它线程中访问的实例变量，这种就被称为线程逃逸。



逃逸分析的好处

- 栈上分配

如果确定一个对象不会逃逸到线程之外，那么久可以考虑将这个对象在栈上分配，对象占用的内存随着栈帧出栈而销毁，这样一来，垃圾收集的压力就降低很多。

- 同步消除

线程同步本身是一个相对耗时的过程，如果逃逸分析能够确定一个变量不会逃逸出线程，无法被其他线程访问，那么这个变量的读写肯定就不会有竞争，对这个变量实施的同步措施也就可以安全地消除掉。

- 标量替换

如果一个数据是基本数据类型，不可拆分，它就被称为标量。把一个Java对象拆散，将其用到的成员变量恢复为原始类型来访问，这个过程就称为标量替换。假如逃逸分析能够证明一个对象不会被方法外部访问，并且这个对象可以被拆散，那么可以不创建对象，直接用创建若干个成员变量代替，可以让对象的成员变量在栈上分配和读写。

JVM调优

32.有哪些常用的命令行性能监控和故障处理工具？

- 操作系统工具
 - top: 显示系统整体资源使用情况
 - vmstat: 监控内存和CPU
 - iostat: 监控IO使用
 - netstat: 监控网络使用

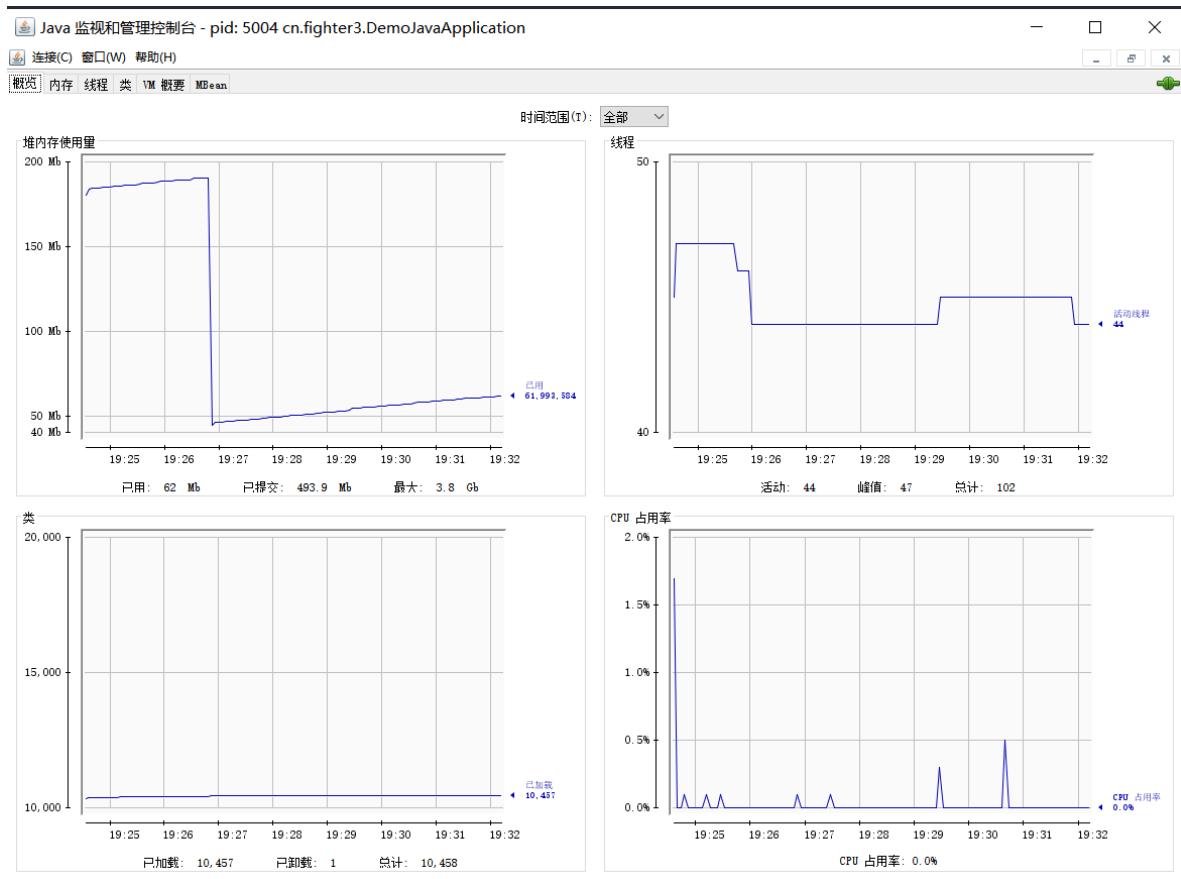
- JDK性能监控工具

- jps: 虚拟机进程查看
- jstat: 虚拟机运行时信息查看
- jinfo: 虚拟机配置查看
- jmap: 内存映像（导出）
- jhat: 堆转储快照分析
- jstack: Java堆栈跟踪
- jcmd: 实现上面除了jstat外所有命令的功能

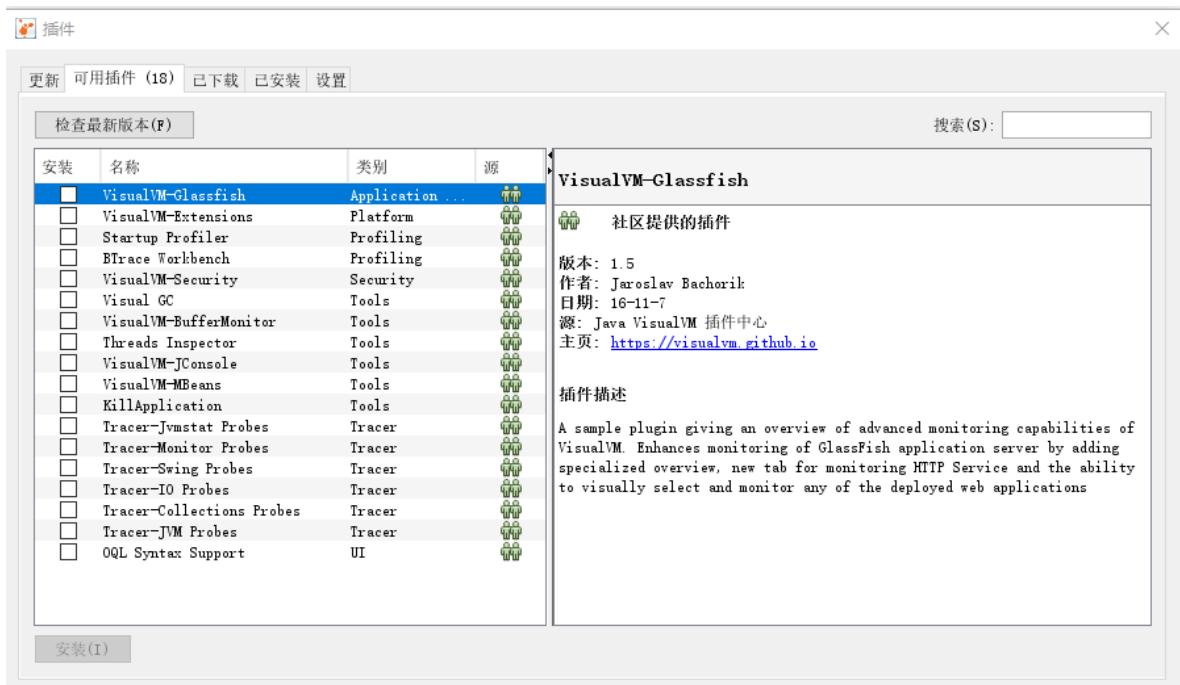
33.了解哪些可视化的性能监控和故障处理工具？

以下是一些JDK自带的可视化性能监控和故障处理工具：

- JConsole



- VisualVM



- Java Mission Control

Oracle Java Mission Control

The screenshot shows the Oracle Java Mission Control interface. At the top, there are tabs for 'JVM 浏览器' and '事件类型'. The 'JVM 浏览器' tab is active, showing a tree view of running JVMs. One JVM, '[1.8.0_221] cn.fighter3.DemoJ', is expanded, revealing its MBeans. A tooltip for the 'MBean 服务器' node states: 'MBean 服务器可用于自测和控制运行的 JVM 的各个方面' (The MBean server can be used for self-testing and controlling the running JVM in various aspects).

除此之外，还有一些第三方的工具：

- MAT

Java 堆内存分析工具。

- GChisto

GC 日志分析工具。

- GCViewer

GC 日志分析工具。

- JProfiler

商用的性能分析利器。

- arthas

阿里开源诊断工具。

- async-profiler

Java 应用性能分析工具，开源、火焰图、跨平台。

34.JVM的常见参数配置知道哪些？

一些常见的参数配置：

堆配置：

- -Xms:初始堆大小
- -Xmx: 最大堆大小
- -XX:NewSize=n:设置年轻代大小
- -XX:NewRatio=n:设置年轻代和年老代的比值。如：为3表示年轻代和年老代比值为1: 3，年轻代占整个年轻代年老代和的1/4
- -XX:SurvivorRatio=n:年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如3表示Eden: 3 Survivor: 2，一个Survivor区占整个年轻代的1/5
- -XX:MaxPermSize=n:设置持久代大小

收集器设置：

- -XX:+UseSerialGC:设置串行收集器
- -XX:+UseParallelGC:设置并行收集器
- -XX:+UseParalledOldGC:设置并行年老代收集器
- -XX:+UseConcMarkSweepGC:设置并发收集器

并行收集器设置

- -XX:ParallelGCThreads=n:设置并行收集器收集时使用的CPU数。并行收集线程数
- -XX:MaxGCPauseMillis=n:设置并行收集最大的暂停时间（如果到这个时间了，垃圾回收器依然没有回收完，也会停止回收）
- -XX:GCTimeRatio=n:设置垃圾回收时间占程序运行时间的百分比。公式为： $1/(1+n)$
- -XX:+CMSIncrementalMode:设置为增量模式。适用于单CPU情况

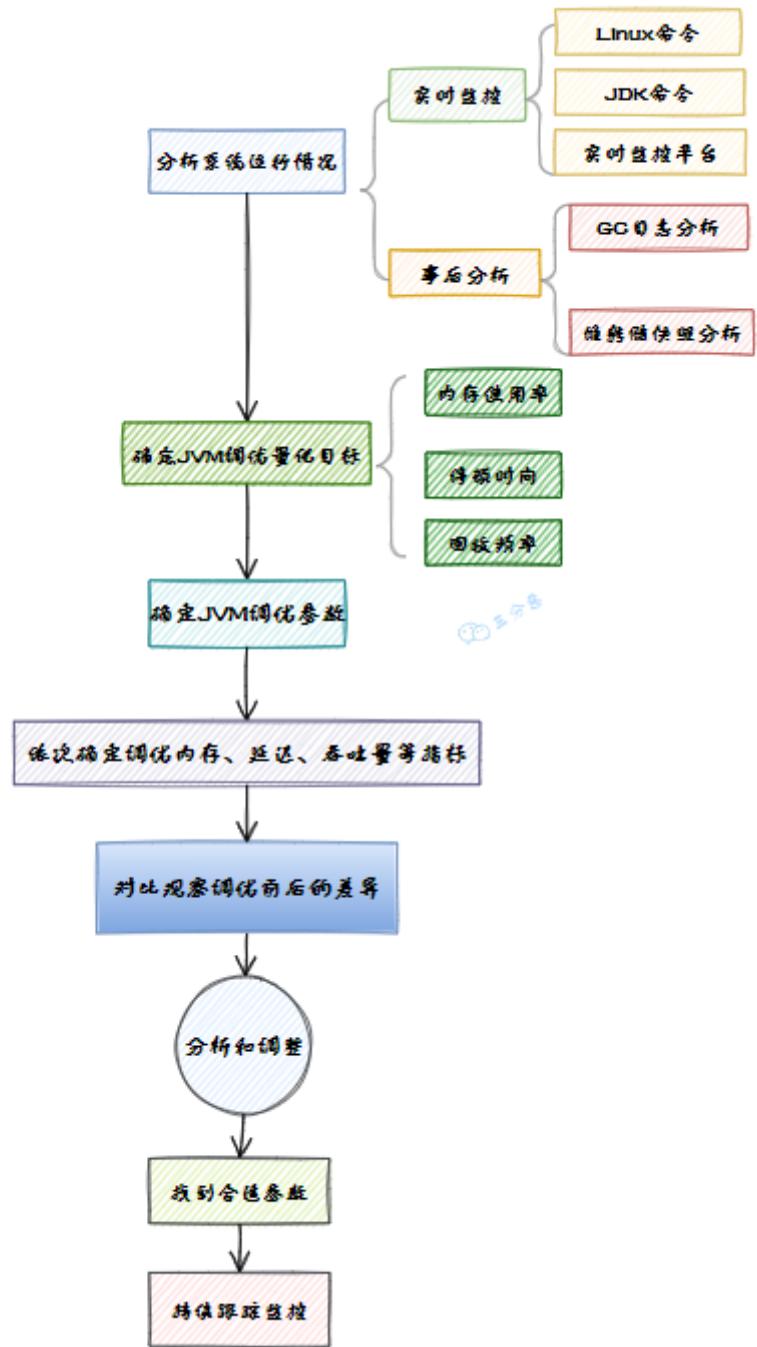
- `-XX:ParallelGCThreads=n`: 设置并发收集器年轻代垃圾回收方式为并行收集时，使用的CPU数。并行收集线程数

打印GC回收的过程日志信息

- `-XX:+PrintGC`
- `-XX:+PrintGCDetails`
- `-XX:+PrintGCTimeStamps`
- `-Xloggc:filename`

35.有做过JVM调优吗？

JVM调优是一件很严肃的事情，不是拍脑门就开始调优的，需要有严密的分析和监控机制，大概的一个JVM调优流程图：



实际上，JVM调优是不得已而为之，有那功夫，好好把烂代码重构一下不比瞎调JVM强。

但是，面试官非要问怎么办？可以从处理问题的角度来回答（对应图中事后），这是一个中规中矩的案例：电商公司的运营后台系统，偶发性的引发OOM异常，堆内存溢出。

- 1、因为是偶发性的，所以第一次简单的认为就是堆内存不足导致，单方面的加大了堆内存从4G调整到8G -Xms8g。
- 2、但是问题依然没有解决，只能从堆内存信息下手，通过开启了-XX:+HeapDumpOnOutOfMemoryError参数 获得堆内存的dump文件。

3、用JProfiler对堆dump文件进行分析，通过JProfiler查看到占用内存最大的对象是String对象，本来想跟踪着String对象找到其引用的地方，但dump文件太大，跟踪进去的时候总是卡死，而String对象占用比较多也比较正常，最开始也没有认定就是这里的问题，于是就从线程信息里面找突破点。

4、通过线程进行分析，先找到了几个正在运行的业务线程，然后逐一跟进业务线程看了下代码，有个方法引起了我的注意，[导出订单信息](#)。

5、因为订单信息导出这个方法可能会有几万的数据量，首先要从数据库里面查询出来订单信息，然后把订单信息生成excel，这个过程会产生大量的String对象。

6、为了验证自己的猜想，于是准备登录后台去测试下，结果在测试的过程中发现导出订单的按钮前端居然没有做点击后按钮置灰交互事件，后端也没有做防止重复提交，因为导出订单数据本来就非常慢，使用的人员可能发现点击后很久后页面都没反应，然后就一直点，结果就大量的请求进入到后台，堆内存产生了大量的订单对象和EXCEL对象，而且方法执行非常慢，导致这一段时间内这些对象都无法被回收，所以最终导致内存溢出。

7、知道了问题就容易解决了，最终没有调整任何JVM参数，只是做了两个处理：

- 在前端的导出订单按钮上加上了置灰状态，等后端响应之后按钮才可以进行点击
- 后端代码加分布式锁，做防重处理

这样双管齐下，保证导出的请求不会一直打到服务端，问题解决！

36.线上服务CPU占用过高怎么排查？

问题分析：CPU高一定是某个程序长期占用了CPU资源。

CPU飙高排查



1、所以先需要找出那个进程占用CPU高。

- top 列出系统各个进程的资源占用情况。

2、然后根据找到对应进行里哪个线程占用CPU高。

- top -Hp 进程ID 列出对应进程里面的线程占用资源情况

3、找到对应线程ID后，再打印出对应线程的堆栈信息

- printf "%x\n" PID 把线程ID转换为16进制。
- jstack PID 打印出进程的所有线程信息，从打印出来的线程信息中找到上一步转换为16进制的线程ID对应的线程信息。

4、最后根据线程的堆栈信息定位到具体业务方法，从代码逻辑中找到问题所在。

查看是否有线程长时间的waiting 或blocked，如果线程长期处于waiting状态下，关注 waiting on xxxxxxx，说明线程在等待这把锁，然后根据锁的地址找到持有锁的线程。

37. 内存飙升问题怎么排查？

分析：内存飙升如果是发生在java进程上，一般是因为创建了大量对象所导致，持续飙升说明垃圾回收跟不上对象创建的速度，或者内存泄露导致对象无法回收。

1、先观察垃圾回收的情况

- jstat -gc PID 1000 查看GC次数，时间等信息，每隔一秒打印一次。
- jmap -histo PID | head -20 查看堆内存占用空间最大的前20个对象类型，可初步查看是哪个对象占用了内存。

如果每次GC次数频繁，而且每次回收的内存空间也正常，那说明是因为对象创建速度快导致内存一直占用很高；如果每次回收的内存非常少，那么很可能是因为内存泄露导致内存一直无法被回收。

2、导出堆内存文件快照

- jmap -dump:live,format=b,file=/home/myheapdump.hprof PID dump堆内存信息到文件。

3、使用visualVM对dump文件进行离线分析，找到占用内存高的对象，再找到创建该对象的业务代码位置，从代码和业务场景中定位具体问题。

38. 频繁 minor gc 怎么办？

优化Minor GC频繁问题：通常情况下，由于新生代空间较小，Eden区很快被填满，就会导致频繁Minor GC，因此可以通过增大新生代空间 `-Xmn` 来降低Minor GC的频率。

39. 频繁Full GC怎么办？

Full GC的排查思路大概如下：

1. 清楚从程序角度，有哪些原因导致FGC？

- 大对象：系统一次性加载了过多数据到内存中（比如SQL查询未做分页），导致大对象进入了老年期。
- 内存泄漏：频繁创建了大量对象，但是无法被回收（比如IO对象使用完后未调用close方法释放资源），先引发FGC，最后导致OOM.
- 程序频繁生成一些长生命周期的对象，当这些对象的存活年龄超过分代年龄时便会进入老年期，最后引发FGC.（即本文中的案例）
- 程序BUG
- 代码中显式调用了gc方法，包括自己的代码甚至框架中的代码。
- JVM参数设置问题：包括总内存大小、新生代和老年期的大小、Eden区和S区的大小、元空间大小、垃圾回收算法等等。

2. 清楚排查问题时能使用哪些工具

- 公司的监控系统：大部分公司都会有，可全方位监控JVM的各项指标。
- JDK的自带工具，包括jmap、jstat等常用命令：

```
1 # 查看堆内存各区域的使用率以及GC情况
2 jstat -gcutil -h20 pid 1000
3 # 查看堆内存中的存活对象，并按空间排序
4 jmap -histo pid | head -n20
5 # dump堆内存文件
6 jmap -dump:format=b,file=heap pid
```

- 可可视化的堆内存分析工具：JVisualVM、MAT等

3. 排查指南

- 查看监控，以了解出现问题的时间点以及当前FGC的频率（可对比正常情况看频率是否正常）
- 了解该时间点之前有没有程序上线、基础组件升级等情况。

- 了解JVM的参数设置，包括：堆空间各个区域的大小设置，新生代和老年代分别采用了哪些垃圾收集器，然后分析JVM参数设置是否合理。
- 再对步骤1中列出的可能原因做排除法，其中元空间被打满、内存泄漏、代码显式调用gc方法比较容易排查。
- 针对大对象或者长生命周期对象导致的FGC，可通过 jmap -histo 命令并结合 dump堆内存文件作进一步分析，需要先定位到可疑对象。
- 通过可疑对象定位到具体代码再次分析，这时候要结合GC原理和JVM参数设置，弄清楚可疑对象是否满足了进入到老年代的条件才能下结论。

40.有没有处理过内存泄漏问题？是如何定位的？

内存泄漏是内在病源，外在病症表现可能有：

- 应用程序长时间连续运行时性能严重下降
- CPU 使用率飙升，甚至到 100%
- 频繁 Full GC，各种报警，例如接口超时报警等
- 应用程序抛出 `OutOfMemoryError` 错误
- 应用程序偶尔会耗尽连接对象

严重 **内存泄漏** 往往伴随频繁的 **Full GC**，所以分析排查内存泄漏问题首先还得从查看 Full GC 入手。主要有以下操作步骤：

1. 使用 `jps` 查看运行的 Java 进程 ID
2. 使用 `top -p [pid]` 查看进程使用 CPU 和 MEM 的情况
3. 使用 `top -Hp [pid]` 查看进程下的所有线程占 CPU 和 MEM 的情况
4. 将线程 ID 转换为 16 进制：`printf "%x\n" [pid]`，输出的值就是线程栈信息中的 **nid**。

例如：`printf "%x\n" 29471`，换行输出 **731f**。

5. 抓取线程栈：`jstack 29452 > 29452.txt`，可以多抓几次做个对比。

在线程栈信息中找到对应线程号的 16 进制值，如下是 **731f** 线程的信息。线程栈分析可使用 Visualvm 插件 **TDA**。

```
1 | "Service Thread" #7 daemon prio=9 os_prio=0
  | tid=0x00007fbe2c164000 nid=0x731f runnable
  | [0x0000000000000000]
2      java.lang.Thread.State: RUNNABLE
```

6. 使用 `jstat -gcutil [pid] 5000 10` 每隔 5 秒输出 GC 信息，输出 10 次，查看 YGC 和 Full GC 次数。通常会出现 YGC 不增加或增加缓慢，而 Full GC 增加很快。

或使用 `jstat -gccause [pid] 5000`，同样是输出 GC 摘要信息。

或使用 `jmap -heap [pid]` 查看堆的摘要信息，关注老年代内存使用是否达到阀值，若达到阀值就会执行 Full GC。

7. 如果发现 Full GC 次数太多，就很大概率存在内存泄漏了
8. 使用 `jmap -histo:live [pid]` 输出每个类的对象数量，内存大小(字节单位)及全限定类名。
9. 生成 dump 文件，借助工具分析哪个对象非常多，基本就能定位到问题在那了

使用 jmap 生成 dump 文件：

```
1 # jmap -dump:live,format=b,file=29471.dump 29471
2 Dumping heap to /root/dump ...
3 Heap dump file created
```

10. dump 文件分析

可以使用 `jhat` 命令分析：`jhat -port 8000 29471.dump`，浏览器访问 jhat 服务，端口是 8000。

通常使用图形化工具分析，如 JDK 自带的 `jvisualvm`，从菜单 > 文件 > 装入 dump 文件。

或使用第三方式具分析的，如 `JProfiler` 也是个图形化工具，`GCHammer` 工具。Eclipse 或以使用 MAT 工具查看。或使用在线分析平台 `GCEasy`。

注意：如果 dump 文件较大的话，分析会占比较大的内存。

11. 在 dump 文析结果中查找存在大量的对象，再查对其的引用。

基本上就可以定位到代码层的逻辑了。

41. 有没有处理过内存溢出问题？

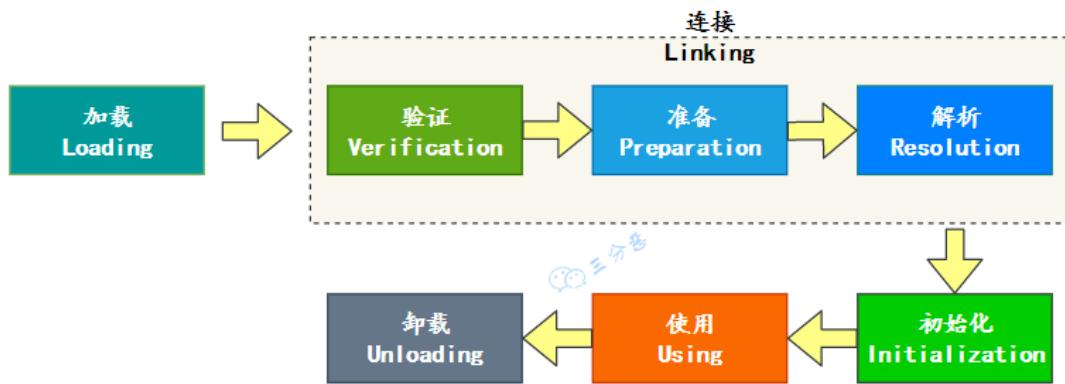
内存泄漏和内存溢出二者关系非常密切，内存溢出可能会有很多原因导致，内存泄漏最可能的罪魁祸首之一。

排查过程和排查内存泄漏过程类似。

虚拟机执行

42.能说一下类的生命周期吗？

一个类从被加载到虚拟机内存中开始，到从内存中卸载，整个生命周期需要经过七个阶段：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）和卸载（Unloading），其中验证、准备、解析三个部分统称为连接（Linking）。



43.类加载的过程知道吗？

加载是JVM加载的起点，具体什么时候开始加载，《Java虚拟机规范》中并没有进行强制约束，可以交给虚拟机的具体实现来自由把握。

在加载过程，JVM要做三件事情：



- 1) 通过一个类的全限定名来获取定义此类的二进制字节流。
- 2) 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
- 3) 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

加载阶段结束后，Java虚拟机外部的二进制字节流就按照虚拟机所设定的格式存储在方法区之中了，方法区中的数据存储格式完全由虚拟机实现自行定义，《Java虚拟机规范》未规定此区域的具体数据结构。

类型数据妥善安置在方法区之后，会在Java堆内存中实例化一个java.lang.Class类的对象，这个对象将作为程序访问方法区中的类型数据的外部接口。

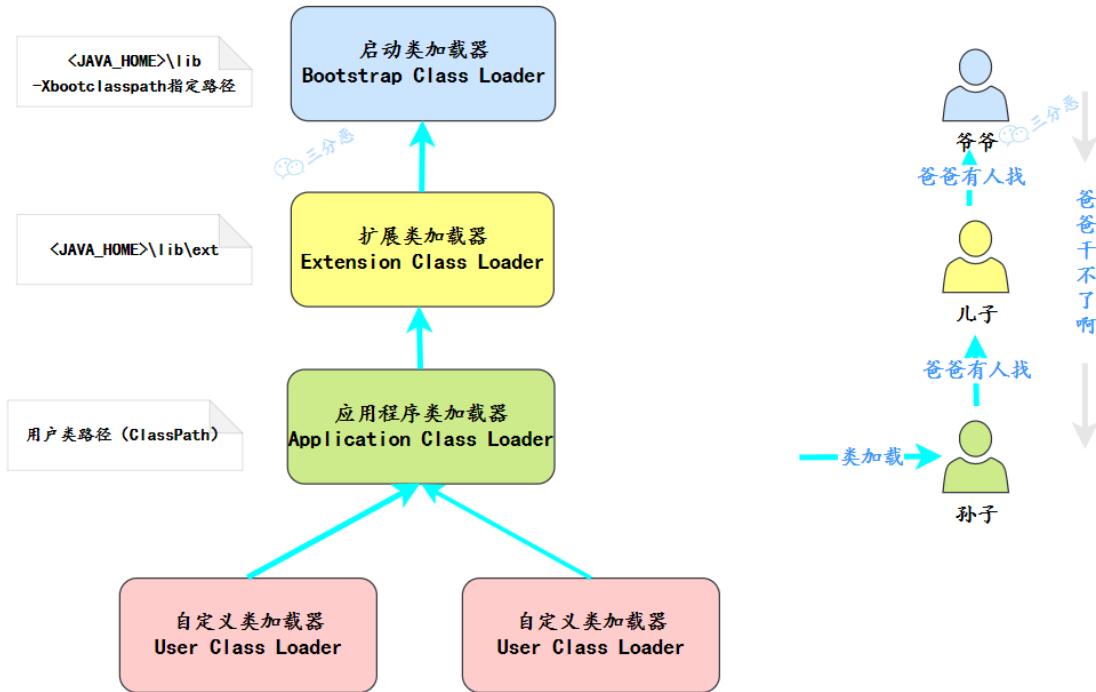
44.类加载器有哪些？

主要有四种类加载器：

- 启动类加载器 (Bootstrap ClassLoader)用来加载java核心类库，无法被java程序直接引用。
- 扩展类加载器 (extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
- 系统类加载器 (system class loader)：它根据 Java 应用的类路径 (CLASSPATH) 来加载Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过ClassLoader.getSystemClassLoader()来获取它。

- 用户自定义类加载器 (user class loader), 用户通过继承 `java.lang.ClassLoader` 类的方式自行实现的类加载器。

45.什么是双亲委派机制?



双亲委派模型的工作过程：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到最顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求时，子加载器才会尝试自己去完成加载。

46.为什么要用双亲委派机制?

答案是为了保证应用程序的稳定有序。

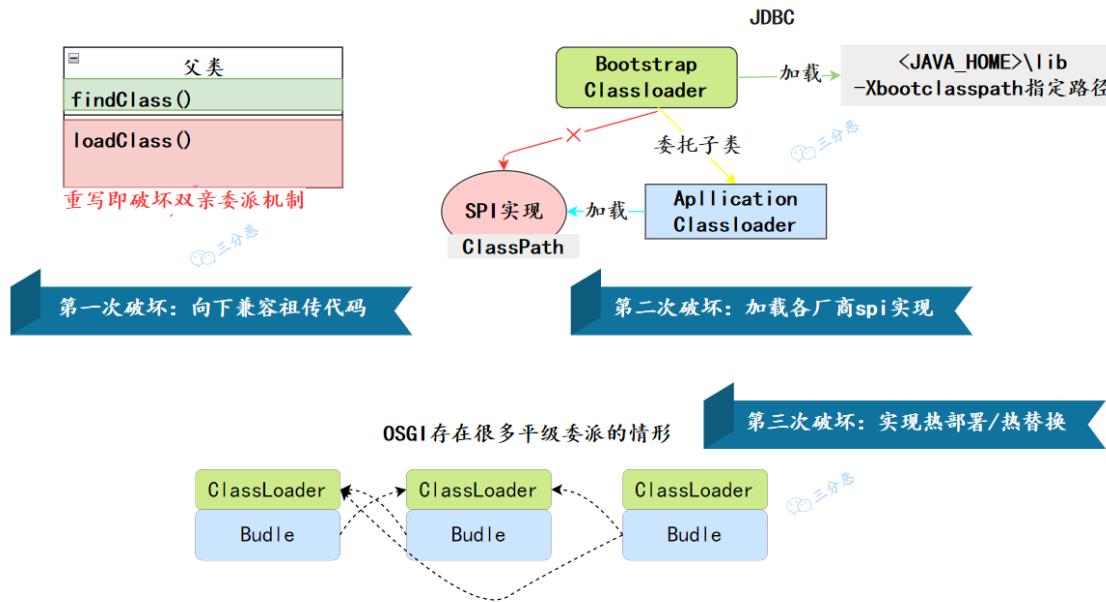
例如类`java.lang.Object`，它存放在`rt.jar`之中，通过双亲委派机制，保证最终都是委派给处于模型最顶端的启动类加载器进行加载，保证`Object`的一致。反之，都由各个类加载器自行去加载的话，如果用户自己也编写了一个名为`java.lang.Object`的类，并放在程序的 `ClassPath`中，那系统中就会出现多个不同的`Object`类。

47.如何破坏双亲委派机制?

如果不想打破双亲委派模型，就重写ClassLoader类中的findClass()方法即可，无法被父类加载器加载的类最终会通过这个方法被加载。而如果想打破双亲委派模型则需要重写loadClass()方法。

48. 历史上有哪几次双亲委派机制的破坏？

双亲委派机制在历史上主要有三次破坏：



第一次破坏

双亲委派模型的第一次“被破坏”其实发生在双亲委派模型出现之前——即JDK 1.2面世以前的“远古”时代。

由于双亲委派模型在JDK 1.2之后才被引入，但是类加载器的概念和抽象类`java.lang.ClassLoader`则在Java的第一个版本中就已经存在，为了向下兼容旧代码，所以无法以技术手段避免`loadClass()`被子类覆盖的可能性，只能在JDK 1.2之后的`java.lang.ClassLoader`中添加一个新的 `protected`方法`findClass()`，并引导用户编写的类加载逻辑时尽可能去重写这个方法，而不是在`loadClass()`中编写代码。

第二次破坏

双亲委派模型的第二次“被破坏”是由这个模型自身的缺陷导致的，如果有基础类型又要调用回用户的代码，那该怎么办呢？

例如我们比较熟悉的JDBC：

各个厂商各有不同的JDBC的实现，Java在核心包 `\lib` 里定义了对应的SPI，那么这个就毫无疑问由 **启动类加载器** 加载器加载。

但是各个厂商的实现，是没办法放在核心包里的，只能放在 `classpath` 里，只能被 **应用类加载器** 加载。那么，问题来了，启动类加载器它就加载不到厂商提供的SPI服务代码。

为了解决这个问题，引入了一个不太优雅的设计：线程上下文类加载器（Thread Context ClassLoader）。这个类加载器可以通过`java.lang.Thread`类的`setContextClassLoader()`方法进行设置，如果创建线程时还未设置，它将会从父线程中继承一个，如果在应用程序的全局范围内都没有设置过的话，那这个类加载器默认就是应用程序类加载器。

JNDI服务使用这个线程上下文类加载器去加载所需的SPI服务代码，这是一种父类加载器去请求子类加载器完成类加载的行为。

第三次破坏

双亲委派模型的第三次“被破坏”是由于用户对程序动态性的追求而导致的，例如代码热替换（Hot Swap）、模块热部署（Hot Deployment）等。

OSGi实现模块化热部署的关键是它自定义的类加载器机制的实现，每一个程序模块（OSGi中称为 Bundle）都有一个自己的类加载器，当需要更换一个Bundle时，就把Bundle连同类加载器一起换掉以实现代码的热替换。在OSGi环境下，类加载器不再双亲委派模型推荐的树状结构，而是进一步发展为更加复杂的网状结构。

49.你觉得应该怎么实现一个热部署功能？

我们已经知道了Java类的加载过程。一个Java类文件到虚拟机里的对象，要经过如下过程：首先通过Java编译器，将Java文件编译成class字节码，类加载器读取class字节码，再将类转化为实例，对实例`newInstance`就可以生成对象。

类加载器ClassLoader功能，也就是将class字节码转换到类的实例。在Java应用中，所有的实例都是由类加载器，加载而来。

一般在系统中，类的加载都是由系统自带的类加载器完成，而且对于同一个全限定名的java类（如`com.csiar.soc.HelloWorld`），只能被加载一次，而且无法被卸载。

这个时候问题就来了，如果我们希望将java类卸载，并且替换更新版本的java类，该怎么做呢？

既然在类加载器中，Java类只能被加载一次，并且无法卸载。那么我们是不是可以直接把Java类加载器干掉呢？答案是可以的，我们可以自定义类加载器，并重写 ClassLoader的findClass方法。

想要实现热部署可以分以下三个步骤：

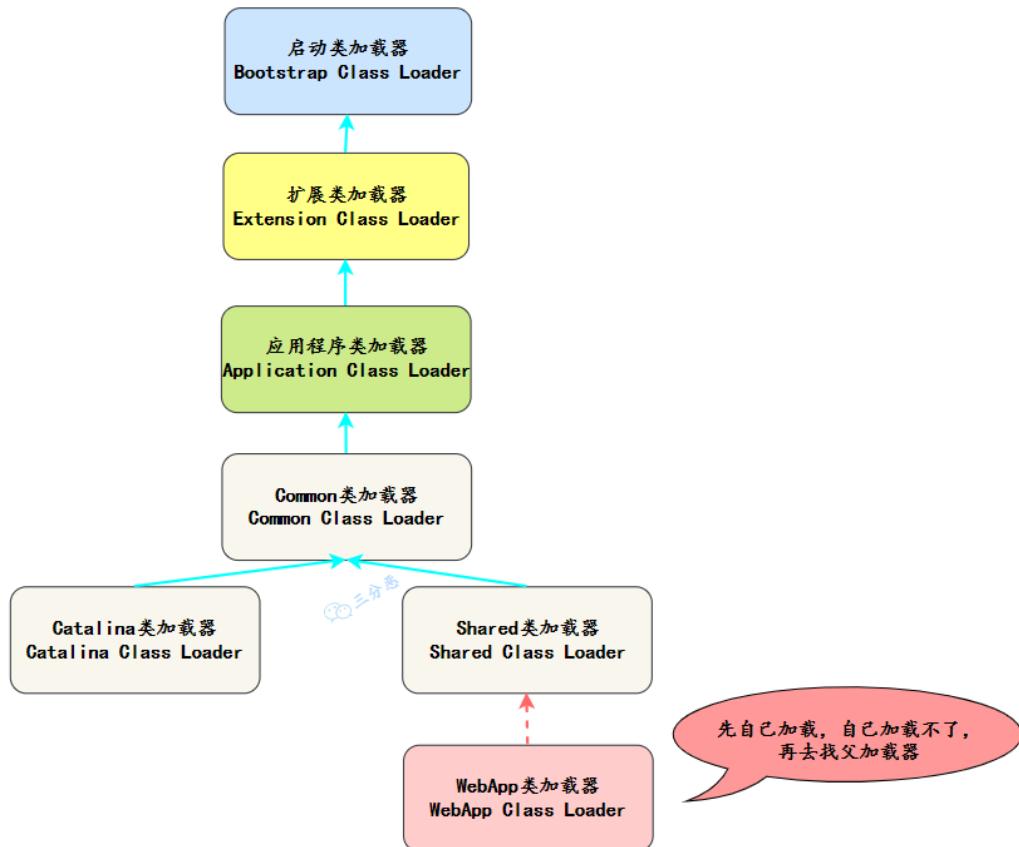
1. 销毁原来的自定义ClassLoader
2. 更新class类文件
3. 创建新的ClassLoader去加载更新后的class类文件。

到此，一个热部署的功能就这样实现了。

50.Tomcat的类加载机制了解吗？

Tomcat是主流的Java Web服务器之一，为了实现一些特殊的功能需求，自定义了一些类加载器。

Tomcat类加载器如下：



Tomcat实际上也是破坏了双亲委派模型的。

Tomcat是web容器，可能需要部署多个应用程序。不同的应用程序可能会依赖同一个第三方类库的不同版本，但是不同版本的类库中某一个类的全路径名可能是一样的。如多个应用都要依赖hollis.jar，但是A应用需要依赖1.0.0版本，但是B应用需要依赖1.0.1版本。这两个版本中都有一个类是com.hollis.Test.class。如果采用默认的双亲委派类加载机制，那么无法加载多个相同的类。

所以，Tomcat破坏了**双亲委派原则**，提供隔离的机制，为每个web容器单独提供一个WebAppClassLoader加载器。每一个WebAppClassLoader负责加载本身的目录下的class文件，加载不到时再交CommonClassLoader加载，这和双亲委派刚好相反。

好了，本期的50道JVM面试题就分享到这了，下期继续分享Java并发相关面试题，**点赞、关注** 不迷路，咱们下期见！

参考：

- [1]. [《深入理解Java虚拟机》](#)
- [2]. [《不看后悔》38个JVM精选问答，让你变成专家！](#)
- [3]. [《不看后悔》超赞！来一份常见 JVM 面试题+“答案”！](#)
- [4]. [JVM性能优化--类加载器,手动实现类的热加载](#)
- [5]. [炸了！一口气问了我18个JVM问！](#)
- [6]. [从实际案例聊聊Java应用的GC优化](#)
- [7]. [JVM系列\(二\): JVM 内存泄漏与内存溢出及问题排查](#)
- [8] . [《实战Java虚拟机性能优化》](#)
- [9]. [再清楚不过了，JVM逃逸分析，你一定得知道](#)
- [10]. [【JVM进阶之路】十：JVM调优总结](#)

关注公众号：三分恶

手册更新动态
即刻送达



添加个人微信：ThirdFighter

技术交流
加大佬云集微信群



第二部分:Java框架

一、Spring

有人说，“Java程序员都是Spring程序员”，老三不太赞成这个观点，但是这也可以看出Spring在Java世界里举足轻重的作用。

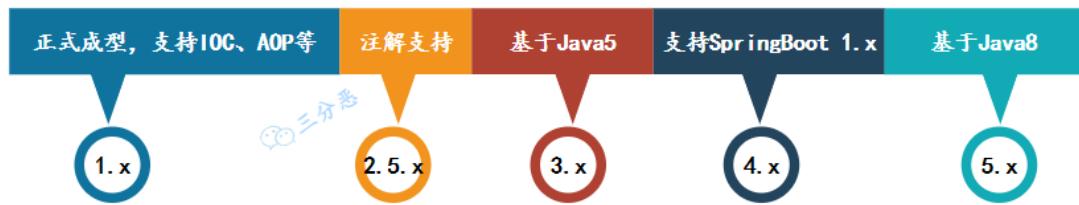
基础

1.Spring是什么？特性？有哪些模块？



一句话概括：**Spring** 是一个轻量级、非入侵式的控制反转 (IoC) 和面向切面 (AOP) 的框架。

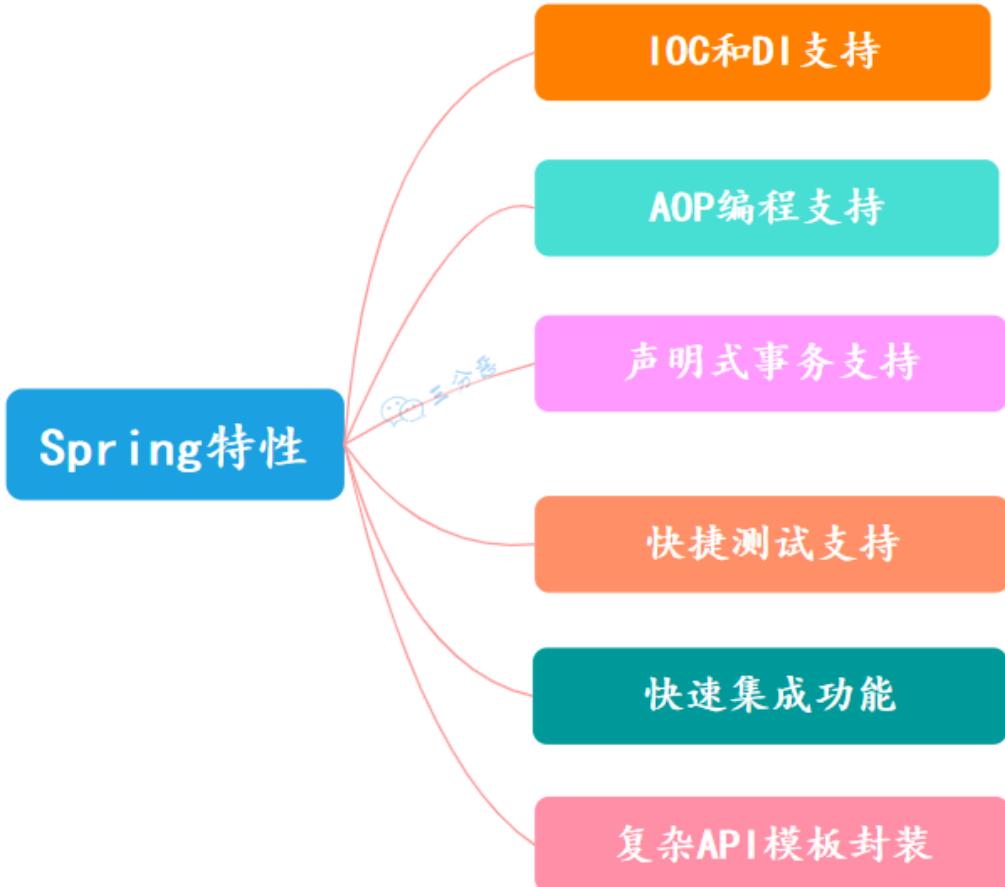
2003年，一个音乐家Rod Johnson决定发展一个轻量级的Java开发框架，**Spring** 作为Java战场的龙骑兵渐渐崛起，并淘汰了**EJB** 这个传统的重装骑兵。



到了现在，企业级开发的标配基本就是 **Spring5** + **Spring Boot 2** + **JDK 8**

Spring有哪些特性呢？

Spring有很多优点：



1. IOC 和 DI 的支持

Spring 的核心就是一个大的工厂容器，可以维护所有对象的创建和依赖关系，Spring 工厂用于生成 Bean，并且管理 Bean 的生命周期，实现 **高内聚低耦合** 的设计理念。

2. AOP 编程的支持

Spring 提供了 **面向切面编程**，可以方便的实现对程序进行权限拦截、运行监控等切面功能。

3. 声明式事务的支持

支持通过配置就来完成对事务的管理，而不需要通过硬编码的方式，以前重复的一些事务提交、回滚的 JDBC 代码，都可以不用自己写了。

4. 快捷测试的支持

Spring 对 Junit 提供支持，可以通过 **注解** 快捷地测试 Spring 程序。

5. 快速集成功能

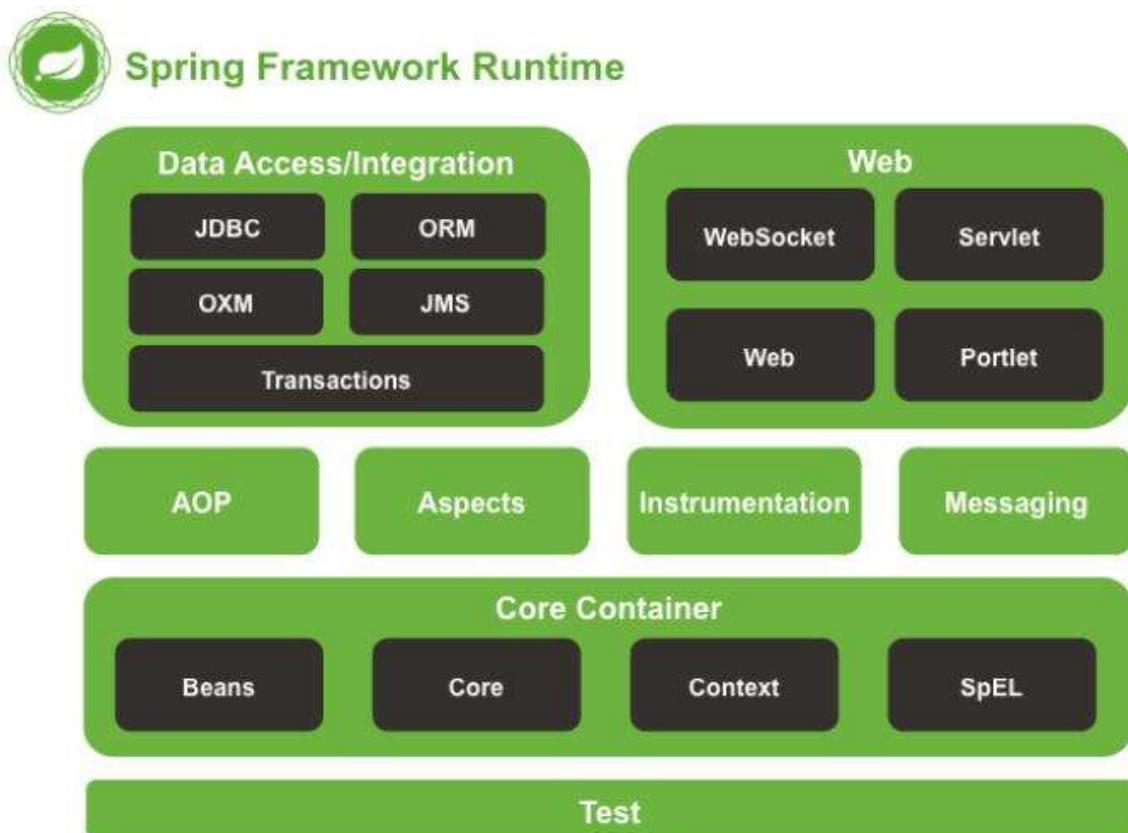
方便集成各种优秀框架，Spring 不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如：Struts、Hibernate、MyBatis、Quartz 等）的直接支持。

6. 复杂API模板封装

Spring 对 JavaEE 开发中非常难用的一些 API (JDBC、JavaMail、远程调用等) 都提供了模板化的封装，这些封装 API 的提供使得应用难度大大降低。

2.Spring有哪些模块呢？

Spring 框架是分模块存在，除了最核心的 **Spring Core Container** 是必要模块之外，其他模块都是 **可选**，大约有 20 多个模块。

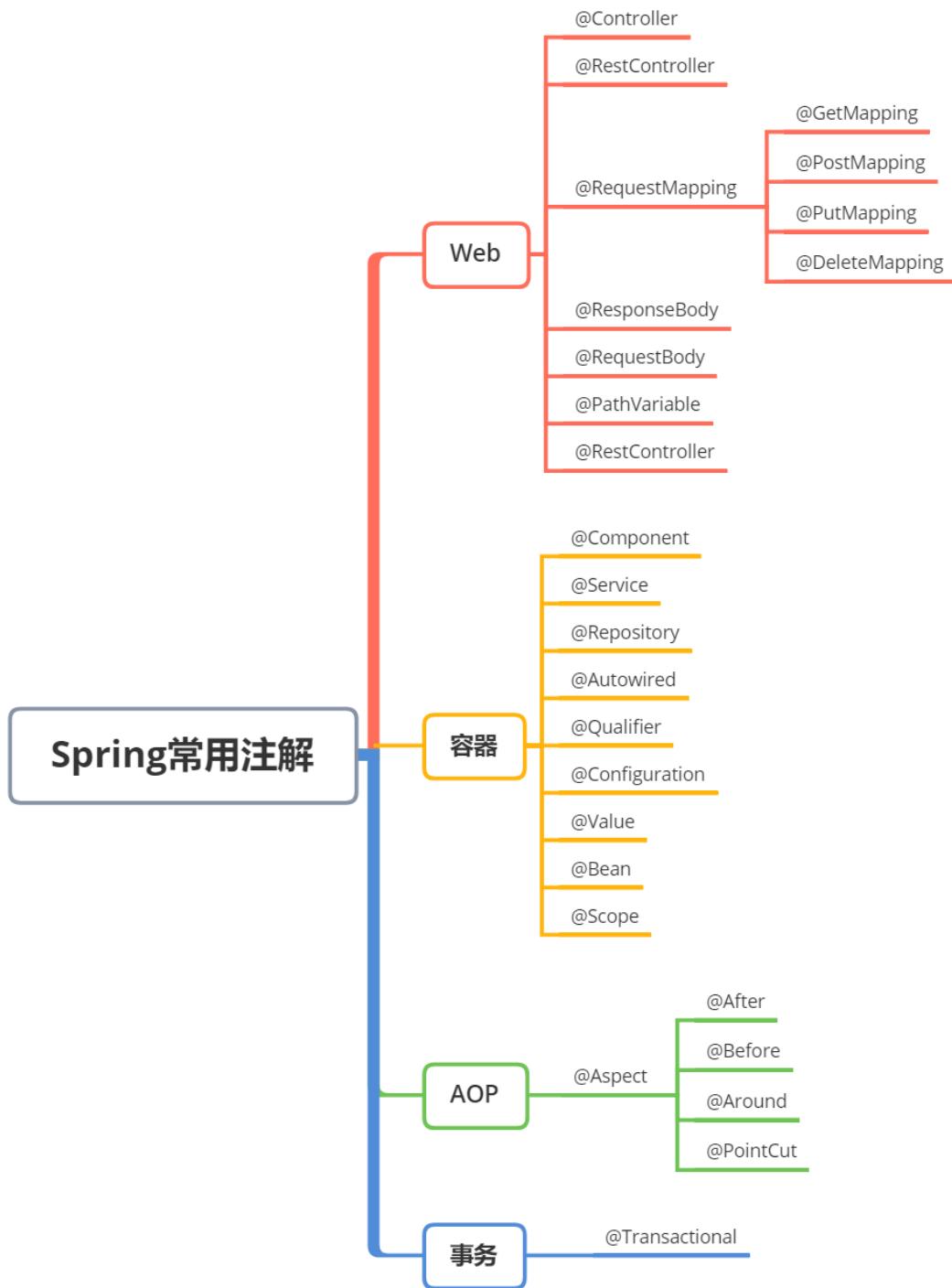


最主要的七大模块：

1. Spring Core : Spring 核心，它是框架最基础的部分，提供 IOC 和依赖注入 DI 特性。
2. Spring Context : Spring 上下文容器，它是 BeanFactory 功能加强的一个子接口。
3. Spring Web : 它提供 Web 应用开发的支持。
4. Spring MVC : 它针对 Web 应用中 MVC 思想的实现。
5. Spring DAO : 提供对 JDBC 抽象层，简化了 JDBC 编码，同时，编码更具有健壮性。
6. Spring ORM : 它支持用于流行的 ORM 框架的整合，比如：Spring + Hibernate、Spring + iBatis、Spring + JDO 的整合等。
7. Spring AOP : 即面向切面编程，它提供了与 AOP 联盟兼容的编程实现。

3.Spring有哪些常用注解呢？

Spring有很多模块，甚至广义的SpringBoot、SpringCloud也算是Spring的一部分，我们来分模块，按功能来看一下一些常用的注解：



Web:

- **@Controller:** 组合注解（组合了@Component注解），应用在MVC层（控制层）。

- **@RestController:** 该注解为一个组合注解，相当于@Controller和@ResponseBody的组合，注解在类上，意味着，该Controller的所有方法都默认加上了@ResponseBody。
- **@RequestMapping:** 用于映射Web请求，包括访问路径和参数。如果是Restful风格接口，还可以根据请求类型使用不同的注解：
 - @GetMapping
 - @PostMapping
 - @PutMapping
 - @DeleteMapping
- **@ResponseBody:** 支持将返回值放在response内，而不是一个页面，通常用户返回json数据。
- **@RequestBody:** 允许request的参数在request体中，而不是在直接连接在地址后面。
- **@PathVariable:** 用于接收路径参数，比如@RequestMapping(“/hello/{name}”)
 申明的路径，将注解放在参数中前，即可获取该值，通常作为Restful的接口实现方法。
- **@RestController:** 该注解为一个组合注解，相当于@Controller和@ResponseBody的组合，注解在类上，意味着，该Controller的所有方法都默认加上了@ResponseBody。

容器：

- **@Component:** 表示一个带注释的类是一个“组件”，成为Spring管理的Bean。当使用基于注解的配置和类路径扫描时，这些类被视为自动检测的候选对象。同时@Component还是一个元注解。
- **@Service:** 组合注解（组合了@Component注解），应用在service层（业务逻辑层）。
- **@Repository:** 组合注解（组合了@Component注解），应用在dao层（数据访问层）。
- **@Autowired:** Spring提供的工具（由Spring的依赖注入工具（BeanPostProcessor、BeanFactoryPostProcessor）自动注入）。
- **@Qualifier:** 该注解通常跟 @Autowired 一起使用，当想对注入的过程做更多的控制，@Qualifier 可帮助配置，比如两个以上相同类型的 Bean 时 Spring 无法抉择，用到此注解
- **@Configuration:** 声明当前类是一个配置类（相当于一个Spring配置的xml文件）

- **@Value:** 可用在字段，构造器参数跟方法参数，指定一个默认值，支持 #{} 跟 \${} 两个方式。一般将 SpringBoot 中的 application.properties 配置的属性值赋值给变量。
- **@Bean:** 注解在方法上，声明当前方法的返回值为一个 Bean。返回的 Bean 对应的类中可以定义 init() 方法和 destroy() 方法，然后在 @Bean(initMethod="init",destroyMethod="destroy") 定义，在构造之后执行 init，在销毁之前执行 destroy。
- **@Scope:** 定义我们采用什么模式去创建 Bean（方法上，得有 @Bean）其设置类型包括： Singleton、Prototype、Request、Session、GlobalSession。

AOP:

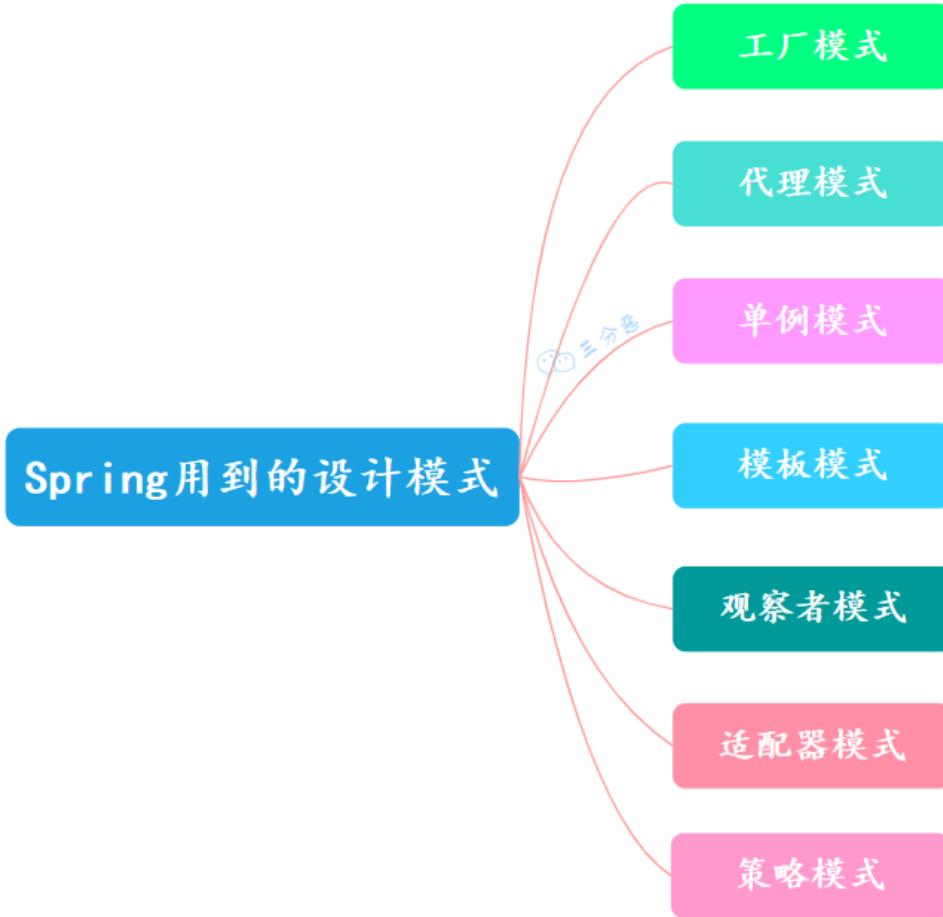
- **@Aspect:** 声明一个切面（类上）使用 @After、@Before、@Around 定义建言（advice），可直接将拦截规则（切点）作为参数。
 - **@After** : 在方法执行之后执行（方法上）。
 - **@Before** : 在方法执行之前执行（方法上）。
 - **@Around** : 在方法执行之前与之后执行（方法上）。
 - **@PointCut** : 声明切点 在 java 配置类中使用 @EnableAspectJAutoProxy 注解开启 Spring 对 AspectJ 代理的支持（类上）。

事务：

- **@Transactional:** 在要开启事务的方法上使用 @Transactional 注解，即可声明式开启事务。

4. Spring 中应用了哪些设计模式呢？

Spring 框架中广泛使用了不同类型的设计模式，下面我们来看看到底有哪些设计模式？



1. 工厂模式 : Spring 容器本质是一个大工厂， 使用工厂模式通过 BeanFactory、 ApplicationContext 创建 bean 对象。
2. 代理模式 : Spring AOP 功能功能就是通过代理模式来实现的， 分为动态代理和 静态代理。
3. 单例模式 : Spring 中的 Bean 默认都是单例的， 这样有利于容器对Bean的管理。
4. 模板模式 : Spring 中 JdbcTemplate、 RestTemplate 等以 Template结尾的对数据 库、 网络等等进行操作的模板类， 就使用到了模板模式。
5. 观察者模式 : Spring 事件驱动模型就是观察者模式很经典的一个应用。
6. 适配器模式 : Spring AOP 的增强或通知 (Advice) 使用到了适配器模式、 Spring MVC 中也是用到了适配器模式适配 Controller。
7. 策略模式 : Spring中有一个Resource接口， 它的不同实现类， 会根据不同的策略 去访问资源。

IOC

5.说一说什么是IOC? 什么是DI?

Java 是面向对象的编程语言，一个个实例对象相互合作组成了业务逻辑，原来，我们都是在代码里创建对象和对象的依赖。

所谓的 **IOC**（控制反转）：就是由容器来负责控制对象的生命周期和对象间的关系。以前是我们想要什么，就自己创建什么，现在是我们需要什么，容器就给我们送来什么。

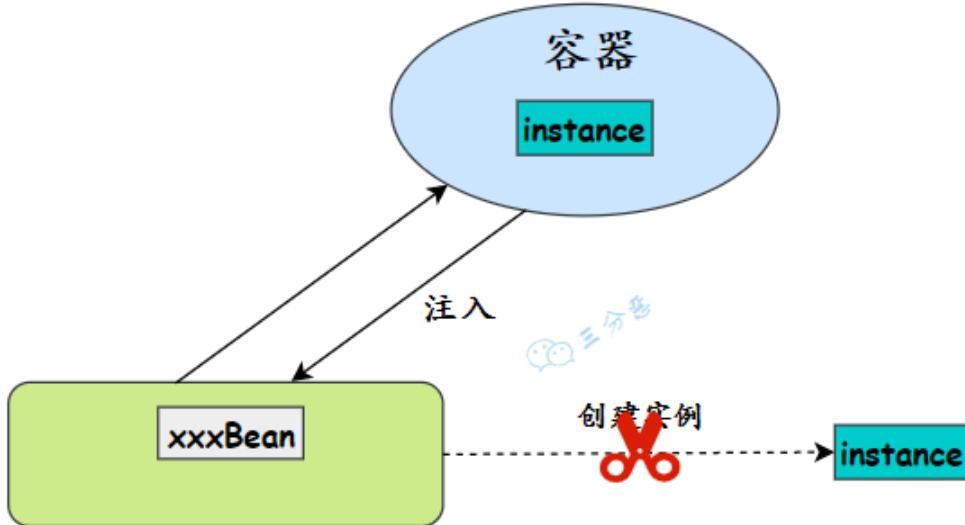
引入 **IOC** 之前，做饭——自己买菜、烧火、煮饭



引入 **IOC** 之后，饭来张口



也就是说，控制对象生命周期的不再是引用它的对象，而是容器。对具体对象，以前是它控制其它对象，现在所有对象都被容器控制，所以这就叫 **控制反转**。



DI（依赖注入）：指的是容器在实例化对象的时候把它依赖的类注入给它。有的说法IOC和DI是一回事，有的说法是IOC是思想，DI是IOC的实现。

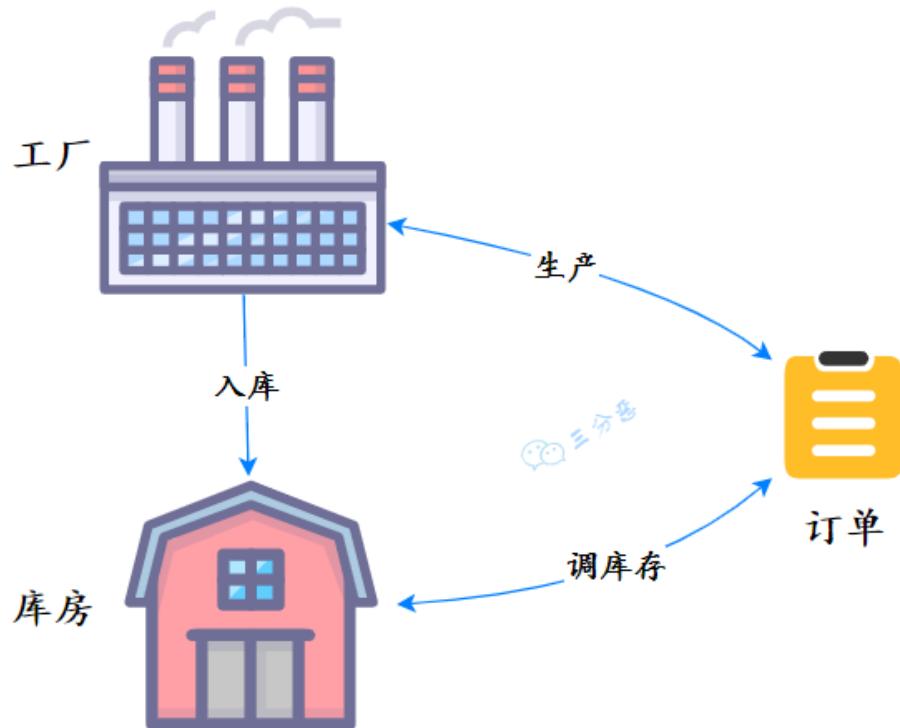
为什么要使用 **IOC** 呢？

最主要的是两个字 **解耦**，硬编码会造成对象间的过度耦合，使用IOC之后，我们可以不用关心对象间的依赖，专心开发应用就行。

6.能简单说一下Spring IOC的实现机制吗？

PS:这道题老三在面试中被问到过，问法是“**你有自己实现过简单的Spring吗？**”

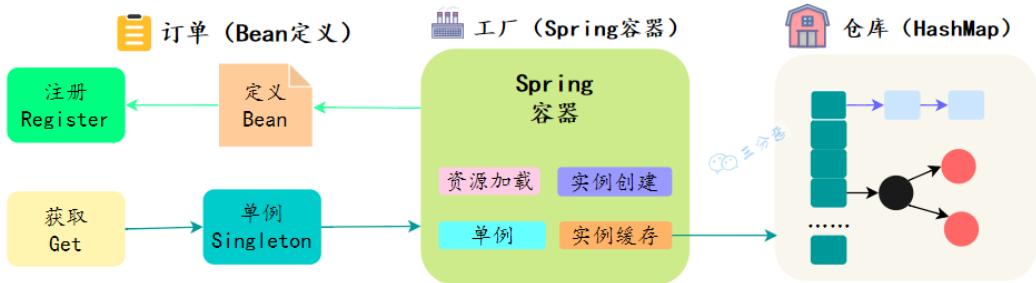
Spring的IOC本质就是一个大工厂，我们想想一个工厂是怎么运行的呢？



- **生产产品**: 一个工厂最核心的功能就是生产产品。在Spring里，不用Bean自己来实例化，而是交给Spring，应该怎么实现呢？——答案毫无疑问，**反射**。
那么这个厂子的生产管理是怎么做的？你应该也知道——**工厂模式**。
- **库存产品**: 工厂一般都是有库房的，用来库存产品，毕竟生产的产品不能立马就拉走。Spring我们都知道是一个容器，这个容器里存的就是对象，不能每次来取对象，都得现场来反射创建对象，得把创建出的对象存起来。
- **订单处理**: 还有最重要的一点，工厂根据什么来提供产品呢？订单。这些订单可能五花八门，有线上签签的、有到工厂签的、还有工厂销售上门签的……最后经过处理，指导工厂的出货。

在Spring里，也有这样的订单，它就是我们bean的定义和依赖关系，可以是xml形式，也可以是我们最熟悉的注解形式。

我们简单地实现一个mini版的Spring IOC:



Bean 定义:

Bean 通过一个配置文件定义，把它解析成一个类型。

- beans.properties
- 偷懒，这里直接用了最方便解析的 properties，这里直接用一个<key,value>类型的配置来代表 Bean 的定义，其中 key 是 beanName，value 是 class

```
1 | userDao:cn.fighter3.bean.UserDao
```

- BeanDefinition.java

bean 定义类，配置文件中 bean 定义对应的实体

```
1 | public class BeanDefinition {
2 |
3 |     private String beanName;
4 |
5 |     private Class beanClass;
6 |     //省略getter、setter
7 | }
```

- ResourceLoader.java

资源加载器，用来完成配置文件中配置的加载

```
1 | public class ResourceLoader {
2 |
3 |     public static Map<String, BeanDefinition>
4 |     getResource() {
5 |         Map<String, BeanDefinition> beanDefinitionMap =
6 |         new HashMap<>(16);
7 |         Properties properties = new Properties();
8 |         try {
```

```

7     InputStream inputStream =
ResourceLoader.class.getResourceAsStream("/beans.properties");
8         properties.load(inputStream);
9         Iterator<String> it =
properties.stringPropertyNames().iterator();
10        while (it.hasNext()) {
11            String key = it.next();
12            String className =
properties.getProperty(key);
13            BeanDefinition beanDefinition = new
BeanDefinition();
14            beanDefinition.setBeanName(key);
15            Class clazz = Class.forName(className);
16            beanDefinition.setBeanClass(clazz);
17            beanDefinitionMap.put(key,
beanDefinition);
18        }
19        inputStream.close();
20    } catch (IOException | ClassNotFoundException e) {
21        e.printStackTrace();
22    }
23    return beanDefinitionMap;
24}
25
26}

```

- BeanRegister.java

对象注册器，这里用于单例bean的缓存，我们大幅简化，默认所有bean都是单例的。可以看到所谓单例注册，也很简单，不过是往HashMap里存对象。

```

1 public class BeanRegister {
2
3     //单例Bean缓存
4     private Map<String, Object> singletonMap = new
HashMap<>(32);
5
6     /**
7      * 获取单例Bean
8      *
9      * @param beanName bean名称

```

```

10     * @return
11     */
12     public Object getSingletonBean(String beanName) {
13         return singletonMap.get(beanName);
14     }
15
16     /**
17      * 注册单例bean
18      *
19      * @param beanName
20      * @param bean
21      */
22     public void registerSingletonBean(String beanName,
23                                     Object bean) {
24         if (singletonMap.containsKey(beanName)) {
25             return;
26         }
27         singletonMap.put(beanName, bean);
28     }
29 }
```

- BeanFactory.java



- 对象工厂，我们最核心的一个类，在它初始化的时候，创建了bean注册器，完成了资源的加载。
- 获取bean的时候，先从单例缓存中取，如果没有取到，就创建并注册一个bean

```
1 | public class BeanFactory {
```

```
2
3     private Map<String, BeanDefinition>
beanDefinitionMap = new HashMap<>();
4
5     private BeanRegister beanRegister;
6
7     public BeanFactory() {
8         //创建bean注册器
9         beanRegister = new BeanRegister();
10        //加载资源
11        this.beanDefinitionMap = new
ResourceLoader().getResource();
12    }
13
14    /**
15     * 获取bean
16     *
17     * @param beanName bean名称
18     * @return
19     */
20    public Object getBean(String beanName) {
21        //从bean缓存中取
22        Object bean =
beanRegister.getSingletonBean(beanName);
23        if (bean != null) {
24            return bean;
25        }
26        //根据bean定义, 创建bean
27        return
createBean(beanDefinitionMap.get(beanName));
28    }
29
30    /**
31     * 创建Bean
32     *
33     * @param beanDefinition bean定义
34     * @return
35     */
36    private Object createBean(BeanDefinition
beanDefinition) {
37        try {
38            Object bean =
beanDefinition.getBeanClass().newInstance();
```

```
39         //缓存bean
40
41     beanRegister.registerSingletonBean(beanDefinition.getBeanName(), bean);
42     return bean;
43 } catch (InstantiationException | IllegalAccessException e) {
44     e.printStackTrace();
45 }
46 return null;
47 }
```

- 测试

- UserDao.java

我们的Bean类，很简单

```
1 public class UserDao {
2
3     public void queryUserInfo(){
4         System.out.println("A good man.");
5     }
6 }
```

- 单元测试

```
1 public class ApiTest {
2     @Test
3     public void test_BeanFactory() {
4         //1.创建bean工厂(同时完成了加载资源、创建注册单例bean
5         //注册器的操作)
6         BeanFactory beanFactory = new BeanFactory();
7
8         //2.第一次获取bean (通过反射创建bean, 缓存bean)
9         UserDao userDao1 = (UserDao)
10        beanFactory.getBean("userDao");
11        userDao1.queryUserInfo();
12
13         //3.第二次获取bean (从缓存中获取bean)
```

```
12     UserDao userDao2 = (UserDao)
13         beanFactory.getBean("userDao");
14     }
15 }
```

- 运行结果

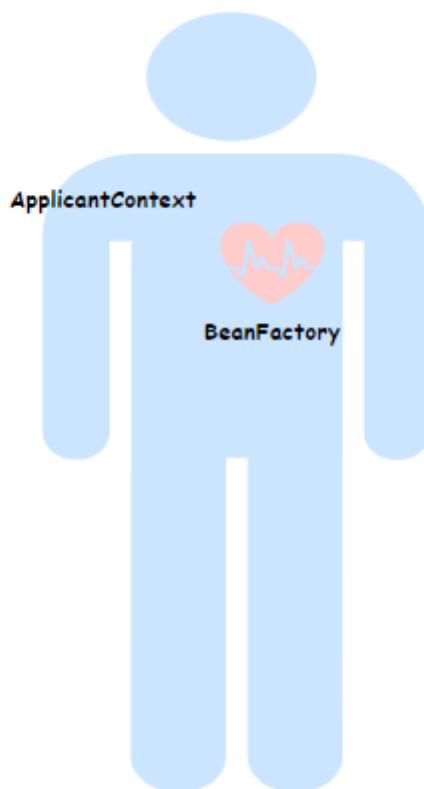
```
1 A good man.
2 A good man.
```

至此，我们一个乞丐+破船版的Spring就完成了，代码也比较完整，有条件的可以跑一下。

PS:因为时间+篇幅的限制，这个demo比较简陋，没有面向接口、没有解耦、边界检查、异常处理……健壮性、扩展性都有很大的不足，感兴趣可以学习参考[15]。

7.说说BeanFactory和ApplicantContext?

可以这么形容， BeanFactory是Spring的“心脏”， ApplicantContext是完整的“身躯”。

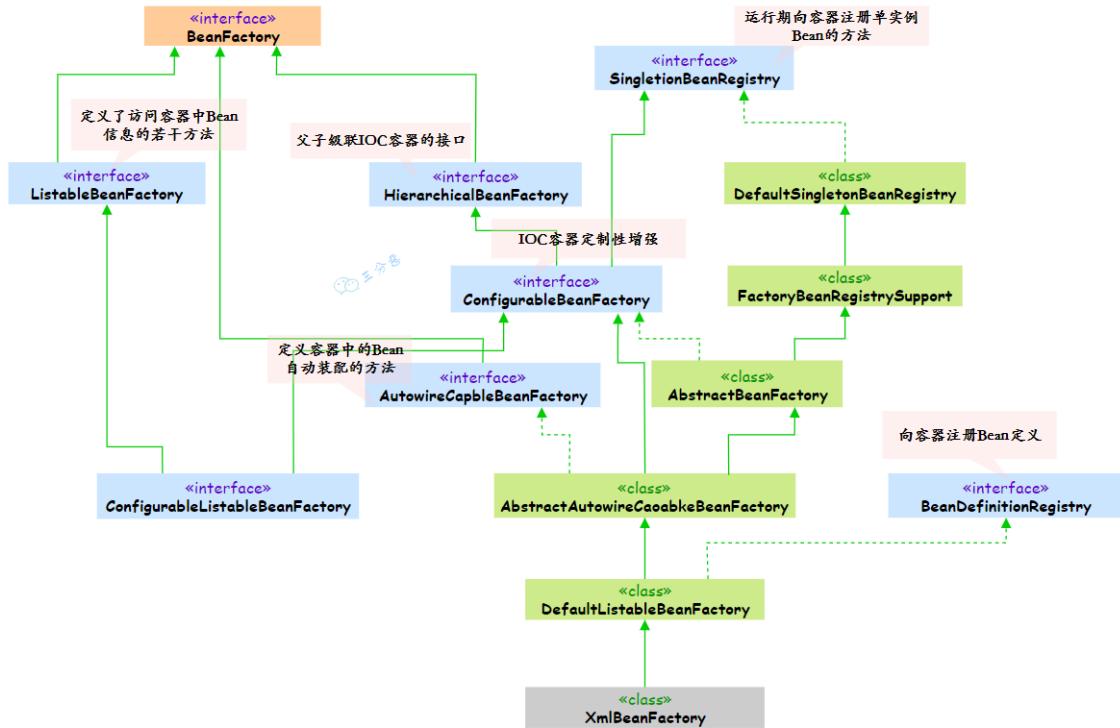


- BeanFactory（Bean工厂）是Spring框架的基础设施，面向Spring本身。
- ApplicationContext（应用上下文）建立在BeanFactoty基础上，面向使用Spring框架的开发者。

H5 BeanFactory 接口

BeanFactory是类的通用工厂，可以创建并管理各种类的对象。

Spring为BeanFactory提供了很多种实现，最常用的是XmlBeanFactory，但在Spring 3.2中已被废弃，建议使用XmlBeanDefinitionReader、DefaultListableBeanFactory。



BeanFactory接口位于类结构树的顶端，它最主要的方法就是getBean(String var1)，这个方法从容器中返回特定名称的Bean。

BeanFactory的功能通过其它的接口得到了不断的扩展，比如 AbstractAutowireCapableBeanFactory定义了将容器中的Bean按照某种规则（比如按名字匹配、按类型匹配等）进行自动装配的方法。

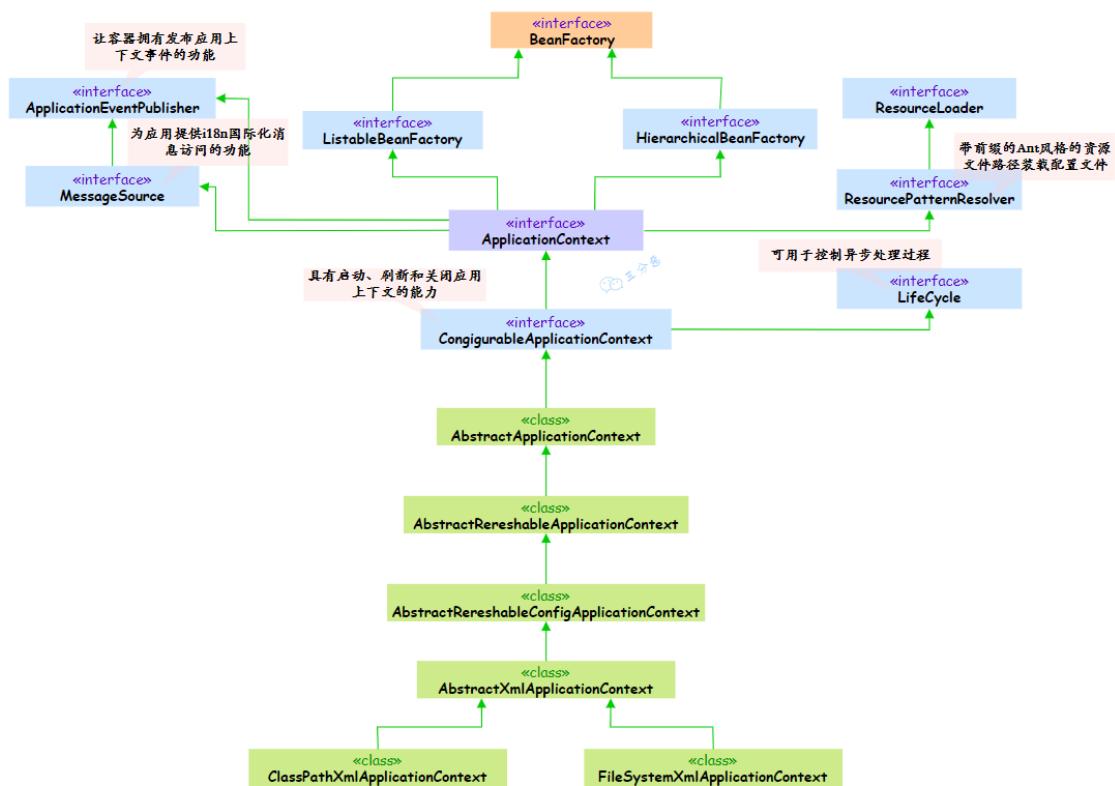
这里看一个 XMLBeanFactory（已过期） 获取bean 的例子：

```

1 public class HelloWorldApp{
2     public static void main(String[] args) {
3         BeanFactory factory = new XmlBeanFactory (new
4             ClassPathResource("beans.xml"));
5         HelloWorld obj = (HelloWorld)
6             factory.getBean("helloWorld");
7         obj.getMessage();
8     }
9 }
```

H5 ApplicationContext 接口

ApplicationContext由BeanFactory派生而来，提供了更多面向实际应用的功能。可以说，使用BeanFactory就是手动档，使用ApplicationContext就是自动档。



ApplicationContext 继承了HierachicalBeanFactory和ListableBeanFactory接口，在此基础上，还通过其他的接口扩展了BeanFactory的功能，包括：

- Bean instantiation/wiring
- Bean 的实例化/串联
- 自动的 BeanPostProcessor 注册
- 自动的 BeanFactoryPostProcessor 注册
- 方便的 MessageSource 访问（i18n）

- ApplicationEvent 的发布与 BeanFactory 懒加载的方式不同，它是预加载，所以，每一个 bean 都在 ApplicationContext 启动之后实例化

这是 ApplicationContext 的使用例子：

```
1 public class HelloWorldApp{  
2     public static void main(String[] args) {  
3         ApplicationContext context=new  
4             ClassPathXmlApplicationContext("beans.xml");  
5             HelloWorld obj = (HelloWorld)  
6                 context.getBean("helloWorld");  
7                 obj.getMessage();  
8             }  
9 }
```

ApplicationContext 包含 BeanFactory 的所有特性，通常推荐使用前者。

8.你知道Spring容器启动阶段会干什么吗？

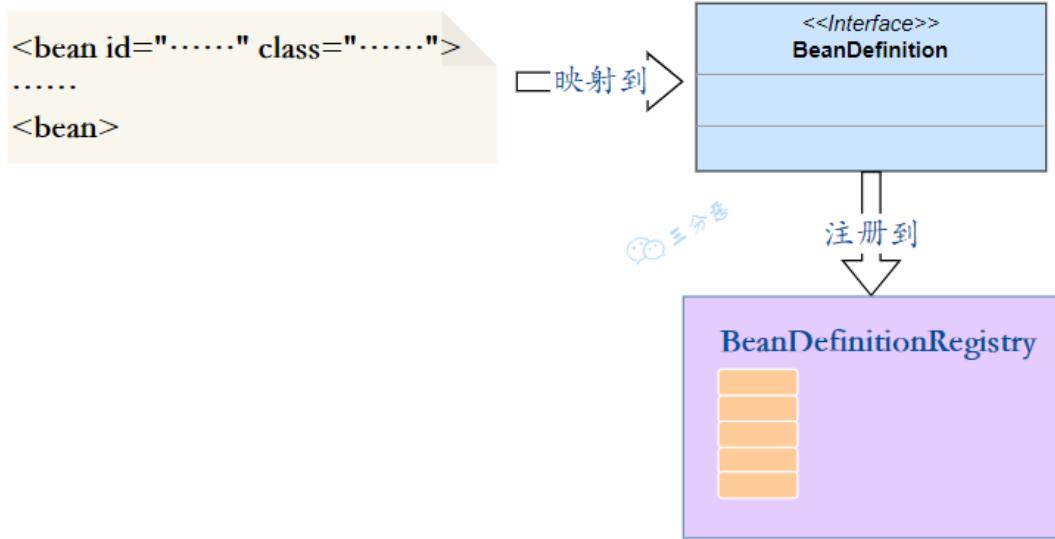
PS：这道题老三面试被问到过

Spring的IOC容器工作的过程，其实可以划分为两个阶段：**容器启动阶段**和**Bean实例化阶段**。

其中容器启动阶段主要做的工作是加载和解析配置文件，保存到对应的Bean定义中。



容器启动开始，首先会通过某种途径加载Congiguration MetaData，在大部分情况下，容器需要依赖某些工具类（BeanDefinitionReader）对加载的Congiguration MetaData进行解析和分析，并将分析后的信息组为相应的BeanDefinition。



最后把这些保存了Bean定义必要信息的BeanDefinition，注册到相应的BeanDefinitionRegistry，这样容器启动就完成了。

9.能说一下Spring Bean生命周期吗？

可以看看：[Spring Bean生命周期，好像人的一生。。](#)

在Spring中，基本容器BeanFactory和扩展容器ApplicationContext的实例化时机不太一样，BeanFactory采用的是延迟初始化的方式，也就是只有在第一次getBean()的时候，才会实例化Bean； ApplicationContext启动之后会实例化所有的Bean定义。

Spring IOC 中Bean的生命周期大致分为四个阶段： 实例化（Instantiation）、属性赋值（Populate）、 初始化（Initialization）、 销毁（Destruction）。

Bean生命周期

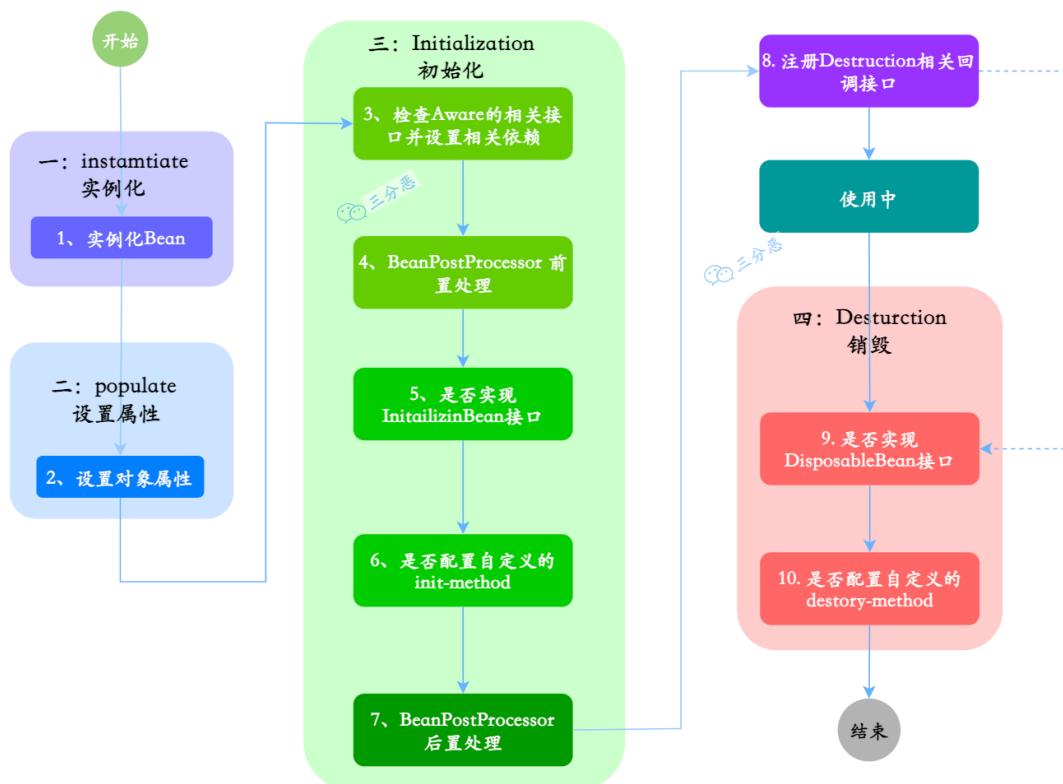


人的一生



我们再来看一个稍微详细一些的过程：

- 实例化：第1步，实例化一个Bean对象
- 属性赋值：第2步，为Bean设置相关属性和依赖
- 初始化：初始化的阶段的步骤比较多，5、6步是真正的初始化，第3、4步为在初始化前执行，第7步在初始化后执行，初始化完成之后，Bean就可以被使用了
- 销毁：第8~10步，第8步其实也可以算到销毁阶段，但不是真正意义上的销毁，而是先在使用前注册了销毁的相关调用接口，为了后面第9、10步真正销毁Bean时再执行相应的方法



简单总结一下，Bean生命周期里初始化的过程相对步骤会多一些，比如前置、后置的处理。

最后通过一个实例来看一下具体的细节：



8.BeanPostProcessor #postProcessAfterInitialization方法

工作

Bean使用中

9.DisposableBean #destroy方法

挂了

10.自定义destroy方法

埋了

结束

- 定义一个 PersonBean 类，实现 DisposableBean , InitializingBean , BeanFactoryAware , BeanNameAware 这4个接口，同时还有自定义的 init-method 和 destroy-method 。

```
1 public class PersonBean implements InitializingBean,  
2     BeanFactoryAware, BeanNameAware, DisposableBean {  
3  
4     /**  
5      * 身份证号  
6      */  
7     private Integer no;  
8  
9     /**  
10    * 姓名  
11    */  
12    private String name;  
13  
14    public PersonBean() {
```

```
14         System.out.println("1.调用构造方法: 我出生了! ");
15     }
16
17     public Integer getNo() {
18         return no;
19     }
20
21     public void setNo(Integer no) {
22         this.no = no;
23     }
24
25     public String getName() {
26         return name;
27     }
28
29     public void setName(String name) {
30         this.name = name;
31         System.out.println("2.设置属性: 我的名字叫" + name);
32     }
33
34     @Override
35     public void setBeanName(String s) {
36         System.out.println("3.调用BeanNameAware#setBeanName方
法:我要上学了, 起了个学名");
37     }
38
39     @Override
40     public void setBeanFactory(BeanFactory beanFactory)
throws BeansException {
41         System.out.println("4.调用
BeanFactoryAware#setBeanFactory方法: 选好学校了");
42     }
43
44     @Override
45     public void afterPropertiesSet() throws Exception {
46
System.out.println("6.InitializingBean#afterPropertiesSet方
法: 入学登记");
47     }
48
49     public void init() {
50         System.out.println("7.自定义init方法: 努力上学ing");
51     }
```

```
52  
53     @Override  
54     public void destroy() throws Exception {  
55         System.out.println("9.DisposableBean#destroy方法: 平淡  
56         的一生落幕了");  
57     }  
58  
59     public void destroyMethod() {  
60         System.out.println("10.自定义destroy方法:睡了, 别想叫醒  
61         我");  
62     }  
63  
64     public void work(){  
65         System.out.println("Bean使用中: 工作, 只有对社会没有用的人  
66         才放假。。。");  
67     }  
68 }
```

- 定义一个 `MyBeanPostProcessor` 实现 `BeanPostProcessor` 接口。

```
1  public class MyBeanPostProcessor implements BeanPostProcessor  
2  {  
3  
4      @Override  
5      public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {  
6  
7          System.out.println("5.BeanPostProcessor.postProcessBeforeInit  
8          ialization方法: 到学校报名啦");  
9          return bean;  
10     }  
11  
12     @Override  
13     public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  
14  
15         System.out.println("8.BeanPostProcessor#postProcessAfterIniti  
16         alization方法: 终于毕业, 拿到毕业证啦!");  
17         return bean;  
18     }  
19 }
```

- 配置文件，指定 `init-method` 和 `destroy-method` 属性

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4
5          xsi:schemaLocation="http://www.springframework.org/schema/beans
6          http://www.springframework.org/schema/beans/spring-
7          beans.xsd">
8
9      <bean name="myBeanPostProcessor"
10         class="cn.fighter3.spring.life.MyBeanPostProcessor" />
11      <bean name="personBean"
12         class="cn.fighter3.spring.life.PersonBean"
13         init-method="init" destroy-method="destroyMethod">
14          <property name="idNo" value="80669865"/>
15          <property name="name" value="张铁钢" />
16      </bean>
17
18  </beans>
```

- 测试

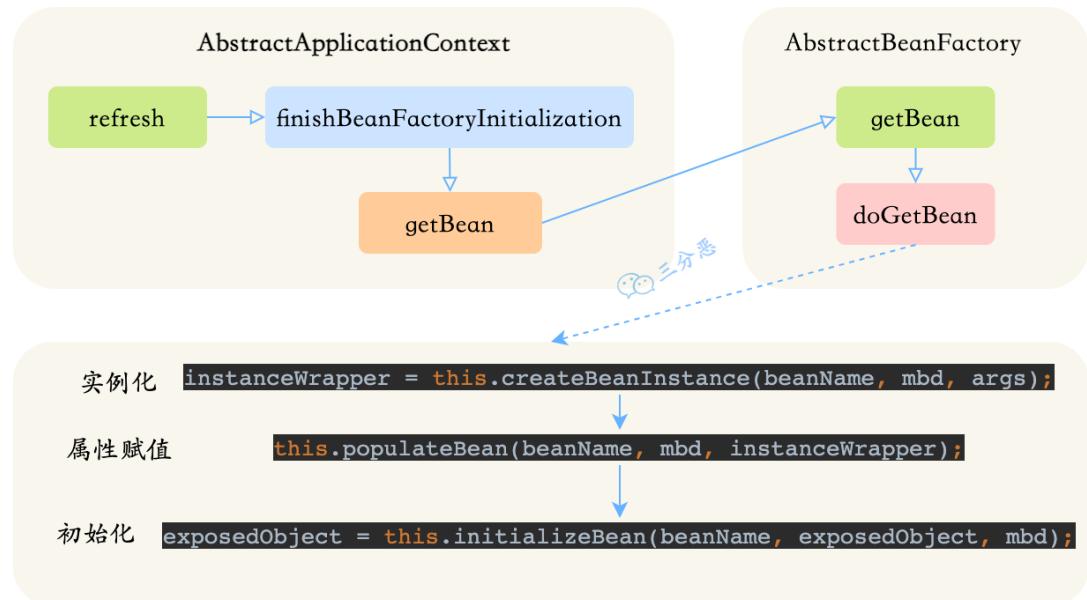
```

1  public class Main {
2
3      public static void main(String[] args) {
4          ApplicationContext context = new
5          ClassPathXmlApplicationContext("spring-config.xml");
6          PersonBean personBean = (PersonBean)
7          context.getBean("personBean");
8          personBean.work();
9          ((ClassPathXmlApplicationContext) context).destroy();
10     }
11 }
```

- 运行结果：

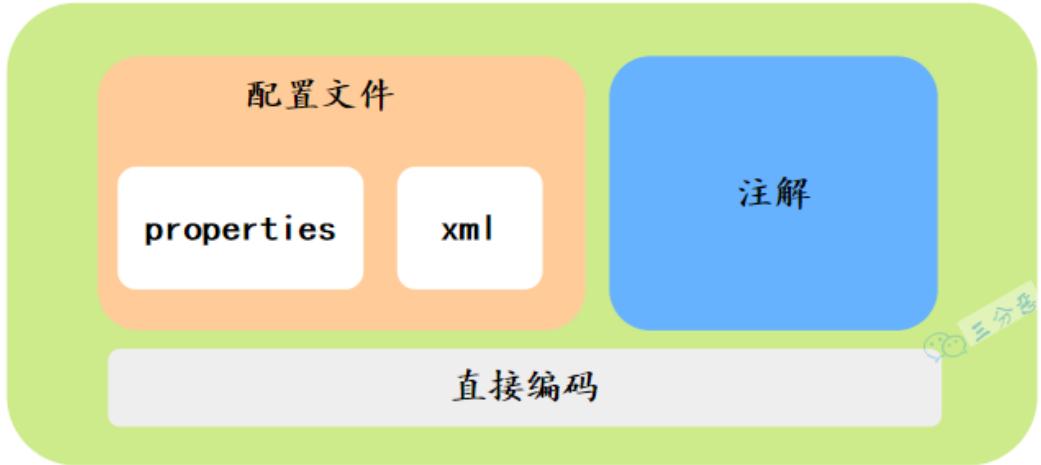
1. 调用构造方法：我出生了！
2. 设置属性：我的名字叫张铁钢
3. 调用BeanNameAware#setBeanName方法：我要上学了，起了个学名
4. 调用BeanFactoryAware#setBeanFactory方法：选好学校了
5. BeanPostProcessor#postProcessBeforeInitialization方法：到学校报名啦
6. InitializingBean#afterPropertiesSet方法：入学登记
7. 自定义init方法：努力上学ing
8. BeanPostProcessor#postProcessAfterInitialization方法：终于毕业，拿到毕业证啦！
9. Bean使用中：工作，只有对社会没有用的人才放假。。
10. DisposableBean#destroy方法：平淡的一生落幕了
11. 10. 自定义destroy方法：睡了，别想叫醒我

关于源码，Bean创建过程可以查看 `AbstractBeanFactory#doGetBean` 方法，在这个方法里可以看到Bean的实例化，赋值、初始化的过程，至于最终的销毁，可以看看 `ConfigurableApplicationContext#close()`。



10.Bean 定义和依赖定义有哪些方式？

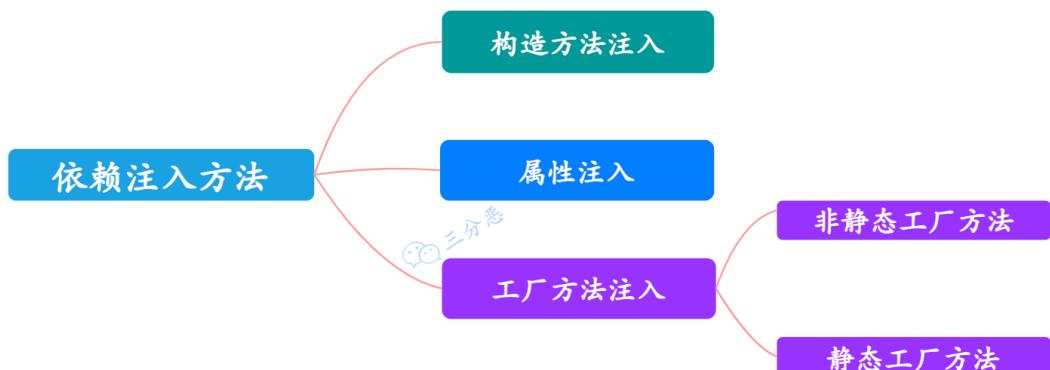
有三种方式：直接编码方式、配置文件方式、注解方式。



- 直接编码方式：我们一般接触不到直接编码的方式，但其实其它的方式最终都要通过直接编码来实现。
- 配置文件方式：通过xml、properties类型的配置文件，配置相应的依赖关系，Spring读取配置文件，完成依赖关系的注入。
- 注解方式：注解方式应该是我们用的最多的一种方式了，在相应的地方使用注解修饰，Spring会扫描注解，完成依赖关系的注入。

11. 有哪些依赖注入的方法？

Spring支持**构造方法注入**、**属性注入**、**工厂方法注入**,其中工厂方法注入，又可以分为**静态工厂方法注入**和**非静态工厂方法注入**。



• 构造方法注入

通过调用类的构造方法，将接口实现类通过构造方法变量传入

```

1 | public CatDaoImpl(String message){
2 |     this.message = message;
3 |

```

```
1 <bean id="CatDaoImpl" class="com.CatDaoImpl">
2   <constructor-arg value=" message "></constructor-arg>
3 </bean>
```

- 属性注入

通过Setter方法完成调用类所需依赖的注入

```
1 public class Id {
2   private int id;
3
4   public int getId() { return id; }
5
6   public void setId(int id) { this.id = id; }
7 }
```

```
1 <bean id="id" class="com.id ">
2   <property name="id" value="123"></property>
3 </bean>
```

- 工厂方法注入

- 静态工厂注入

静态工厂顾名思义，就是通过调用静态工厂的方法来获取自己需要的对象，为了让 Spring 管理所有对象，我们不能直接通过"工程类.静态方法()"来获取对象，而是依然通过 Spring 注入的形式获取：

```
1 public class DaoFactory { //静态工厂
2
3   public static final FactoryDao
4     getStaticFactoryDaoImpl(){
5       return new StaticFacotryDaoImpl();
6     }
7
8   public class SpringAction {
9
10   //注入对象
11   private FactoryDao staticFactoryDao;
12 }
```

```
13 //注入对象的 set 方法
14 public void setStaticFactoryDao(FactoryDao
15 staticFactoryDao) {
16     this.staticFactoryDao = staticFactoryDao;
17 }
18 }
```

```
1 //factory-method="getStaticFactoryDaoImpl"指定调用哪个工厂
方法
2 <bean name="springAction" class=" SpringAction" >
3     <!--使用静态工厂的方法注入对象,对应下面的配置文件-->
4     <property name="staticFactoryDao"
5 ref="staticFactoryDao"></property>
6 </bean>
7
8     <!--此处获取对象的方式是从工厂类中获取静态方法-->
9 <bean name="staticFactoryDao" class="DaoFactory"
factory-method="getStaticFactoryDaoImpl"></bean>
```

- 非静态工厂注入

非静态工厂，也叫实例工厂，意思是工厂方法不是静态的，所以我们需要首先 new 一个工厂实例，再调用普通的实例方法。

```
1 //非静态工厂
2 public class DaoFactory {
3     public FactoryDao getFactoryDaoImpl(){
4         return new FactoryDaoImpl();
5     }
6 }
7
8 public class SpringAction {
9     //注入对象
10    private FactoryDao factoryDao;
11
12    public void setFactoryDao(FactoryDao factoryDao) {
13        this.factoryDao = factoryDao;
14    }
15 }
```

```
1 <bean name="springAction" class="SpringAction">
2   <!--使用非静态工厂的方法注入对象,对应下面的配置文件-->
3   <property name="factoryDao" ref="factoryDao">
4   </property>
5 
6   <!--此处获取对象的方式是从工厂类中获取实例方法-->
7   <bean name="daoFactory" class="com.DaoFactory"></bean>
8 
9   <bean name="factoryDao" factory-bean="daoFactory"
factory-method="getFactoryDaoImpl"></bean>
```

12.Spring有哪些自动装配的方式？

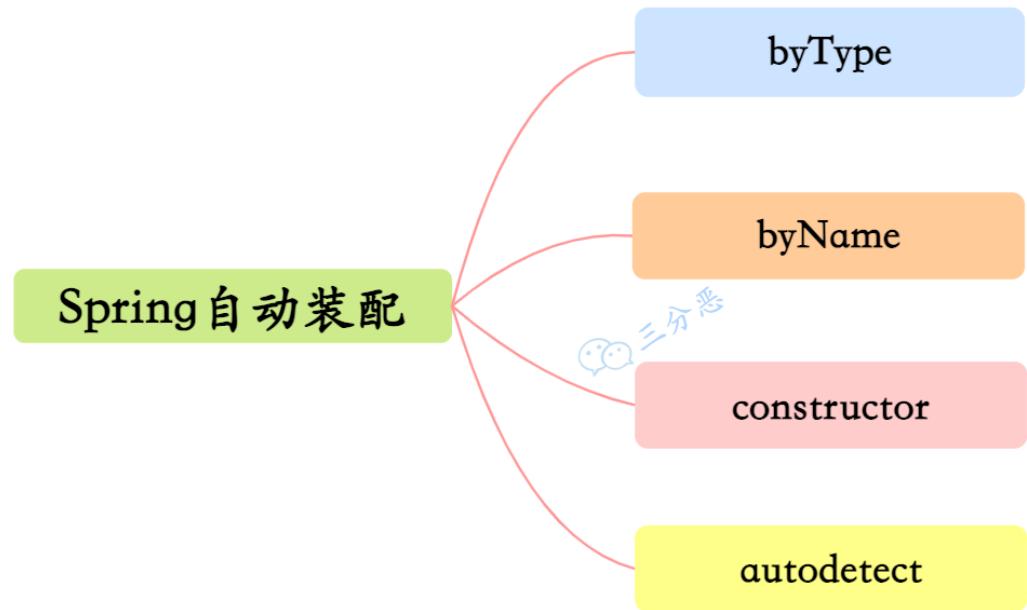
什么是自动装配？

Spring IOC容器知道所有Bean的配置信息，此外，通过Java反射机制还可以获知实现类的结构信息，如构造方法的结构、属性等信息。掌握所有Bean的这些信息后，Spring IOC容器就可以按照某种规则对容器中的Bean进行自动装配，而无须通过显式的方式进行依赖配置。

Spring提供的这种方式，可以按照某些规则进行Bean的自动装配，元素提供了一个指定自动装配类型的属性： autowire="<自动装配类型>"

Spring提供了哪几种自动装配类型？

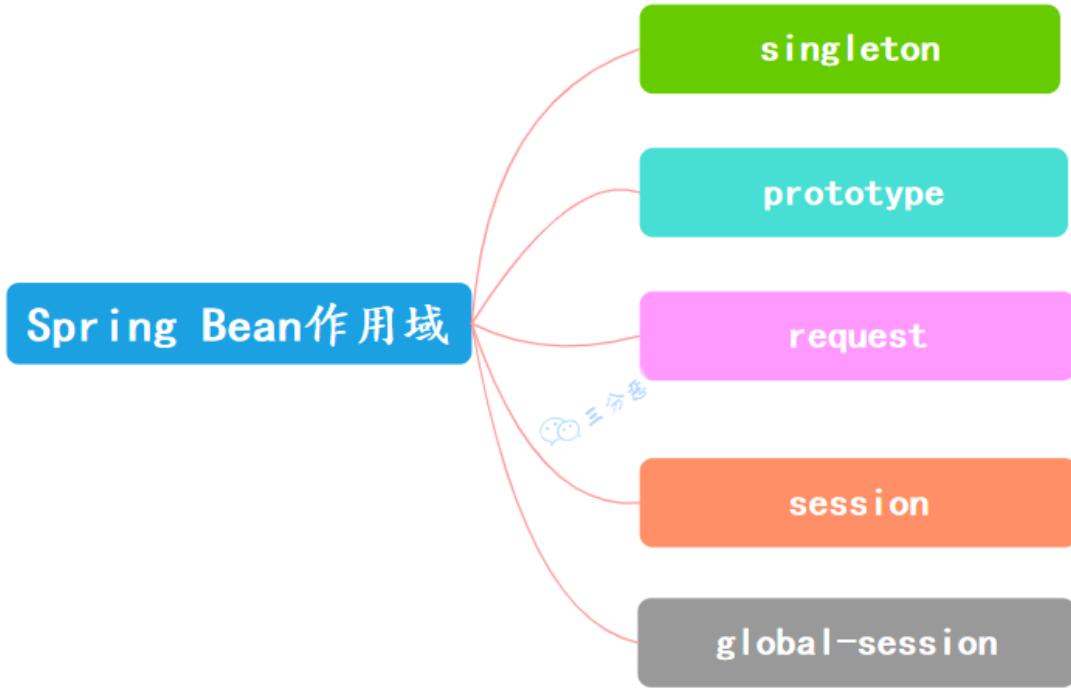
Spring提供了4种自动装配类型：



- **byName**：根据名称进行自动匹配，假设Boss又一个名为car的属性，如果容器中刚好有一个名为car的bean，Spring就会自动将其装配给Boss的car属性
- **byType**：根据类型进行自动匹配，假设Boss有一个Car类型的属性，如果容器中刚好有一个Car类型的Bean，Spring就会自动将其装配给Boss这个属性
- **constructor**：与 byType类似，只不过它是针对构造函数注入而言的。如果Boss有一个构造函数，构造函数包含一个Car类型的入参，如果容器中有一个Car类型的Bean，则Spring将自动把这个Bean作为Boss构造函数的入参；如果容器中没有找到和构造函数入参匹配类型的Bean，则Spring将抛出异常。
- **autodetect**：根据Bean的自省机制决定采用byType还是constructor进行自动装配，如果Bean提供了默认的构造函数，则采用byType，否则采用constructor。

13.Spring 中的 Bean 的作用域有哪些？

Spring的Bean主要支持五种作用域：



- singleton : 在Spring容器仅存在一个Bean实例， Bean以单实例的方式存在，是Bean默认的作用域。
 - prototype : 每次从容器重调用Bean时，都会返回一个新的实例。

以下三个作用域于只在Web应用中适用：

- `request` : 每一次HTTP请求都会产生一个新的Bean，该Bean仅在当前HTTP Request内有效。
 - `session` : 同一个HTTP Session共享一个Bean，不同的HTTP Session使用不同的Bean。
 - `globalSession` : 同一个全局Session共享一个Bean，只用于基于Protlet的Web应用，Spring5中已经不存在了。

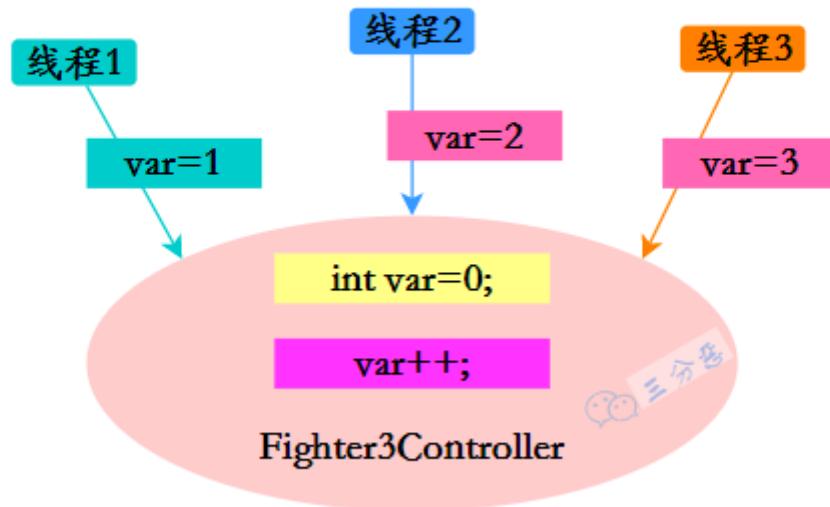
14. Spring 中的单例 Bean 会存在线程安全问题吗？

首先结论在这：Spring中的单例Bean不是线程安全的。

因为单例Bean，是全局只有一个Bean，所有线程共享。如果说单例Bean，是一个无状态的，也就是线程中的操作不会对Bean中的成员变量执行查询以外的操作，那么这个单例Bean是线程安全的。比如Spring mvc 的 Controller、Service、Dao等，这些Bean大多是无状态的，只关注于方法本身。

假如这个Bean是有状态的，也就是会对Bean中的成员变量进行写操作，那么可能就存在线程安全的问题。

正确结果: var=1;



单例Bean线程安全问题怎么解决呢？

常见的有这么些解决办法：

1. 将Bean定义为多例

这样每一个线程请求过来都会创建一个新的Bean，但是这样容器就不好管理Bean，不能这么办。

2. 在Bean对象中尽量避免定义可变的成员变量

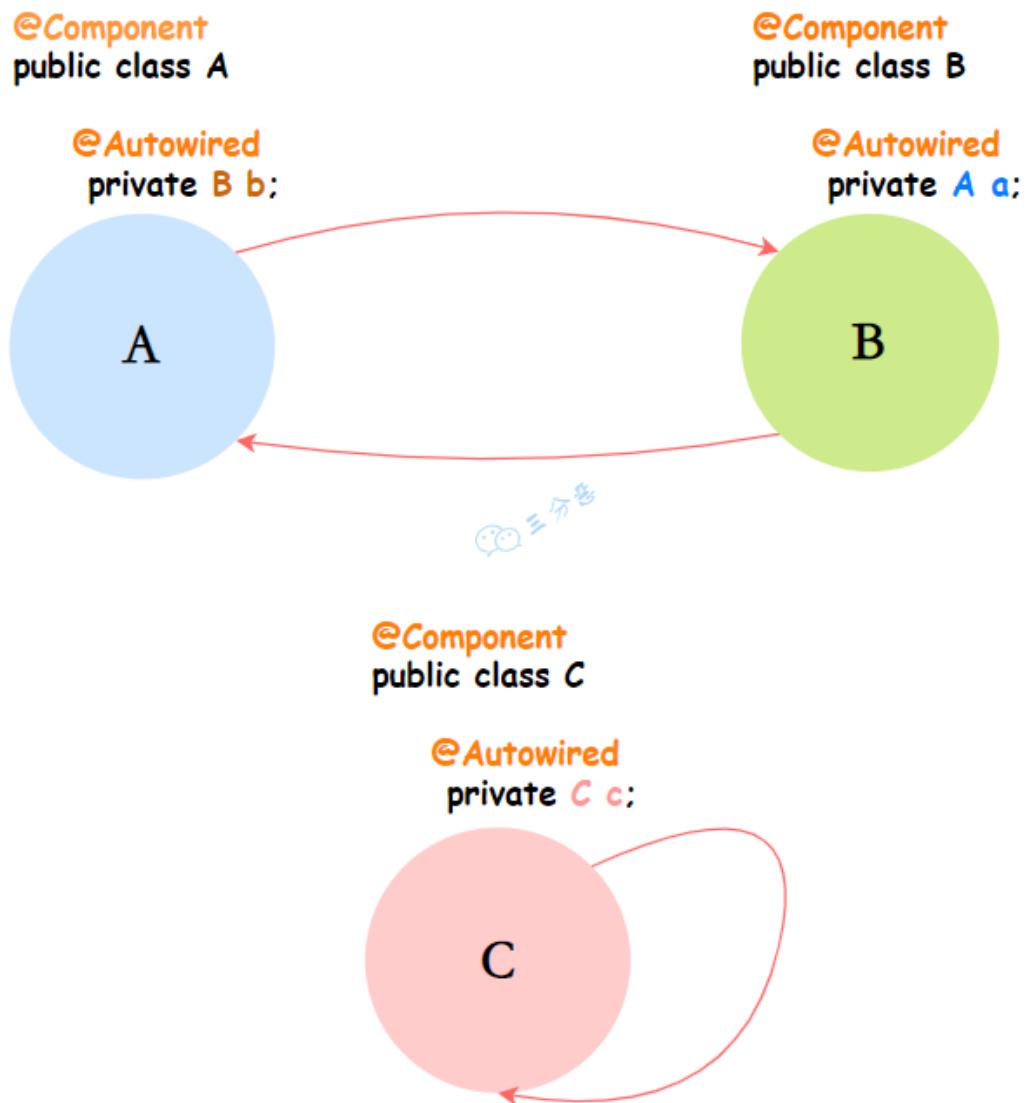
削足适履了属于是，也不能这么干。

3. 将Bean中的成员变量保存在ThreadLocal中☆

我们知道ThredLoca能保证多线程下变量的隔离，可以在类中定义一个ThreadLocal成员变量，将需要的可变成员变量保存在ThreadLocal里，这是推荐的一种方式。

15.说说循环依赖？

什么是循环依赖？



Spring 循环依赖：简单说就是自己依赖自己，或者和别的Bean相互依赖。



只有单例的Bean才存在循环依赖的情况，**原型**(Prototype)情况下，Spring会直接抛出异常。原因很简单，AB循环依赖，A实例化的时候，发现依赖B，创建B实例，创建B的时候发现需要A，创建A1实例……无限套娃，直接把系统干垮。

Spring可以解决哪些情况的循环依赖？

Spring不支持基于构造器注入的循环依赖，但是假如AB循环依赖，如果一个是构造器注入，一个是setter注入呢？

看看几种情形：

循环依赖		
依赖情况	依赖注入方式	是否支持
AB循环依赖	AB均采用构造器注入	X
AB循环依赖	AB均采用setter方法注入	✓
AB循环依赖	AB均采用属性自动注入	✓
AB循环依赖	A中注入的B为setter注入，B中注入的A为构造器注入	✓
AB循环依赖	B中注入的A为setter注入，A中注入的B为构造器注入	X

第四种可以而第五种不可以的原因是 Spring 在创建 Bean 时默认会根据自然排序进行创建，所以 A 会先于 B 进行创建。

所以简单总结，当循环依赖的实例都采用setter方法注入的时候，Spring可以支持，都采用构造器注入的时候，不支持，构造器注入和setter注入同时存在的时候，看天。

16.那Spring怎么解决循环依赖的呢？

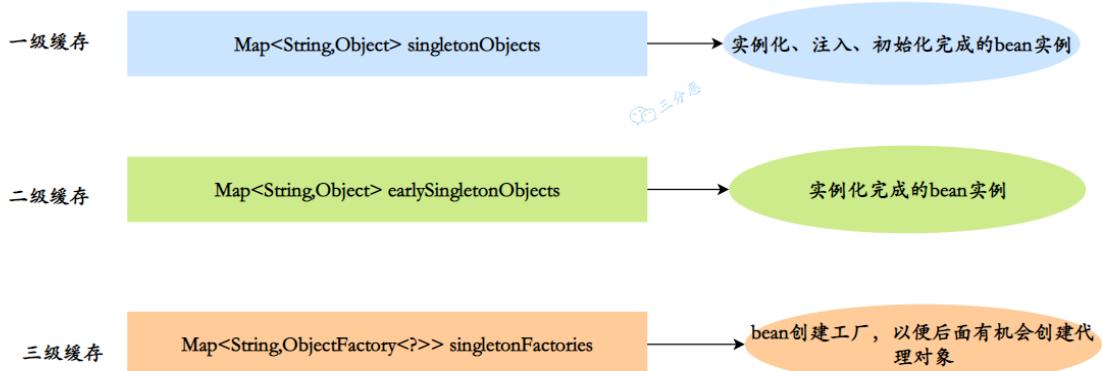
PS：其实正确答案是开发人员做好设计，别让Bean循环依赖，但是没办法，面试官不想听这个。

我们都知道，单例Bean初始化完成，要经历三步：



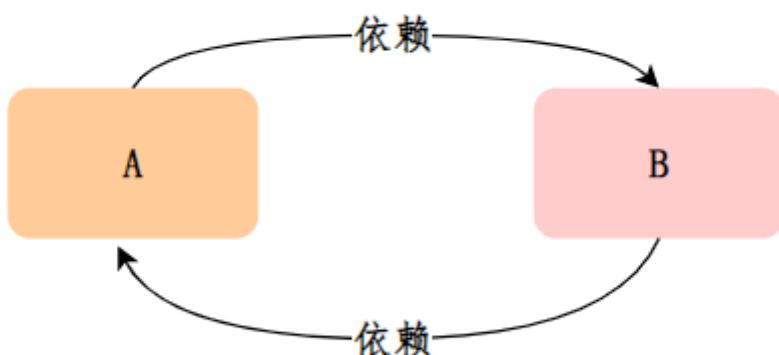
注入就发生在第二步，属性赋值，结合这个过程，Spring 通过三级缓存解决了循环依赖：

1. 一级缓存 : `Map<String, Object> singletonObjects`，单例池，用于保存实例化、属性赋值（注入）、初始化完成的 bean 实例
2. 二级缓存 : `Map<String, Object> earlySingletonObjects`，早期曝光对象，用于保存实例化完成的 bean 实例
3. 三级缓存 : `Map<String, ObjectFactory<?>> singletonFactories`，早期曝光对象工厂，用于保存 bean 创建工厂，以便于后面扩展有机会创建代理对象。



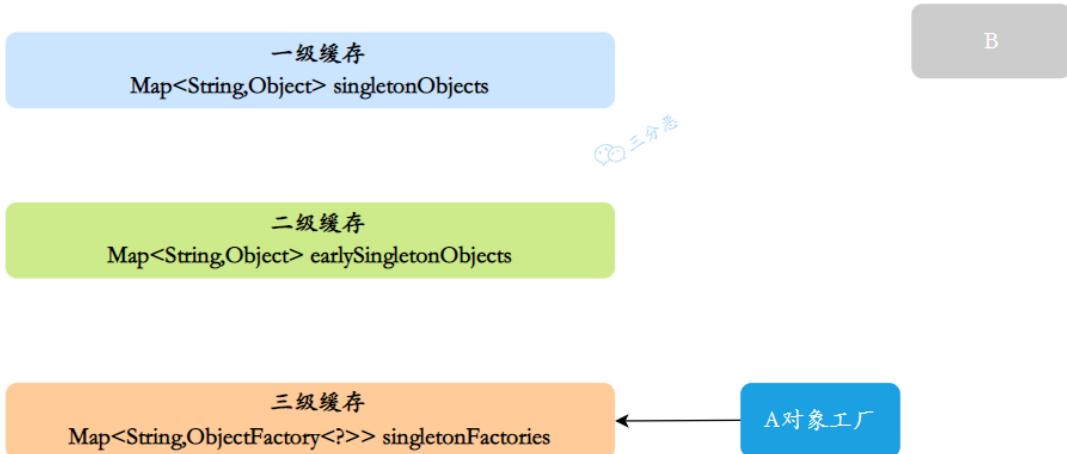
我们来看一下三级缓存解决循环依赖的过程：

当 A、B 两个类发生循环依赖时：

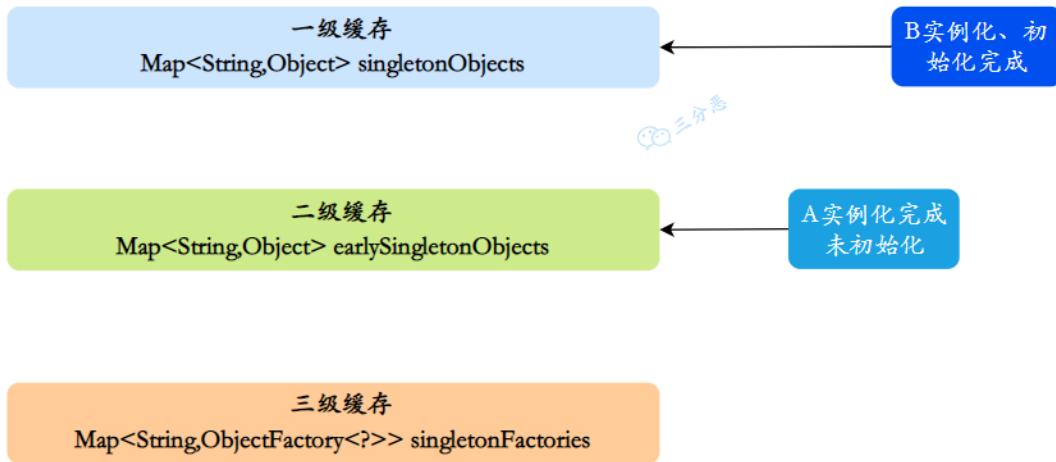


A实例的初始化过程：

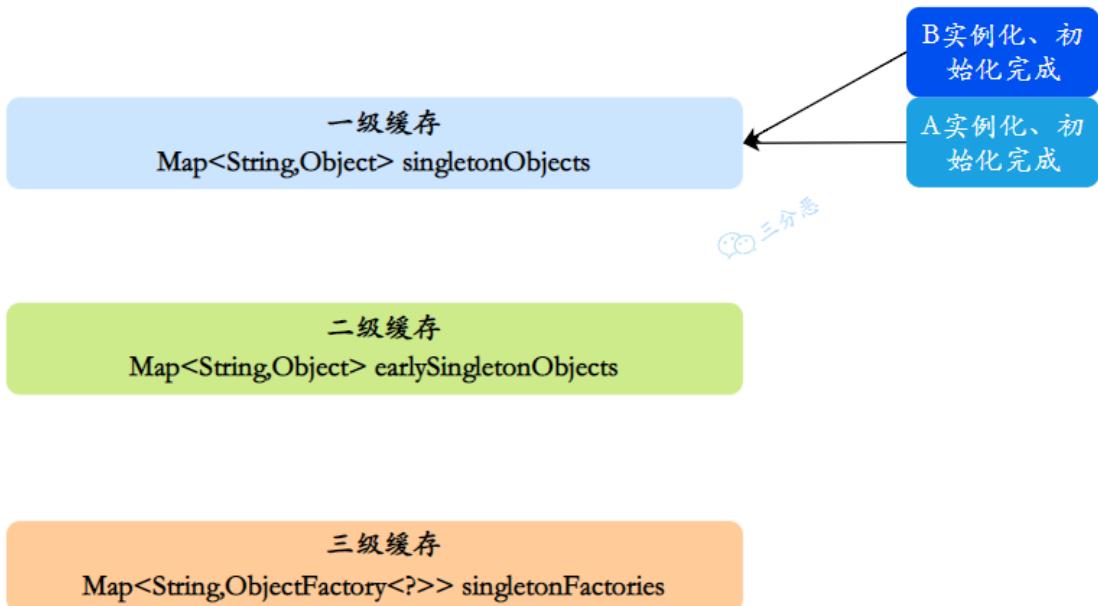
1. 创建A实例，实例化的时候把A对象工厂放入三级缓存，表示A开始实例化了，虽然我这个对象还不完整，但是先曝光出来让大家知道



2. A注入属性时，发现依赖B，此时B还没有被创建出来，所以去实例化B
3. 同样，B注入属性时发现依赖A，它就会从缓存里找A对象。依次从一级到三级缓存查询A，从三级缓存通过对象工厂拿到A，发现A虽然不太完善，但是存在，把A放入二级缓存，同时删除三级缓存中的A，此时，B已经实例化并且初始化完成，把B放入一级缓存。



4. 接着A继续属性赋值，顺利从一级缓存拿到实例化且初始化完成的B对象，A对象创建也完成，删除二级缓存中的A，同时把A放入一级缓存
5. 最后，一级缓存中保存着实例化、初始化都完成的A、B对象



所以，我们就知道为什么Spring能解决setter注入的循环依赖了，因为实例化和属性赋值是分开的，所以里面有操作的空间。如果都是构造器注入的话，那么都得在实例化这一步完成注入，所以自然是无法支持了。

17.为什么要三级缓存？二级不行吗？

不行，主要是为了**生成代理对象**。如果没有代理的情况下，使用二级缓存解决循环依赖也是OK的。但是如果存在代理，三级没有问题，二级就不行了。

因为三级缓存中放的是生成具体对象的匿名内部类，获取Object的时候，它可以生成代理对象，也可以返回普通对象。使用三级缓存主要是为了保证不管什么时候使用的都是一个对象。

假设只有二级缓存的情况，往二级缓存中放的显示一个普通的Bean对象，Bean初始化过程中，通过 BeanPostProcessor 去生成代理对象之后，覆盖掉二级缓存中的普通Bean对象，那么可能就导致取到的Bean对象不一致了。



18.@Autowired的实现原理？

实现@.Autowired的关键是： **AutowiredAnnotationBeanPostProcessor**

在Bean的初始化阶段，会通过Bean后置处理器来进行一些前置和后置的处理。

实现@Autowired的功能，也是通过后置处理器来完成的。这个后置处理器就是AutowiredAnnotationBeanPostProcessor。

- Spring在创建bean的过程中，最终会调用到doCreateBean()方法，在doCreateBean()方法中会调用populateBean()方法，来为bean进行属性填充，完成自动装配等工作。
- 在populateBean()方法中一共调用了两次后置处理器，第一次是为了判断是否需要属性填充，如果不需要进行属性填充，那么就会直接进行return，如果需要进行属性填充，那么方法就会继续向下执行，后面会进行第二次后置处理器的调用，这个时候，就会调用到AutowiredAnnotationBeanPostProcessor的postProcessPropertyValues()方法，在该方法中就会进行@Autowired注解的解析，然后实现自动装配。

```
1  /**
2  * 属性赋值
3  */
4  protected void populateBean(String beanName,
5      RootBeanDefinition mbd, @Nullable BeanWrapper bw) {
6      // .....
7      if (hasInstAwareBpps) {
8          if (pvs == null) {
9              pvs = mbd.getPropertyValues();
10         }
11
12         PropertyValues pvsToUse;
13         for(Iterator var9 =
14             this.getBeanPostProcessorCache().instantiationAware.iterator();
15             var9.hasNext(); pvs = pvsToUse) {
16             InstantiationAwareBeanPostProcessor bp
17             = (InstantiationAwareBeanPostProcessor)var9.next();
18             pvsToUse =
19                 bp.postProcessProperties((PropertyValues)pvs,
20                     bw.getWrappedInstance(), beanName);
21             if (pvsToUse == null) {
22                 if (filteredPds == null) {
23                     filteredPds =
24                         this.filterPropertyDescriptorsForDependencyCheck(bw,
25                             mbd.allowCaching);
26                 }
27             }
28         }
29     }
30 }
```

```

20 //调用
| InstantiationAwareBeanPostProcessor的
| postProcessPropertyValues()方法
21         pvsToUse =
|     bp.postProcessPropertyValues((PropertyValues)pvs,
|         filteredPds, bw.getWrappedInstance(), beanName);
22             if (pvsToUse == null) {
23                 return;
24             }
25         }
26     }
27 }
28 // .....
29 }
```

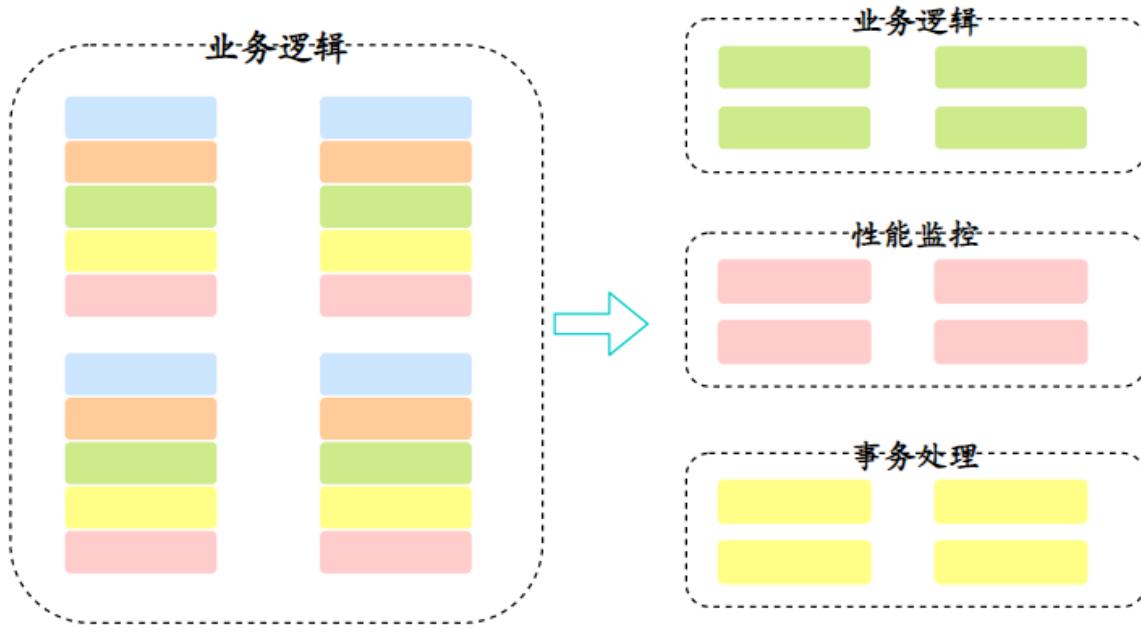
- postProcessorPropertyValues()方法的源码如下，在该方法中，会先调用findAutowiringMetadata()方法解析出bean中带有@Autowired注解、@Inject和@Value注解的属性和方法。然后调用metadata.inject()方法，进行属性填充。

```

1 public PropertyValues
2 postProcessProperties(PropertyValues pvs, Object bean,
3 String beanName) {
4     // @Autowired注解、@Inject和@Value注解的属性和方法
5     InjectionMetadata metadata =
6     this.findAutowiringMetadata(beanName, bean.getClass(),
7     pvs);
8
9     try {
10         // 属性填充
11         metadata.inject(bean, beanName, pvs);
12         return pvs;
13     } catch (BeanCreationException var6) {
14         throw var6;
15     } catch (Throwable var7) {
16         throw new BeanCreationException(beanName,
17             "Injection of autowired dependencies failed", var7);
18     }
19 }
```

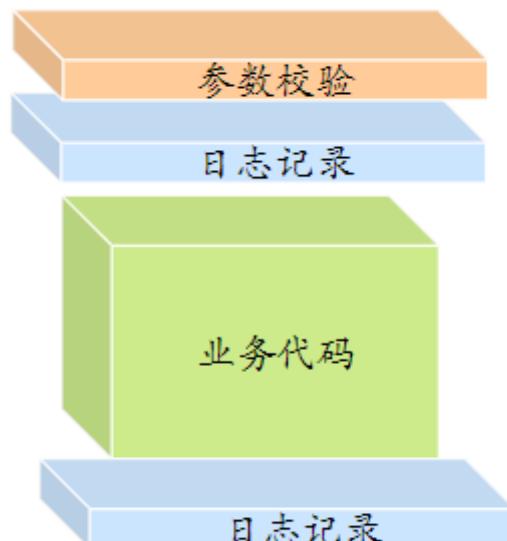
19.说说什么是AOP?

AOP: 面向切面编程。简单说，就是把一些业务逻辑中的相同的代码抽取到一个独立的模块中，让业务逻辑更加清爽。



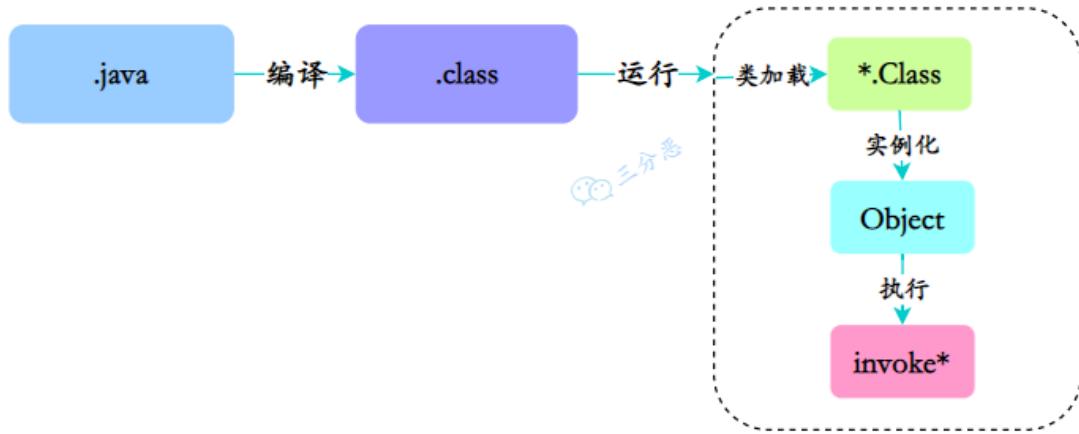
具体来说，假如我现在要crud写一堆业务，可是如何业务代码前后前后进行打印日志和参数的校验呢？

我们可以把 **日志记录** 和 **数据校验** 可重用的功能模块分离出来，然后在程序的执行的合适的地方动态地植入这些代码并执行。这样就简化了代码的书写。



业务逻辑代码中没有参和通用逻辑的代码，业务模块更简洁，只包含核心业务代码。实现了业务逻辑和通用逻辑的代码分离，便于维护和升级，降低了业务逻辑和通用逻辑的耦合性。

AOP 可以将遍布应用各处的功能分离出来形成可重用的组件。在编译期间、装载期间或运行期间实现在不修改源代码的情况下给程序动态添加功能。从而实现对业务逻辑的隔离，提高代码的模块化能力。



AOP 的核心其实就是**动态代理**，如果是实现了接口的话就会使用 JDK 动态代理，否则使用 CGLIB 代理，主要应用于处理一些具有横切性质的系统级服务，如日志收集、事务管理、安全检查、缓存、对象池管理等。

AOP有哪些核心概念？

- **切面 (Aspect)**：类是对物体特征的抽象，切面就是对横切关注点的抽象
- **连接点 (Joinpoint)**：被拦截到的点，因为 Spring 只支持方法类型的连接点，所以在 Spring 中连接点指的就是被拦截到的方法，实际上连接点还可以是字段或者构造器
- **切点 (Pointcut)**：对连接点进行拦截的定位
- **通知 (Advice)**：所谓通知指的就是指拦截到连接点之后要执行的代码，也可以称作**增强**
- **目标对象 (Target)**：代理的目标对象
- **织入 (Weaving)**：织入是将增强添加到目标类的具体连接点上的过程。
 - 编译期织入：切面在目标类编译时被织入
 - 类加载期织入：切面在目标类加载到JVM时被织入。需要特殊的类加载器，它可以在目标类被引入应用之前增强该目标类的字节码。

- 运行期织入：切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象动态地创建一个代理对象。Spring AOP就是以这种方式织入切面。

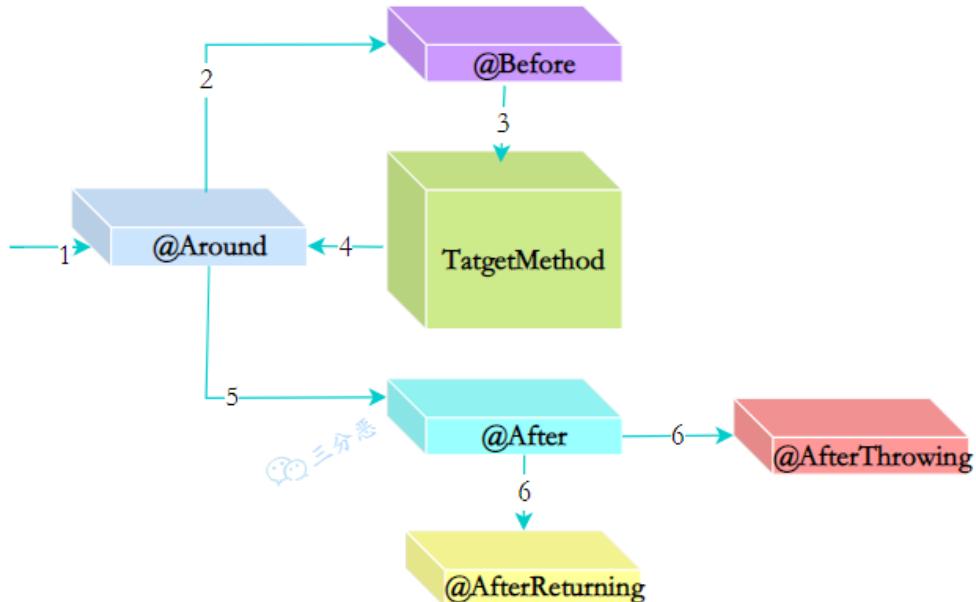
Spring采用运行期织入，而AspectJ采用编译期织入和类加载器织入。

- 引介 (introduction) :** 引介是一种特殊的增强，可以动态地为类添加一些属性和方法

AOP有哪些环绕方式？

AOP一般有**5种**环绕方式：

- 前置通知 (@Before)
- 返回通知 (@AfterReturning)
- 异常通知 (@AfterThrowing)
- 后置通知 (@After)
- 环绕通知 (@Around)



多个切面的情况下，可以通过 @Order 指定先后顺序，数字越小，优先级越高。

20.说说你平时有用到AOP吗？

PS：这道题老三的同事面试候选人的时候问到了，候选人说了一堆AOP原理，同事就势来一句，你能现场写一下AOP的应用吗？结果——场面一度很尴尬。虽然我对面试写这种百度就能出来的东西持保留意见，但是还是加上了这一问，毕竟招人最后都是要撸代码的。

这里给出一个小例子，SpringBoot项目中，利用AOP打印接口的入参和出参日志，以及执行时间，还是比较快捷的。

- 引入依赖：引入AOP依赖

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-
4       aop</artifactId>
5   </dependency>
```

- 自定义注解：自定义一个注解作为切点

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target({ElementType.METHOD})
3 @Documented
4 public @interface WebLog {
5 }
```

- 配置AOP切面：

- `@Aspect`: 标识切面
- `@Pointcut`: 设置切点，这里以自定义注解为切点，定义切点有很多其它种方式，自定义注解是比较常用的一种。
- `@Before`: 在切点之前织入，打印了一些入参信息
- `@Around`: 环绕切点，打印返回参数和接口执行时间

```
1 @Aspect
2 @Component
3 public class WebLogAspect {
4
5   private final static Logger logger      =
6   LoggerFactory.getLogger(WebLogAspect.class);
7
8   /**
9    * 以自定义 @WebLog 注解为切点
10   */
11
12   @Pointcut("@annotation(cn.fighter3.spring.aop_demo.WebLog)
13   ")
```

```
11 |     public void webLog() {}  
12 |  
13 |     /**  
14 |      * 在切点之前织入  
15 |      */  
16 |     @Before("webLog()")  
17 |     public void doBefore(JoinPoint joinPoint) throws  
18 |     Throwable {  
19 |         // 开始打印请求日志  
20 |         ServletRequestAttributes attributes =  
21 |         (ServletRequestAttributes)  
22 |         RequestContextHolder.getRequestAttributes();  
23 |         HttpServletRequest request =  
24 |         attributes.getRequest();  
25 |         // 打印请求相关参数  
26 |  
27 |         logger.info("===== Start =====");  
28 |         // 打印请求 url  
29 |         logger.info("URL : {}",  
30 |         request.getRequestURL().toString());  
31 |         // 打印 Http method  
32 |         logger.info("HTTP Method : {}",  
33 |         request.getMethod());  
34 |         // 打印调用 controller 的全路径以及执行方法  
35 |         logger.info("Class Method : {}.{}",  
36 |         joinPoint.getSignature().getDeclaringTypeName(),  
37 |         joinPoint.getSignature().getName());  
38 |         // 打印请求的 IP  
39 |         logger.info("IP : {}",  
40 |         request.getRemoteAddr());  
41 |         // 打印请求入参  
42 |         logger.info("Request Args : {}", new  
43 |         ObjectMapper().writeValueAsString(joinPoint.getArgs()));  
44 |     }  
45 |  
46 |     /**  
47 |      * 在切点之后织入  
48 |      * @throws Throwable  
49 |      */  
50 |     @After("webLog()")  
51 |     public void doAfter() throws Throwable {  
52 |         // 结束后打个分隔线，方便查看
```

```

42     logger.info("=====");
43     End =====);
44 }
45 /**
46 * 环绕
47 */
48 @Around("webLog()")
49 public Object doAround(ProceedingJoinPoint
proceedingJoinPoint) throws Throwable {
50     //开始时间
51     long startTime = System.currentTimeMillis();
52     Object result = proceedingJoinPoint.proceed();
53     // 打印出参
54     logger.info("Response Args : {}", new
ObjectMapper().writeValueAsString(result));
55     // 执行耗时
56     logger.info("Time-Consuming : {} ms",
System.currentTimeMillis() - startTime);
57     return result;
58 }
59
60 }

```

- 使用：只需要在接口上加上自定义注解

```

1 @GetMapping("/hello")
2 @WebLog(desc = "这是一个欢迎接口")
3 public String hello(String name){
4     return "Hello "+name;
5 }

```

- 执行结果：可以看到日志打印了入参、出参和执行时间

```

2022-04-17 09:04:22.515 INFO 39700 --- [nio-8080-exec-1] c.fighter3.spring.aop_demo.WebLogAspect : ===== Start =====
2022-04-17 09:04:22.515 INFO 39700 --- [nio-8080-exec-1] c.fighter3.spring.aop_demo.WebLogAspect : URL      : http://localhost:8080/fighter3/aop/demo/hello
2022-04-17 09:04:22.516 INFO 39700 --- [nio-8080-exec-1] c.fighter3.spring.aop_demo.WebLogAspect : HTTP Method : GET
2022-04-17 09:04:22.516 INFO 39700 --- [nio-8080-exec-1] c.fighter3.spring.aop_demo.WebLogAspect : Class Method : cn.fighter3.spring.aop_demo.controller.HelloController.hello
2022-04-17 09:04:22.517 INFO 39700 --- [nio-8080-exec-1] c.fighter3.spring.aop_demo.WebLogAspect : IP        : 0:0:0:0:0:0:0:1
2022-04-17 09:04:22.533 INFO 39700 --- [nio-8080-exec-1] c.fighter3.spring.aop_demo.WebLogAspect : Request Args : ["World"]
2022-04-17 09:04:22.541 INFO 39700 --- [nio-8080-exec-1] c.fighter3.spring.aop_demo.WebLogAspect : ===== End =====
2022-04-17 09:04:22.541 INFO 39700 --- [nio-8080-exec-1] c.fighter3.spring.aop_demo.WebLogAspect : Response Args : "Hello World"
2022-04-17 09:04:22.541 INFO 39700 --- [nio-8080-exec-1] c.fighter3.spring.aop_demo.WebLogAspect : Time-Consuming : 2142 ms

```

21.说说JDK 动态代理和 CGLIB 代理？

Spring的AOP是通过[动态代理](#)来实现的，动态代理主要有两种方式JDK动态代理和Cglib动态代理，这两种动态代理的使用和原理有些不同。

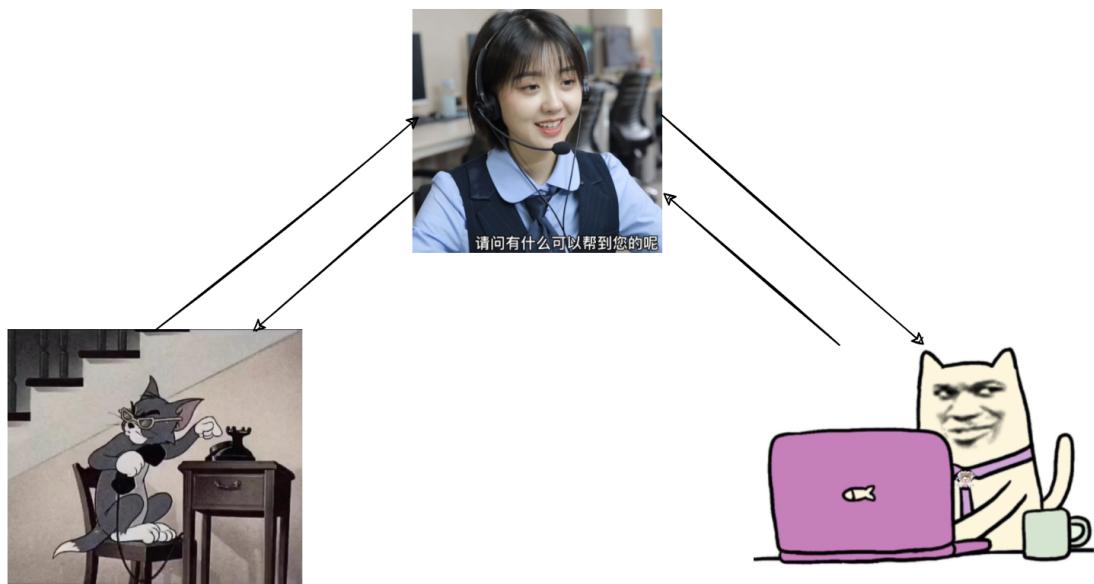
JDK 动态代理

1. Interface：对于 JDK 动态代理，目标类需要实现一个Interface。
2. InvocationHandler：InvocationHandler是一个接口，可以通过实现这个接口，定义横切逻辑，再通过反射机制（invoke）调用目标类的代码，在此过程，可能包装逻辑，对目标方法进行前置后置处理。
3. Proxy：Proxy利用InvocationHandler动态创建一个符合目标类实现的接口的实例，生成目标类的代理对象。

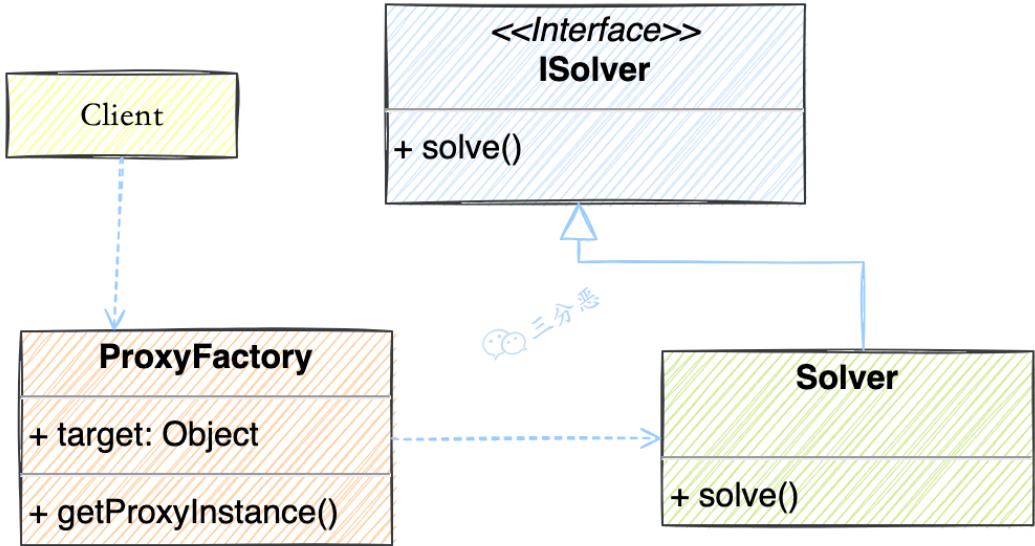
CgLib 动态代理

1. 使用JDK创建代理有一大限制，它只能为接口创建代理实例，而CgLib 动态代理就没有这个限制。
2. CgLib 动态代理是使用字节码处理框架 ASM，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。
3. CgLib 创建的动态代理对象性能比 JDK 创建的动态代理对象的性能高不少，但是 CGLib 在创建代理对象时所花费的时间却比 JDK 多得多，所以对于单例的对象，因为无需频繁创建对象，用 CGLib 合适，反之，使用 JDK 方式要更为合适一些。同时，由于 CGLib 由于是采用动态创建子类的方法，对于 final 方法，无法进行代理。

我们来看一个常见的小场景，客服中转，解决用户问题：



JDK动态代理实现：



- 接口

```

1 | public interface ISolver {
2 |     void solve();
3 |

```

- 目标类:需要实现对应接口

```

1 | public class Solver implements ISolver {
2 |     @Override
3 |     public void solve() {
4 |         System.out.println("疯狂掉头发解决问题.....");
5 |     }
6 |

```

- 态代理工厂:ProxyFactory, 直接用反射方式生成一个目标对象的代理对象, 这里用了一个匿名内部类方式重写InvocationHandler方法, 实现接口重写也差不多

```

1 | public class ProxyFactory {
2 |
3 |     // 维护一个目标对象
4 |     private Object target;
5 |
6 |     public ProxyFactory(Object target) {
7 |         this.target = target;
8 |     }

```

```

9
10    // 为目标对象生成代理对象
11    public Object getProxyInstance() {
12        return
13            Proxy.newProxyInstance(target.getClass().getClassLoader(),
14                target.getClass().getInterfaces(),
15                    new InvocationHandler() {
16                        @Override
17                        public Object invoke(Object proxy,
18                            Method method, Object[] args) throws Throwable {
19                                System.out.println("请问有什么可以帮助您? ");
20
21                                // 调用目标对象方法
22                                Object returnValue =
23                                    method.invoke(target, args);
24
25                                System.out.println("问题已经解决
26                                啦! ");
27                                return null;
28                            }
29                        });
30        }
31    }

```

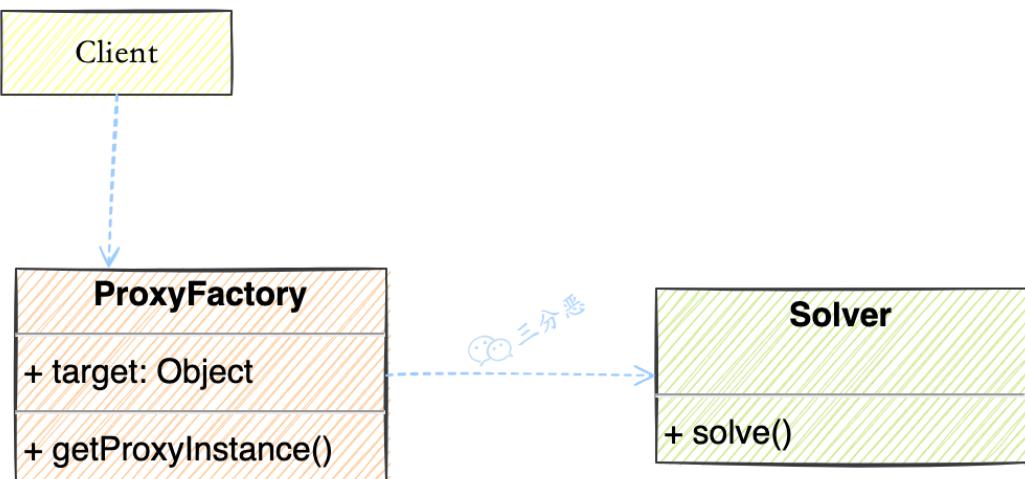
- 客户端: Client, 生成一个代理对象实例, 通过代理对象调用目标对象方法

```

1  public class Client {
2      public static void main(String[] args) {
3          // 目标对象: 程序员
4          ISolver developer = new Solver();
5          // 代理: 客服小姐姐
6          ISolver csProxy = (ISolver) new
7              ProxyFactory(developer).getProxyInstance();
8          // 目标方法: 解决问题
9          csProxy.solve();
10     }

```

Cglib动态代理实现:



- 目标类：Solver，这里目标类不用再实现接口。

```

1 public class Solver {
2
3     public void solve() {
4         System.out.println("疯狂掉头发解决问题.....");
5     }
6 }

```

- 动态代理工厂：

```

1 public class ProxyFactory implements MethodInterceptor {
2
3     //维护一个目标对象
4     private Object target;
5
6     public ProxyFactory(Object target) {
7         this.target = target;
8     }
9
10    //为目标对象生成代理对象
11    public Object getProxyInstance() {
12        //工具类
13        Enhancer en = new Enhancer();
14        //设置父类
15        en.setSuperclass(target.getClass());
16        //设置回调函数
17        en.setCallback(this);
18        //创建子类对象代理

```

```

19     return en.create();
20 }
21
22     @Override
23     public Object intercept(Object obj, Method method,
24                             Object[] args, MethodProxy proxy) throws Throwable {
25         System.out.println("请问有什么可以帮到您? ");
26         // 执行目标对象的方法
27         Object returnValue = method.invoke(target, args);
28         System.out.println("问题已经解决啦! ");
29         return null;
30     }
31 }
```

- 客户端: Client

```

1 public class Client {
2     public static void main(String[] args) {
3         // 目标对象: 程序员
4         Solver developer = new Solver();
5         // 代理: 客服小姐姐
6         Solver csProxy = (Solver) new
7             ProxyFactory(developer).getProxyInstance();
8         // 目标方法: 解决问题
9         csProxy.solve();
10    }
```

22. 说说Spring AOP 和 AspectJ AOP 区别?

Spring AOP

Spring AOP 属于 **运行时增强**，主要具有如下特点：

1. 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
2. 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

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：

- 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。
- 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。
- 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

整体对比如下：

Spring AOP	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
易学习使用	比AOP更复杂
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

事务

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。Spring 只提供统一事务管理接口，具体实现都是由各数据库自己实现，数据库事务的提交和回滚是通过数据库自己的事务机制实现。

23.Spring 事务的种类？

Spring 支持 **编程式事务** 管理和 **声明式** 事务管理两种方式：



1. 编程式事务

编程式事务管理使用 `TransactionTemplate`，需要显式执行事务。

2. 声明式事务

3. 声明式事务管理建立在 AOP 之上的。其本质是通过 AOP 功能，对方法前后进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前启动一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务
4. 优点是不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明或通过 `@Transactional` 注解的方式，便可以将事务规则应用到业务逻辑中，减少业务代码的污染。唯一不足地方是，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

24.Spring 的事务隔离级别？

Spring 的接口 `TransactionDefinition` 中定义了表示隔离级别的常量，当然其实主要还是对应数据库的事务隔离级别：

1. `ISOLATION_DEFAULT`: 使用后端数据库默认的隔离级别，MySQL 默认可重复读，Oracle 默认读已提交。
2. `ISOLATION_READ_UNCOMMITTED`: 读未提交
3. `ISOLATION_READ_COMMITTED`: 读已提交

4. ISOLATION_REPEATABLE_READ: 可重复读

5. ISOLATION_SERIALIZABLE: 串行化

25. Spring 的事务传播机制？

Spring 事务的传播机制说的是，当多个事务同时存在的时候——一般指的是多个事务方法相互调用时，Spring 如何处理这些事务的行为。

事务传播机制是使用简单的 ThreadLocal 实现的，所以，如果调用的方法是在新线程调用的，事务传播实际上会失效的。

7种事务传播机制

事务传播机制	描述
REQUIRED	默认，如果没有当前事务，就新建一个事务，如果存在一个事务，就加入到这个事务中
SUPPORTS	支持当前事务，如果没有当前事务，就以非事务方法执行
MANDATORY	使用当前事务，如果没有当前事务，就抛出异常
REQUIRED_NEW	新建事务，如果存在当前事务，就把当前事务挂起
NOT_SUPPORTED	以非事务方式执行操作，如果当前事务存在则抛出异常
NESTED	如果当前存在事务，则在事务内执行，如果当前没有事务，则执行与REQUIRED类似操作

Spring 默认的事务传播行为是PROPAGATION_REQUIRED，它适合绝大多数情况，如果多个ServiceX#methodX()都工作在事务环境下（均被Spring事务增强），且程序中存在调用链Service1#method1()->Service2#method2()->Service3#method3()，那么这3个服务类的三个方法通过Spring的事务传播机制都工作在同一个事务中。

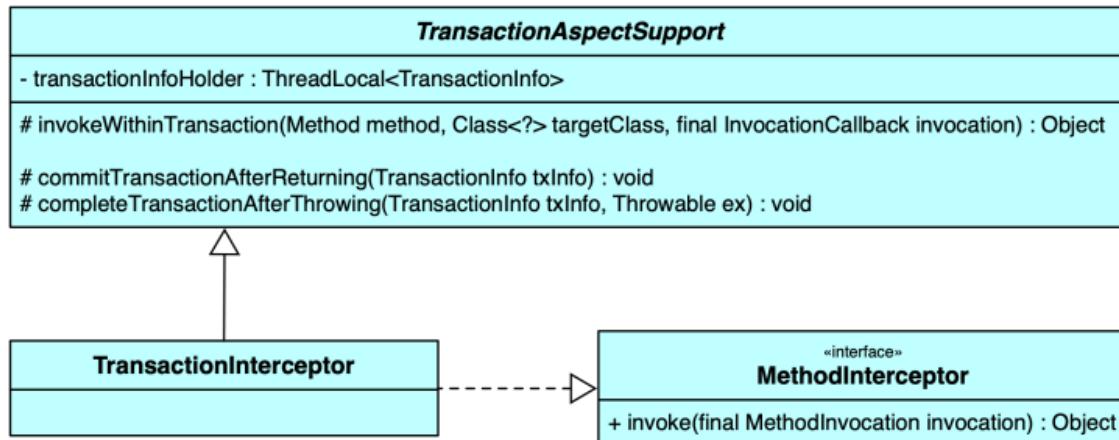
26. 声明式事务实现原理了解吗？

就是通过AOP/动态代理。

- 在Bean初始化阶段创建代理对象：Spring容器在初始化每个单例bean的时候，会遍历容器中的所有BeanPostProcessor实现类，并执行其postProcessAfterInitialization方法，在执行AbstractAutoProxyCreator类的postProcessAfterInitialization方法时会遍历容器中所有的切面，查找与当前实例化bean匹配的切面，这里会获取事务属性切面，查找@Transactional注解及其属性

值，然后根据得到的切面创建一个代理对象，默认是使用JDK动态代理创建代理，如果目标类是接口，则使用JDK动态代理，否则使用Cglib。

- 在执行目标方法时进行事务增强操作：当通过代理对象调用Bean方法的时候，会触发对应的AOP增强拦截器，声明式事务是一种环绕增强，对应接口为 `MethodInterceptor`，事务增强对该接口的实现为 `TransactionInterceptor`，类图如下：



事务拦截器 `TransactionInterceptor` 在 `invoke` 方法中，通过调用父类 `TransactionAspectSupport` 的 `invokeWithinTransaction` 方法进行事务处理，包括开启事务、事务提交、异常回滚。

27. 声明式事务在哪些情况下会失效？



1、`@Transactional` 应用在非 `public` 修饰的方法上

如果 `Transactional` 注解应用在非 `public` 修饰的方法上，`Transactional` 将会失效。

是因为在Spring AOP 代理时， TransactionInterceptor（事务拦截器）在目标方法执行前后进行拦截， DynamicAdvisedInterceptor（CglibAopProxy 的内部类）的 intercept方法 或 JdkDynamicAopProxy的invoke方法会间接调用 AbstractFallbackTransactionAttributeSource的 **computeTransactionAttribute**方法， 获取Transactional 注解的事务配置信息。

```
1 protected TransactionAttribute  
2     computeTransactionAttribute(Method method,  
3         Class<?> targetClass) {  
4             // Don't allow no-public methods as required.  
5             if (allowPublicMethodsOnly() &&  
6                 !Modifier.isPublic(method.getModifiers())) {  
7                 return null;  
8             }  
9         }
```

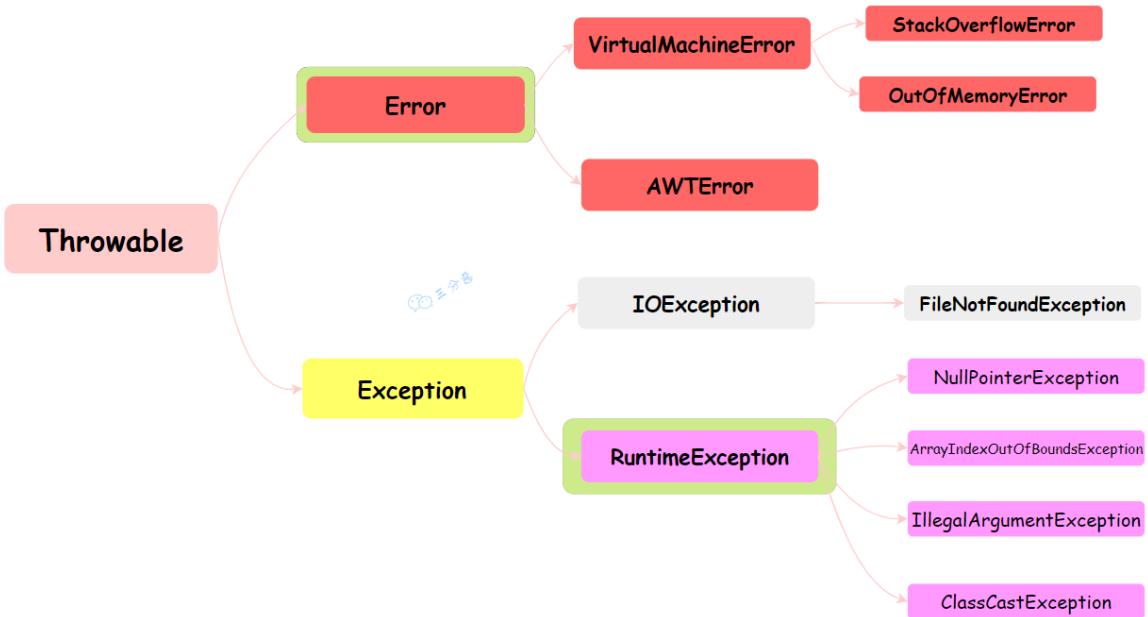
此方法会检查目标方法的修饰符是否为 public，不是 public则不会获取 @Transactional 的属性配置信息。

2、@Transactional注解属性 propagation 设置错误

- TransactionDefinition.PROPAGATION_SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
- TransactionDefinition.PROPAGATION_NOT_SUPPORTED：以非事务方式运行，如果当前存在事务，则把当前事务挂起。
- TransactionDefinition.PROPAGATION_NEVER：以非事务方式运行，如果当前存在事务，则抛出异常。

3、@Transactional注解属性 rollbackFor 设置错误

rollbackFor 可以指定能够触发事务回滚的异常类型。Spring默认抛出了未检查 unchecked异常（继承自 RuntimeException的异常）或者 Error才回滚事务，其他异常不会触发回滚事务。



```

1 // 希望自定义的异常可以进行回滚
2 @Transactional(propagation= Propagation.REQUIRED, rollbackFor=
MyException.class)

```

若在目标方法中抛出的异常是 `rollbackFor` 指定的异常的子类，事务同样会回滚。

4、同一个类中方法调用，导致`@Transactional`失效

开发中避免不了会对同一个类里面的方法调用，比如有一个类Test，它的一个方法A，A再调用本类的方法B（不论方法B是用public还是private修饰），但方法A没有声明注解事务，而B方法有。则外部调用方法A之后，方法B的事务是不会起作用的。这也是经常犯错误的一个地方。

那为啥会出现这种情况？其实这还是由于使用Spring AOP代理造成的，因为只有当事务方法被当前类以外的代码调用时，才会由Spring生成的代理对象来管理。

```

1 // @Transactional
2 @GetMapping("/test")
3 private Integer A() throws Exception {
4     CityInfoDict cityInfoDict = new CityInfoDict();
5     cityInfoDict.setCityName("2");
6     /**
7      * B 插入字段为 3的数据
8      */
9     this.insertB();
10    /**
11     * A 插入字段为 2的数据

```

```

12     */
13     int insert = cityInfoDictMapper.insert(cityInfoDict);
14     return insert;
15 }
16
17 @Transactional()
18 public Integer insertB() throws Exception {
19     CityInfoDict cityInfoDict = new CityInfoDict();
20     cityInfoDict.setCityName("3");
21     cityInfoDict.setParentCityId(3);
22
23     return cityInfoDictMapper.insert(cityInfoDict);
24 }
25

```

这种情况是最常见的一种@Transactional注解失效场景

```

1 @Transactional
2 private Integer A() throws Exception {
3     int insert = 0;
4     try {
5         CityInfoDict cityInfoDict = new CityInfoDict();
6         cityInfoDict.setCityName("2");
7         cityInfoDict.setParentCityId(2);
8         /**
9          * A 插入字段为 2的数据
10         */
11         insert = cityInfoDictMapper.insert(cityInfoDict);
12         /**
13          * B 插入字段为 3的数据
14         */
15         b.insertB();
16     } catch (Exception e) {
17         e.printStackTrace();
18     }
19 }
20

```

如果B方法内部抛了异常，而A方法此时try catch了B方法的异常，那这个事务就不能正常回滚了，会抛出异常：

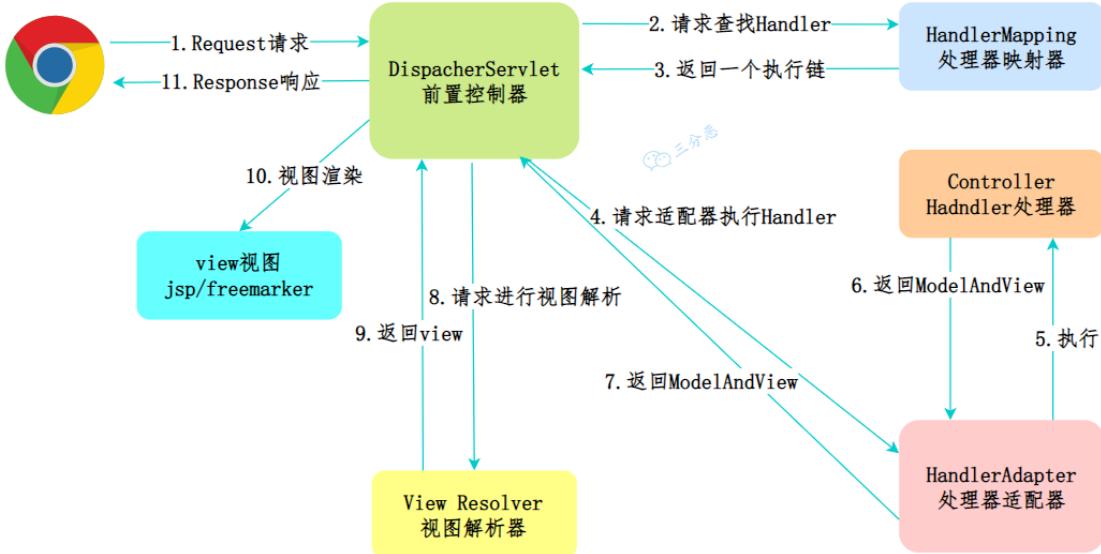
```
1 | org.springframework.transaction.UnexpectedRollbackException:  
| Transaction rolled back because it has been marked as  
| rollback-only
```

MVC

28.Spring MVC 的核心组件？

1. DispatcherServlet：前置控制器，是整个流程控制的核心，控制其他组件的执行，进行统一调度，降低组件之间的耦合性，相当于总指挥。
2. Handler：处理器，完成具体的业务逻辑，相当于 Servlet 或 Action。
3. HandlerMapping：DispatcherServlet 接收到请求之后，通过 HandlerMapping 将不同的请求映射到不同的 Handler。
4. HandlerInterceptor：处理器拦截器，是一个接口，如果需要完成一些拦截处理，可以实现该接口。
5. HandlerExecutionChain：处理器执行链，包括两部分内容：Handler 和 HandlerInterceptor（系统会有一个默认的 HandlerInterceptor，如果需要额外设置拦截，可以添加拦截器）。
6. HandlerAdapter：处理器适配器，Handler 执行业务方法之前，需要进行一系列的操作，包括表单数据的验证、数据类型的转换、将表单数据封装到 JavaBean 等，这些操作都是由 HandlerAdapter 来完成，开发者只需将注意力集中业务逻辑的处理上，DispatcherServlet 通过 HandlerAdapter 执行不同的 Handler。
7. ModelAndView：装载了模型数据和视图信息，作为 Handler 的处理结果，返回给 DispatcherServlet。
8. ViewResolver：视图解析器，DispatcherServlet 通过它将逻辑视图解析为物理视图，最终将渲染结果响应给客户端。

29.Spring MVC 的工作流程？



1. 客户端向服务端发送一次请求，这个请求会先到前端控制器DispatcherServlet(也叫中央控制器)。
2. DispatcherServlet接收到请求后会调用HandlerMapping处理器映射器。由此得知，该请求该由哪个Controller来处理（并未调用Controller，只是得知）
3. DispatcherServlet调用HandlerAdapter处理器适配器，告诉处理器适配器应该要去执行哪个Controller
4. HandlerAdapter处理器适配器去执行Controller并得到 ModelAndView(数据和视图)，并层层返回给DispatcherServlet
5. DispatcherServlet将 ModelAndView交给 ViewReslover 视图解析器解析，然后返回真正的视图。
6. DispatcherServlet将模型数据填充到视图中
7. DispatcherServlet将结果响应给客户端

Spring MVC 虽然整体流程复杂，但是实际开发中很简单，大部分的组件不需要开发人员创建和管理，只需要通过配置文件的方式完成配置即可，真正需要开发人员进行处理的只有 **Handler (Controller)** 、 **View** 、 **Model**。

当然我们现在大部分的开发都是前后端分离，Restful风格接口，后端只需要返回Json数据就行了。

30.SpringMVC Restful风格的接口的流程是什么样的呢？

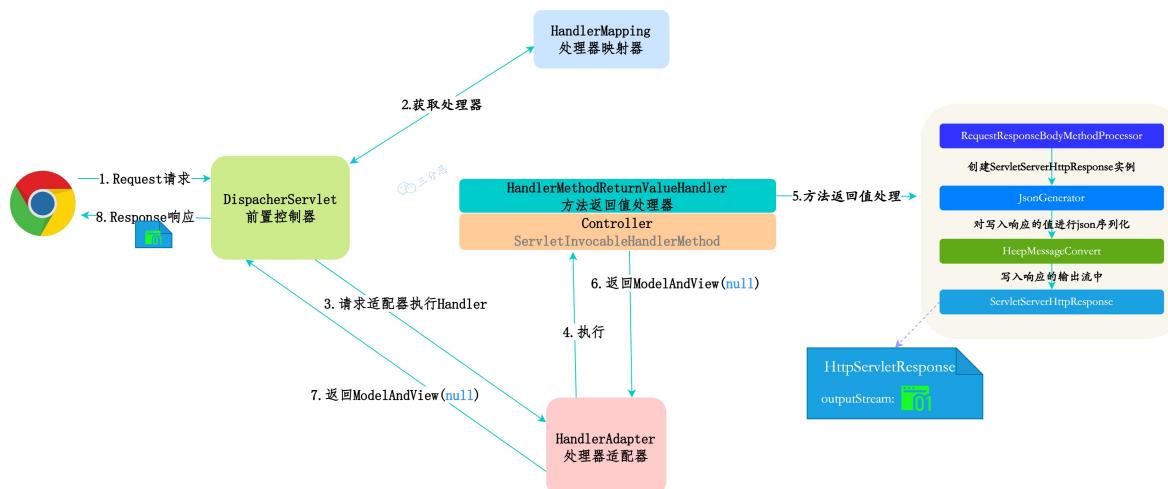
PS:这是一道全新的八股，毕竟 ModelAndView 这种方式应该没人用了吧？现在都是前后端分离接口，八股也该更新换代了。

我们都知道Restful接口，响应格式是json，这就用到了一个常用注解：

@ResponseBody

```
1  @GetMapping("/user")
2  @ResponseBody
3  public User user(){
4      return new User(1, "张三");
5  }
```

加入了这个注解后，整体的流程上和使用 ModelAndView 大体上相同，但是细节上有一些不同：



1. 客户端向服务端发送一次请求，这个请求会先到前端控制器DispatcherServlet
2. DispatcherServlet接收到请求后会调用HandlerMapping处理器映射器。由此得知，该请求该由哪个Controller来处理
3. DispatcherServlet调用HandlerAdapter处理器适配器，告诉处理器适配器应该要去执行哪个Controller
4. Controller被封装成了ServletInvocableHandlerMethod，HandlerAdapter处理器适配器去执行invokeAndHandle方法，完成对Controller的请求处理
5. HandlerAdapter执行完对Controller的请求，会调用HandlerMethodReturnValueHandler去处理返回值，主要的过程：
 - 5.1. 调用RequestResponseBodyMethodProcessor，创建ServletServerHttpResponse（Spring对原生ServerHttpResponse的封装）实例
 - 5.2. 使用HttpMessageConverter的write方法，将返回值写入ServletServerHttpResponse的OutputStream输出流中

5.3. 在写入的过程中，会使用JsonGenerator（默认使用Jackson框架）对返回值进行Json序列化

6. 执行完请求后，返回的ModealAndView为null，ServletServerHttpResponse里也已经写入了响应，所以不用关心View的处理

Spring Boot

31. 介绍一下SpringBoot，有哪些优点？

Spring Boot 基于 Spring 开发，Spring Boot 本身并不提供 Spring 框架的核心特性以及扩展功能，只是用于快速、敏捷地开发新一代基于 Spring 框架的应用程序。它并不是用来替代 Spring 的解决方案，而是和 Spring 框架紧密结合用于提升 Spring 开发者体验的工具。



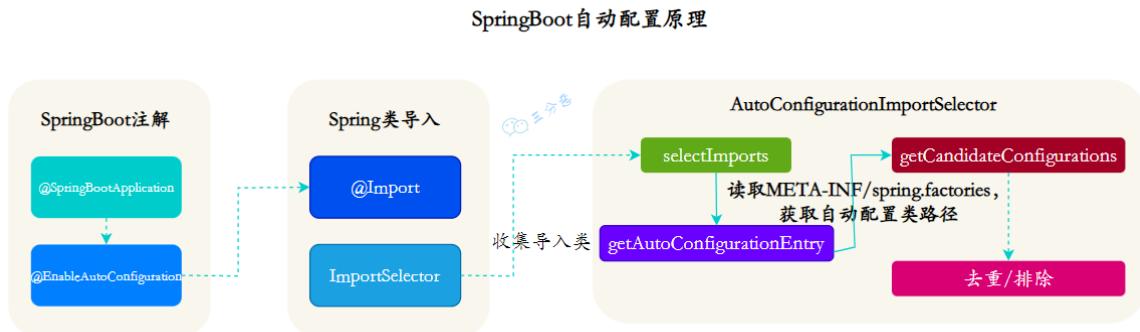
Spring Boot 以 **约定大于配置** 核心思想开展工作，相比Spring具有如下优势：

1. Spring Boot 可以快速创建独立的Spring应用程序。
2. Spring Boot 内嵌了如Tomcat, Jetty和Undertow这样的容器，也就是说可以直接跑起来，用不着再做部署工作了。
3. Spring Boot 无需再像Spring一样使用一堆繁琐的xml文件配置。
4. Spring Boot 可以自动配置(核心)Spring。SpringBoot将原有的XML配置改为Java配置，将bean注入改为使用注解注入的方式(@Autowire)，并将多个xml、properties配置浓缩在一个application.yml配置文件中。
5. Spring Boot 提供了一些现有的功能，如量度工具，表单数据验证以及一些外部配置这样的一些第三方功能。
6. Spring Boot 可以快速整合常用依赖（开发库，例如spring-webmvc、jackson-json、validation-api和tomcat等），提供的POM可以简化Maven的配置。当我们引

入核心依赖时，SpringBoot会自引入其他依赖。

32.SpringBoot自动配置原理了解吗？

SpringBoot开启自动配置的注解是 `@EnableAutoConfiguration`，启动类上的注解 `@SpringBootApplication` 是一个复合注解，包含了 `@EnableAutoConfiguration`：



- `EnableAutoConfiguration` 只是一个简单的注解，自动装配核心功能的实现实际上是通过 `AutoConfigurationImportSelector` 类

```
1  @AutoConfigurationPackage //将main同级的包下的所有组件注册到容器中
2  @Import({AutoConfigurationImportSelector.class}) //加载自动装配类 xxxAutoconfiguration
3  public @interface EnableAutoConfiguration {
4      String ENABLED_OVERRIDE_PROPERTY =
5          "spring.boot.enableautoconfiguration";
6
7      Class<?>[] exclude() default {};
8
9      String[] excludeName() default {};
}
```

- `AutoConfigurationImportSelector` 实现了 `ImportSelector` 接口，这个接口的作用就是收集需要导入的配置类，配合 `@Import()` 就可以将相应的类导入到Spring容器中
- 获取注入类的方法是 `selectImports()`，它实际调用的是 `getAutoConfigurationEntry`，这个方法是获取自动装配类的关键，主要流程可以分为这么几步：

1. 获取注解的属性，用于后面的排除
2. 获取所有需要自动装配的配置类的路径：这一步是最关键的，从META-INF/spring.factories获取自动配置类的路径
3. 去掉重复的配置类和需要排除的重复类，把需要自动加载的配置类的路径存储起来

```
1  protected
2      AutoConfigurationImportSelector.AutoConfigurationEntry
3      getAutoConfigurationEntry(AnnotationMetadata
4          annotationMetadata) {
5          if (!this.isEnabled(annotationMetadata)) {
6              return EMPTY_ENTRY;
7          } else {
8              //1. 获取到注解的属性
9              AnnotationAttributes attributes =
10             this.getAttributes(annotationMetadata);
11             //2. 获取需要自动装配的所有配置类，读取META-
12             INF/spring.factories，获取自动配置类路径
13             List<String> configurations =
14             this.getCandidateConfigurations(annotationMetadata,
15                 attributes);
16             //3.1. 移除重复的配置
17             configurations =
18             this.removeDuplicates(configurations);
19             //3.2. 处理需要排除的配置
20             Set<String> exclusions =
21             this.getExclusions(annotationMetadata, attributes);
22             this.checkExcludedClasses(configurations,
23                 exclusions);
24             configurations.removeAll(exclusions);
25             configurations =
26             this.getConfigurationClassFilter().filter(configurations);
27
28             this.fireAutoConfigurationImportEvents(configurations,
29                 exclusions);
30             return new
31                 AutoConfigurationImportSelector.AutoConfigurationEntry(config
32                     urations, exclusions);
33         }
34     }
```

33.如何自定义一个SpringBoot Starter?

知道了自动配置原理，创建一个自定义SpringBoot Starter也很简单。

1. 创建一个项目，命名为demo-spring-boot-starter，引入SpringBoot相关依赖

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter</artifactId>
4 </dependency>
5 <dependency>
6   <groupId>org.springframework.boot</groupId>
7   <artifactId>spring-boot-configuration-
processor</artifactId>
8   <optional>true</optional>
9 </dependency>
```

2. 编写配置文件

这里定义了属性配置的前缀

```
1 @ConfigurationProperties(prefix = "hello")
2 public class HelloProperties {
3
4     private String name;
5
6     //省略getter、setter
7 }
```

3. 自动装配

创建自动配置类HelloPropertiesConfigure

```
1 @Configuration
2 @EnableConfigurationProperties(HelloProperties.class)
3 public class HelloPropertiesConfigure {
4 }
```

4. 配置自动类

在 `/resources/META-INF/spring.factories` 文件中添加自动配置类路径

```
1 | org.springframework.boot.autoconfigure.EnableAutoConfigura
  | tion=\n
2 | cn.fighter3.demo.starter.configure.HelloPropertiesConfigur
  | e
```

5. 测试

- 创建一个工程，引入自定义starter依赖

```
1 | <dependency>
2 |   <groupId>cn.fighter3</groupId>
3 |   <artifactId>demo-spring-boot-
  | starter</artifactId>
4 |   <version>0.0.1-SNAPSHOT</version>
5 | </dependency>
```

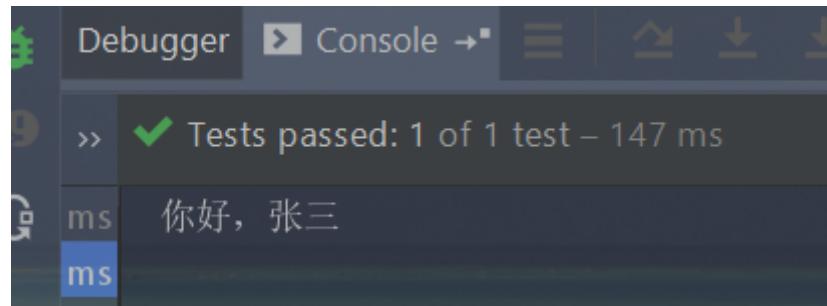
- 在配置文件里添加配置

```
1 | hello.name=张三
```

- 测试类

```
1 | @RunWith(SpringRunner.class)
2 | @SpringBootTest
3 | public class HelloTest {
4 |     @Autowired
5 |     HelloProperties helloProperties;
6 |
7 |     @Test
8 |     public void hello(){
9 |         System.out.println("你
  | 好, "+helloProperties.getName());
10 |     }
11 | }
```

- 运行结果



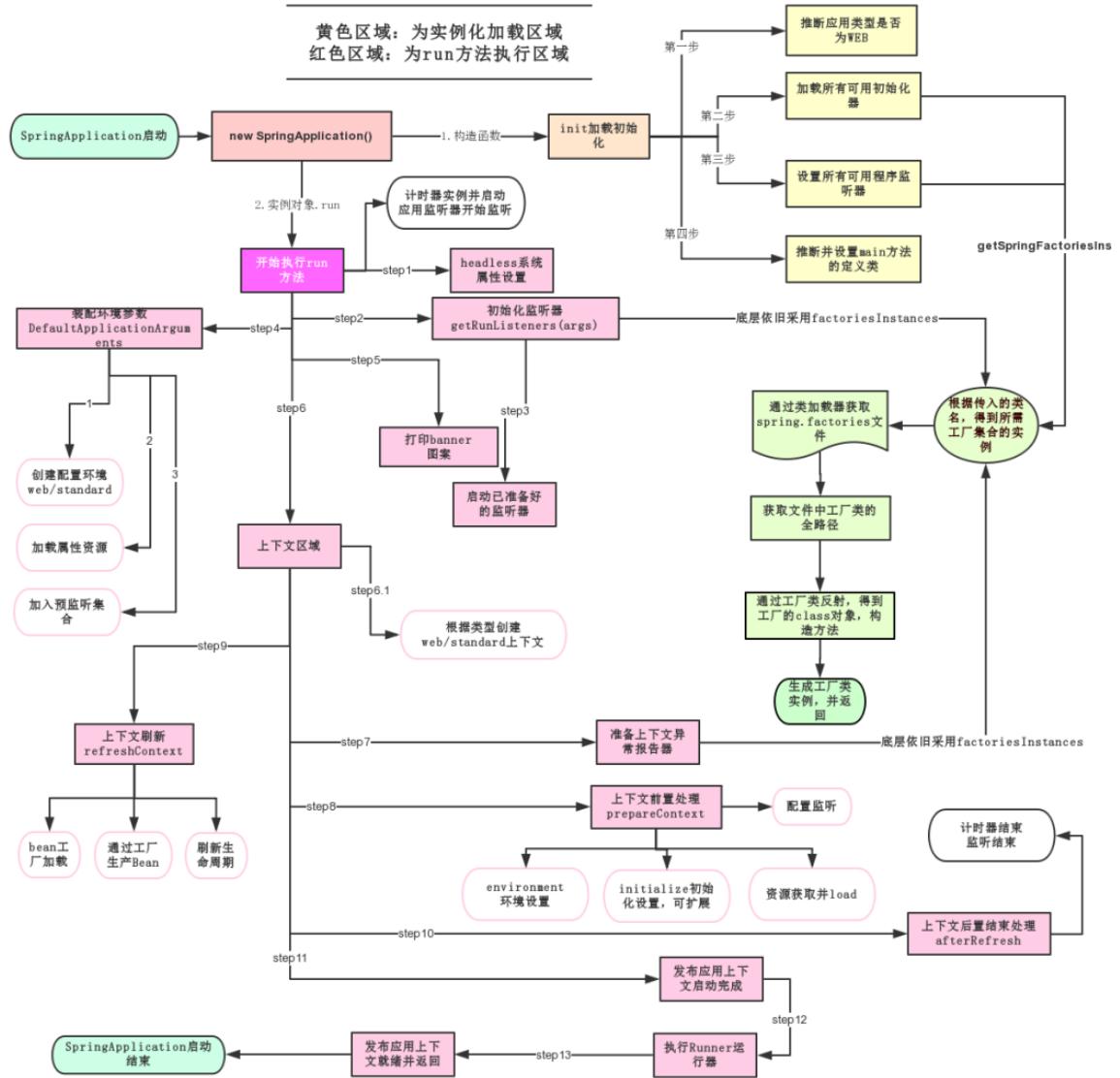
至此，随手写的一个自定义SpringBoot-Starter就完成了，虽然比较简单，但是完成了主要的自动装配的能力。

34.Springboot 启动原理？

SpringApplication 这个类主要做了以下四件事情：

1. 推断应用的类型是普通的项目还是 Web 项目
2. 查找并加载所有可用初始化器，设置到 initializers 属性中
3. 找出所有的应用程序监听器，设置到 listeners 属性中
4. 推断并设置 main 方法的定义类，找到运行的主类

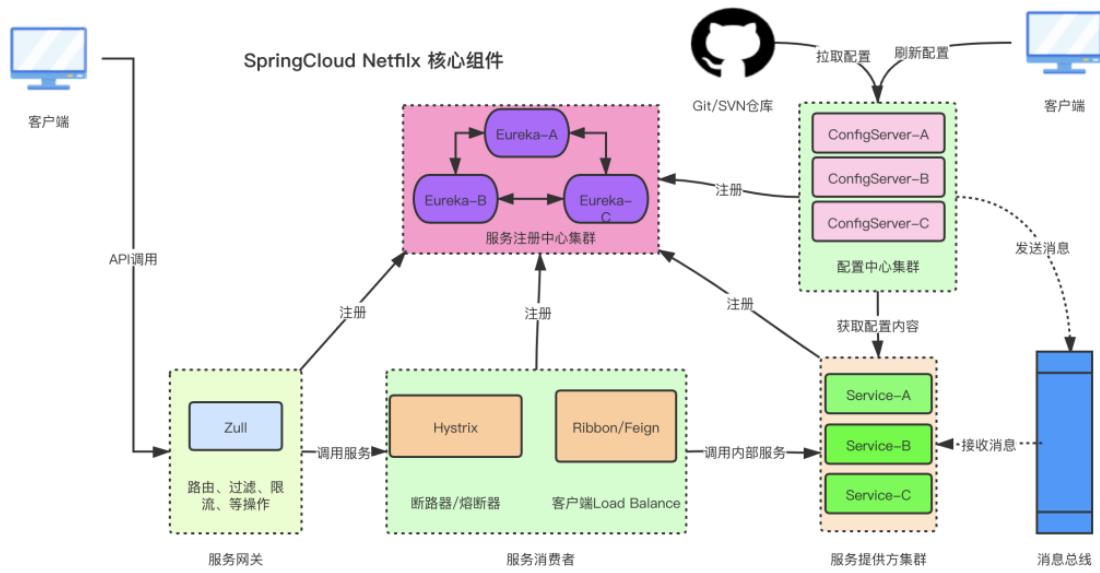
SpringBoot 启动大致流程如下：



Spring Cloud

35. 对 SpringCloud 了解多少？

SpringCloud是Spring官方推出的微服务治理框架。



什么是微服务？

1. 2014 年 Martin Fowler 提出的一种新的架构形式。微服务架构是一种 架构模式，提倡将单一应用程序划分成一组小的服务，服务之间相互协调，互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务与服务之间采用轻量级的通信机制(如HTTP或Dubbo)互相协作，每个服务都围绕着具体的业务进行构建，并且能够被独立的部署到生产环境中，另外，应尽量避免统一的，集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言、工具(如Maven)对其进行构建。
2. 微服务化的核心就是将传统的一站式应用，根据业务拆分成一个一个的服务，彻底地去耦合，每一个微服务提供单个业务功能的服务，一个服务做一件事情，从技术角度看就是一种小而独立的处理过程，类似进程的概念，能够自行单独启动或销毁，拥有自己独立的数据库。

微服务架构主要要解决哪些问题？

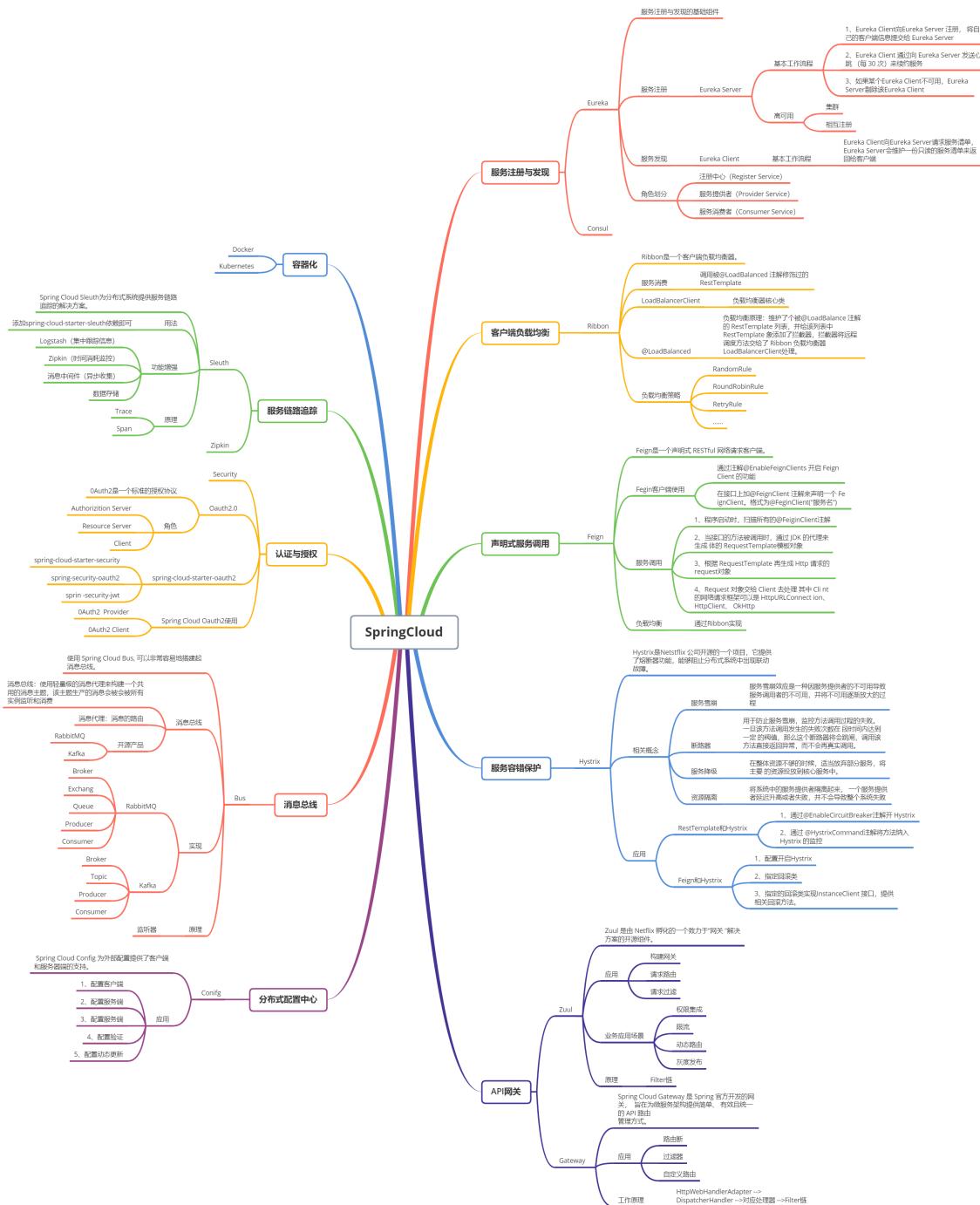
1. 服务很多，客户端怎么访问，如何提供对外网关？
2. 这么多服务，服务之间如何通信? HTTP还是RPC?
3. 这么多服务，如何治理? 服务的注册和发现。
4. 服务挂了怎么办？熔断机制。

有哪些主流微服务框架？

1. Spring Cloud Netflix
2. Spring Cloud Alibaba

3. SpringBoot + Dubbo + ZooKeeper

SpringCloud有哪些核心组件？



PS: 微服务后面有机会再扩展，其实面试一般都是结合项目去问。

参考：

[1]. 《Spring揭秘》

[2]. [面试官：关于Spring就问这13个](#)

- [3]. [15个经典的Spring面试常见问题](#)
- [4]. [面试还不知道BeanFactory和ApplicationContext的区别？](#)
- [5]. [Java面试中常问的Spring方面问题（涵盖七大方向共55道题，含答案）](#)
- [6] . [Spring Bean 生命周期（实例结合源码彻底讲透）](#)
- [7]. [@Autowired注解的实现原理](#)
- [8]. [万字长文，带你从源码认识Spring事务原理，让Spring事务不再是面试噩梦](#)
- [9]. [【技术干货】Spring事务原理一探](#)
- [10]. [Spring的声明式事务@Transactional注解的6种失效场景](#)
- [11]. Spring官网
- [12]. [Spring使用了哪些设计模式？](#)
- [13]. 《精通Spring4.X企业应用开发实战》
- [14]. [Spring 中的bean 是线程安全的吗？](#)
- [15]. 小傅哥 《手撸Spring》
- [16]. [手撸架构，Spring 面试63问](#)
- [17]. [@Autowired注解的实现原理](#)
- [18]. [如何优雅地在 Spring Boot 中使用自定义注解](#)
- [19]. [Spring MVC源码\(三\) ----- @RequestBody和@ResponseBody原理解析](#)

关注公众号：三分恶

手册更新动态
即刻送达



添加个人微信：ThirdFighter

技术交流
加大佬云集微信群



二、MyBatis

基础

1. 说说什么是MyBatis?



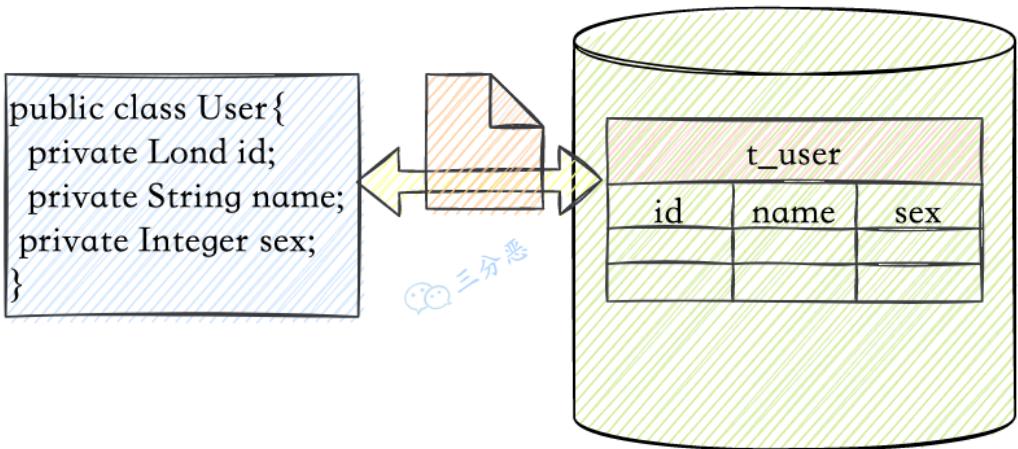
先吹一下：

- Mybatis 是一个半 ORM（对象关系映射）框架，它内部封装了 JDBC，开发时只需要关注 SQL 语句本身，不需要花费精力去处理加载驱动、创建连接、创建 statement 等繁杂的过程。程序员直接编写原生态 sql，可以严格控制 sql 执行性能，灵活度高。
- MyBatis 可以使用 XML 或注解来配置和映射原生信息，将 POJO 映射成数据库中的记录，避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。

再说一下缺点

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

ORM是什么？



- ORM (Object Relational Mapping)，对象关系映射，是一种为了解决关系型数据库数据与简单Java对象（POJO）的映射关系的技术。简单来说，ORM是通过使用描述对象和数据库之间映射的元数据，将程序中的对象自动持久化到关系型数据库中。

为什么说Mybatis是半自动ORM映射工具？它与全自动的区别在哪里？

- Hibernate属于全自动ORM映射工具，使用Hibernate查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。
- 而Mybatis在查询关联对象或关联集合对象时，需要手动编写SQL来完成，所以，被称之为半自动ORM映射工具。

JDBC编程有哪些不足之处，MyBatis是如何解决的？

```

String URL = "jdbc:mysql://localhost:3306/demo";
String USER = "root";
String PASSWORD = "123456";
//加载驱动程序
Class.forName("com.mysql.jdbc.Driver");
//获得数据库连接
Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
//定义sql
String sql = "SELECT name, age FROM t_user where name=?";
//操作数据库，实现增删改查
PreparedStatement pstmt = conn.prepareStatement(sql);
String queryName = "fighter3";
pstmt.setString(1, queryName);
ResultSet rs = pstmt.executeQuery();
//如果有数据，rs.next()返回true
while (rs.next()) {
    System.out.println(rs.getString("name") + " 性别: " + rs.getInt("sex"));
}

```

数据库连接频繁创建，消耗资源

sql语句写在代码里不好维护

三分熟

sql语句传参麻烦

结果集解析麻烦

- 1、数据连接创建、释放频繁造成系统资源浪费从而影响系统性能

- 解决：在mybatis-config.xml中配置数据链接池，使用连接池统一管理数据库连接。
- 2、sql语句写在代码中造成代码不易维护
 - 解决：将sql语句配置在XXXXmapper.xml文件中与java代码分离。
- 3、向sql语句传参数麻烦，因为sql语句的where条件不一定，可能多也可能少，占位符需要和参数一一对应。
 - 解决：Mybatis自动将java对象映射至sql语句。
- 4、对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成pojo对象解析比较方便。
 - 解决：Mybatis自动将sql执行结果映射至java对象。

2.Hibernate 和 MyBatis 有什么区别？

PS:直接用Hibernate的应该不多了吧，毕竟大家都是“敏捷开发”，但架不住面试爱问。

相同点

- 都是对jdbc的封装，都是应用于持久层的框架。



这还用说？

不同点

- 映射关系
 - MyBatis 是一个半自动映射的框架，配置Java对象与sql语句执行结果的对应关系，多表关联关系配置简单

- Hibernate 是一个全表映射的框架，配置Java对象与数据库表的对应关系，多表关联关系配置复杂
- **SQL优化和移植性**
 - Hibernate 对SQL语句封装，提供了日志、缓存、级联（级联比 MyBatis 强大）等特性，此外还提供 HQL（Hibernate Query Language）操作数据库，数据库无关性支持好，但会多消耗性能。如果项目需要支持多种数据库，代码开发量少，但SQL语句优化困难。
 - MyBatis 需要手动编写 SQL，支持动态 SQL、处理列表、动态生成表名、支持存储过程。开发工作量相对大些。直接使用SQL语句操作数据库，不支持数据库无关性，但sql语句优化容易。

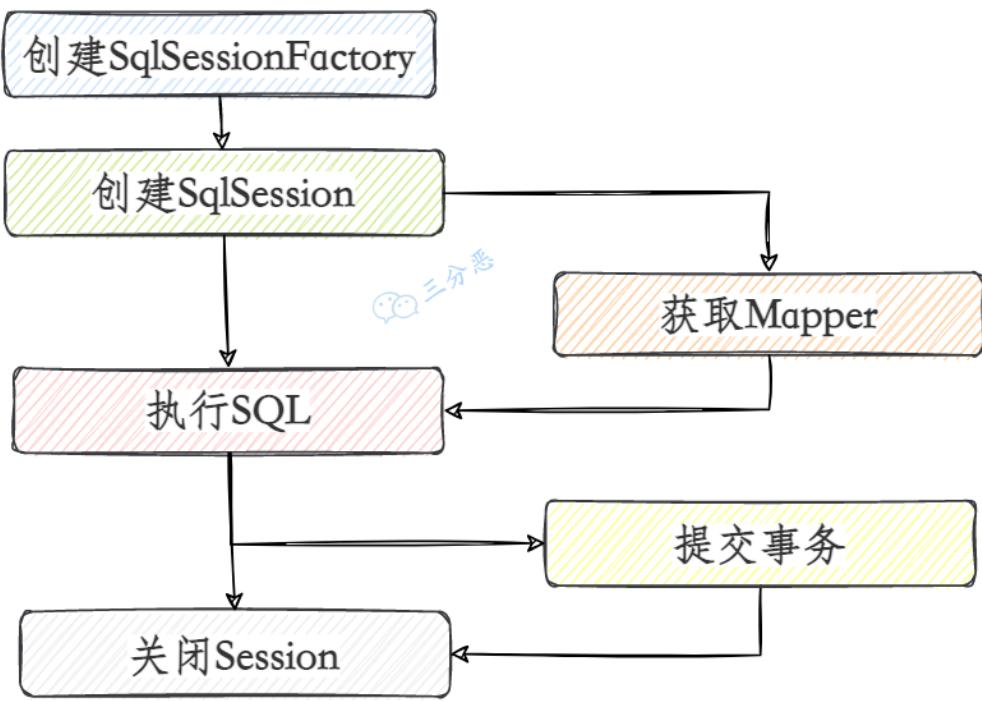
MyBatis和Hibernate的适用场景？



- Hibernate 是标准的ORM框架，SQL编写量较少，但不够灵活，适合于需求相对稳定，中小型的软件项目，比如：办公自动化系统
- MyBatis 是半ORM框架，需要编写较多SQL，但是比较灵活，适合于需求变化频繁，快速迭代的项目，比如：电商网站

3.MyBatis使用过程？生命周期？

MyBatis基本使用的过程大概可以分为这么几步：



- 1、创建SqlSessionFactory

可以从配置或者直接编码来创建SqlSessionFactory

```

1 | String resource = "org/mybatis/example/mybatis-config.xml";
2 | InputStream inputStream =
3 | Resources.getResourceAsStream(resource);
4 | SqlSessionFactory sqlSessionFactory = new
5 | SqlSessionFactoryBuilder().build(inputStream);

```

- 2、通过SqlSessionFactory创建SqlSession

SqlSession（会话）可以理解为程序和数据库之间的桥梁

```

1 | SqlSession session = sqlSessionFactory.openSession();

```

- 3、通过sqlsession执行数据库操作

- 可以通过 SqlSession 实例来直接执行已映射的 SQL 语句:

```

1 | Blog blog =
2 | (Blog)session.selectOne("org.mybatis.example.BlogMapper
3 | .selectBlog", 101);

```

- 更常用的方式是先获取Mapper(映射), 然后再执行SQL语句:

```
1 BlogMapper mapper =  
session.getMapper(BlogMapper.class);  
2 Blog blog = mapper.selectBlog(101);
```

- 4、调用session.commit()提交事务

如果是更新、删除语句, 我们还需要提交一下事务。

- 5、调用session.close()关闭会话

最后一定要记得关闭会话。

MyBatis生命周期?

上面提到了几个MyBatis的组件, 一般说的MyBatis生命周期就是这些组件的生命周期。

- SqlSessionFactoryBuilder

一旦创建了 SqlSessionFactory, 就不再需要它了。因此 SqlSessionFactoryBuilder 实例的生命周期只存在于方法的内部。

- SqlSessionFactory

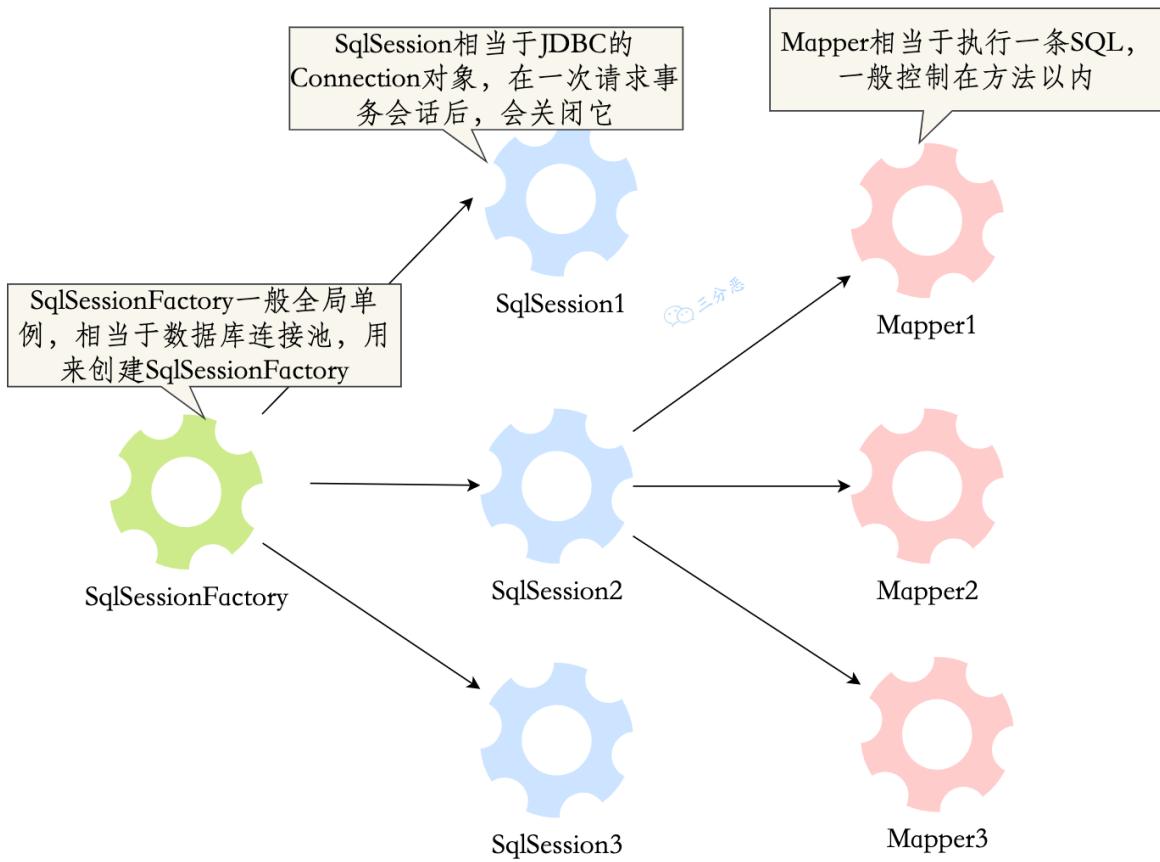
SqlSessionFactory 是用来创建SqlSession的, 相当于一个数据库连接池, 每次创建SqlSessionFactory都会使用数据库资源, 多次创建和销毁是对资源的浪费。所以SqlSessionFactory是应用级的生命周期, 而且应该是单例的。

- SqlSession

SqlSession相当于JDBC中的Connection, SqlSession 的实例不是线程安全的, 因此是不能被共享的, 所以它的最佳的生命周期是一次请求或一个方法。

- Mapper

映射器是一些绑定映射语句的接口。映射器接口的实例是从 SqlSession 中获得的, 它的生命周期在sqlsession事务方法之内, 一般会控制在方法级。



当然，万物皆可集成Spring，MyBatis通常也是和Spring集成使用，Spring可以帮助我们创建线程安全的、基于事务的 SqlSession 和映射器，并将它们直接注入到我们的 bean 中，我们不需要关心它们的创建过程和生命周期，那就是另外的故事了。

ps：接下来看看Mybatis的基本使用，不会有人不会吧？不会吧！

这都不会回家种地吧



4.在mapper中如何传递多个参数？

Mapper传多个参数

顺序传参法

@Param注解传参法

Map传参法

Java Bean传参法

方法1：顺序传参法

```
1 public User selectUser(String name, int deptId);  
2  
3     <select id="selectUser" resultMap="UserResultMap">  
4         select * from user  
5         where user_name = #{0} and dept_id = #{1}  
6     </select>
```

- # {}里面的数字代表传入参数的顺序。
- 这种方法不建议使用，sql层表达不直观，且一旦顺序调整容易出错。

方法2：@Param注解传参法

```
1 public User selectUser(@Param("userName") String name, int  
2 @Param("deptId") deptId);  
3  
4     <select id="selectUser" resultMap="UserResultMap">  
5         select * from user  
6         where user_name = #{userName} and dept_id = #{deptId}  
7     </select>
```

- # {}里面的名称对应的是注解@Param括号里面修饰的名称。
- 这种方法在参数不多的情况下还是比较直观的，（推荐使用）。

方法3：Map传参法

```

1  public User selectUser(Map<String, Object> params);
2
3      <select id="selectUser" parameterType="java.util.Map"
4          resultMap="UserResultMap">
5          select * from user
6          where user_name = #{userName} and dept_id = #{deptId}
7      </select>

```

- # {}里面的名称对应的是Map里面的key名称。
- 这种方法适合传递多个参数，且参数易变能灵活传递的情况。

方法4：Java Bean传参法

```

1  public User selectUser(User user);
2
3      <select id="selectUser" parameterType="com.jourwon.pojo.User"
4          resultMap="UserResultMap">
5          select * from user
6          where user_name = #{userName} and dept_id = #{deptId}
7      </select>

```

- # {}里面的名称对应的是User类里面的成员属性。
- 这种方法直观，需要建一个实体类，扩展不容易，需要加属性，但代码可读性强，业务逻辑处理方便，推荐使用。（推荐使用）。

5.实体类属性名和表中字段名不一样，怎么办？

- 第1种：通过在查询的SQL语句中定义字段名的别名，让字段名的别名和实体类的属性名一致。

```

1  <select id="getOrder" parameterType="int"
2      resultType="com.jourwon.pojo.Order">
3          select order_id id, order_no orderno ,order_price
4          price form orders where order_id=#{id};
5      </select>
6

```

- 第2种：通过resultMap 中的<result>来映射字段名和实体类属性名的一一对应的关系。

```

1 <select id="getOrder" parameterType="int"
2   resultMap="orderResultMap">
3     select * from orders where order_id=#{id}
4   </select>
5
6   <resultMap type="com.jourwon.pojo.Order"
7     id="orderResultMap">
8     <!--用id属性来映射主键字段-->
9     <id property="id" column="order_id">
10    <!--用result属性来映射非主键字段, property为实体类属性名,
11      column为数据库表中的属性-->
12    <result property ="orderno" column ="order_no"/>
13    <result property="price" column="order_price" />
14  </reslutMap>

```

6.Mybatis是否可以映射Enum枚举类？

- Mybatis当然可以映射枚举类，不单可以映射枚举类，Mybatis可以映射任何对象到表的一列上。映射方式为自定义一个TypeHandler，实现TypeHandler的setParameter()和getResult()接口方法。
- TypeHandler有两个作用，一是完成从javaType至jdbcType的转换，二是完成jdbcType至javaType的转换，体现为setParameter()和getResult()两个方法，分别代表设置sql问号占位符参数和获取列查询结果。

7.#{ }和\${ }的区别？

- # {}是占位符，预编译处理；\${ }是拼接符，字符串替换，没有预编译处理。
- Mybatis在处理#{}时，#{}传入参数是以字符串传入，会将SQL中的#{}替换为?号，调用PreparedStatement的set方法来赋值。
- # {} 可以有效的防止SQL注入，提高系统安全性；\${ } 不能防止SQL注入
- # {} 的变量替换是在DBMS 中；\${ } 的变量替换是在 DBMS 外

8.模糊查询like语句该怎么写?



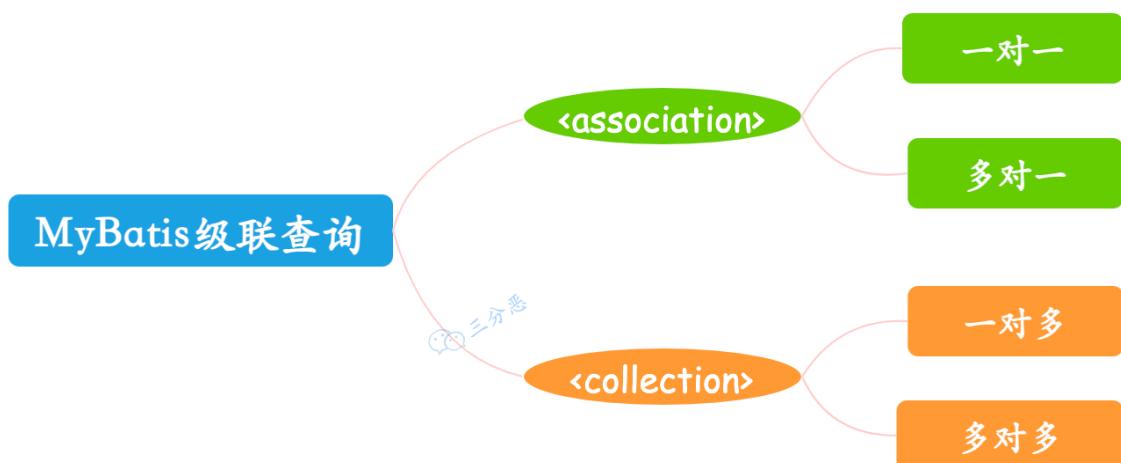
CONCAT('%',#{question},'%')

- 1 '%\${question}%' 可能引起SQL注入，不推荐
- 2 "%#\${question}%" 注意：因为#{...}解析成sql语句时候，会在变量外侧自动加单引号''，所以这里 % 需要使用双引号" "，不能使用单引号 ''，不然会查不到任何结果。
- 3 CONCAT('%',#{question},'%') 使用CONCAT()函数，（推荐）
- 4 使用bind标签（不推荐）

```
1 <select id="listUserLikeUsername"
2   resultType="com.jourwon.pojo.User">
3     <bind name="pattern" value="'" + username + "'" />
4       select id,sex,age,username,password from person where
      username LIKE #{pattern}
6   </select>
```

9.Mybatis能执行一对一、一对多的关联查询吗？

当然可以，不止支持一对一、一对多的关联查询，还支持多对多、多对一的关联查询。



- 一对~~一~~—**<association>**

比如订单和支付是一对一的关系，这种关联的实现：

- 实体类：

```
1 public class Order {  
2     private Integer orderId;  
3     private String orderDesc;  
4  
5     /**  
6      * 支付对象  
7      */  
8     private Pay pay;  
9     // .....  
10 }
```

- 结果映射

```
1 <!-- 订单resultMap -->  
2 <resultMap id="peopleResultMap"  
3 type="cn.fighter3.entity.Order">  
4     <id property="orderId" column="order_id" />  
5     <result property="orderDesc" column="order_desc" />  
6     <!--一对一结果映射-->  
7     <association property="pay"  
8         javaType="cn.fighter3.entity.Pay">  
9         <id column="payId" property="pay_id"/>  
10        <result column="account" property="account" />  
11    </association>  
12 </resultMap>
```

- 查询就是普通的关联查

```
1     <select id="getTeacher" resultMap="getTeacherMap"  
2     parameterType="int">  
3         select * from order o  
4             left join pay p on o.order_id=p.order_id  
5             where o.order_id=#{orderId}  
6     </select>
```

- 一对多<collection>

比如商品分类和商品，是一对多的关系。

- 实体类

```
1 public class Category {  
2     private int categoryId;  
3     private String categoryName;  
4  
5     /**  
6      * 商品列表  
7      **/  
8     List<Product> products;  
9     // .....10}
```

- 结果映射

```
1         <resultMap type="Category" id="categoryBean">  
2             <id column="categoryId"  
3                 property="category_id" />  
4                 <result column="categoryName"  
5                 property="category_name" />  
6  
7                 <!-- 一对多的关系 -->  
8                 <!-- property: 指的是集合属性的值, ofType: 指的  
是集合中元素的类型 -->  
9                 <collection property="products"  
10                ofType="Product">  
11                    <id column="product_id"  
12                    property="productId" />  
13                    <result column="productName"  
14                    property="productName" />  
15                    <result column="price" property="price"  
16                />  
17            </collection>  
18        </resultMap>
```

- 查询

查询就是一个普通的关联查询

```

1      <!-- 关联查询分类和产品表 -->
2      <select id="listCategory"
3          resultMap="categoryBean">
4          select c.*, p.* from category_ c left
join product_ p on c.id = p.cid
</select>

```

那么多对一、多对多怎么实现呢？还是利用<association>和<collection>，篇幅所限，这里就不展开了。

10.Mybatis是否支持延迟加载？原理？

- Mybatis支持association关联对象和collection关联集合对象的延迟加载，association指的就是一对一，collection指的就是一对多查询。在Mybatis配置文件中，可以配置是否启用延迟加载lazyLoadingEnabled=true|false。
- 它的原理是，使用CGLIB创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用a.getB().getName()，拦截器invoke()方法发现a.getB()是null值，那么就会单独发送事先保存好的查询关联B对象的sql，把B查询上来，然后调用a.setB(b)，于是a的对象b属性就有值了，接着完成a.getB().getName()方法的调用。这就是延迟加载的基本原理。
- 当然了，不光是Mybatis，几乎所有的包括Hibernate，支持延迟加载的原理都是一样的。

11.如何获取生成的主键？

- 新增标签中添加：keyProperty=" ID " 即可

```

1      <insert id="insert" useGeneratedKeys="true"
2          keyProperty="userId" >
3          insert into user(
4              user_name, user_password, create_time)
5              values(#{userName}, #{userPassword} , #{createTime,
jdbcType= TIMESTAMP})
6      </insert>

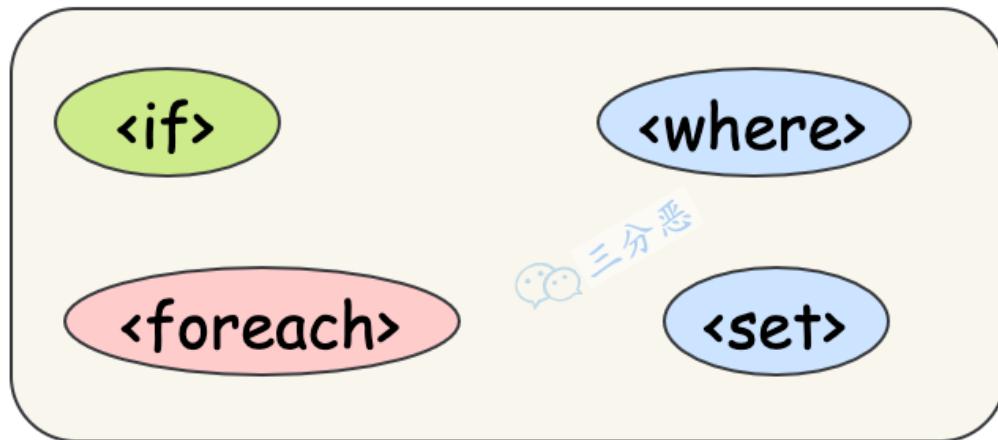
```

- 这时候就可以完成回填主键

```
1 | mapper.insert(user);  
2 | user.getId;
```

12.MyBatis支持动态SQL吗？

MyBatis中有一些支持动态SQL的标签，它们的原理是使用OGNL从SQL参数对象中计算表达式的值，根据表达式的值动态拼接SQL，以此来完成动态SQL的功能。



- if

根据条件来组成where子句

```
1 | <select id="findActiveBlogWithTitleLike"  
2 |     resultType="Blog">  
3 |     SELECT * FROM BLOG  
4 |     WHERE state = 'ACTIVE'  
5 |     <if test="title != null">  
6 |         AND title like #{title}  
7 |     </if>  
8 | </select>
```

- choose (when, otherwise)

这个和Java 中的 switch 语句有点像

```
1 | <select id="findActiveBlogLike"  
2 |     resultType="Blog">  
3 |     SELECT * FROM BLOG WHERE state = 'ACTIVE'
```

```
4 <choose>
5   <when test="title != null">
6     AND title like #{title}
7   </when>
8   <when test="author != null and author.name != null">
9     AND author_name like #{author.name}
10  </when>
11  <otherwise>
12    AND featured = 1
13  </otherwise>
14 </choose>
15 </select>
```

- trim (where, set)
 - <where>可以用在所有的查询条件都是动态的情况

```
1 <select id="findActiveBlogLike"
2   resultType="Blog">
3   SELECT * FROM BLOG
4   <where>
5     <if test="state != null">
6       state = #{state}
7     </if>
8     <if test="title != null">
9       AND title like #{title}
10    </if>
11    <if test="author != null and author.name != null">
12      AND author_name like #{author.name}
13    </if>
14   </where>
15 </select>
```

- <set> 可以用在动态更新的时候

```
1 <update id="updateAuthorIfNecessary">
2     update Author
3         <set>
4             <if test="username != null">username=#{username},
5             </if>
6             <if test="password != null">password=#{password},
7             </if>
8             <if test="email != null">email=#{email}, </if>
9             <if test="bio != null">bio=#{bio}</if>
10            </set>
11        where id=#{id}
12    </update>
```

- foreach

看到名字就知道了，这个是用来循环的，可以对集合进行遍历

```
1 <select id="selectPostIn" resultType="domain.blog.Post">
2     SELECT *
3     FROM POST P
4     <where>
5         <foreach item="item" index="index" collection="list"
6             open="ID in (" separator="," close=")"
7             nullable="true">
8             #{item}
9         </foreach>
10    </where>
11 </select>
```

13.MyBatis如何执行批量操作？

MyBatis批量操作

<foreach>标签

ExecutorType.BATCH



第一种方法：使用foreach标签

foreach的主要用在构建in条件中，它可以在SQL语句中进行迭代一个集合。foreach标签的属性主要有item, index, collection, open, separator, close。

- item 表示集合中每一个元素进行迭代时的别名，随便起的变量名；
- index 指定一个名字，用于表示在迭代过程中，每次迭代到的位置，不常用；
- open 表示该语句以什么开始，常用“(”；
- separator 表示在每次进行迭代之间以什么符号作为分隔符，常用“,”；
- close 表示以什么结束，常用“)”。

在使用foreach的时候最关键的也是最容易出错的就是collection属性，该属性是必须指定的，但是在不同情况下，该属性的值是不一样的，主要有以下3种情况：

1. 如果传入的是单参数且参数类型是一个List的时候，collection属性值为list
2. 如果传入的是单参数且参数类型是一个array数组的时候，collection的属性值为array
3. 如果传入的参数是多个的时候，我们就需要把它们封装成一个Map了，当然单参数也可以封装成map，实际上如果你在传入参数的时候，在MyBatis里面也是会把它封装成一个Map的，
map的key就是参数名，所以这个时候collection属性值就是传入的List或array对象在自己封装的map里面的key

看看批量保存的两种用法：

```

1 <!-- MySQL下批量保存，可以foreach遍历 mysql支持values(),(),()语法
--> //推荐使用
2 <insert id="addEmpsBatch">
3     INSERT INTO emp(ename,gender,email,did)
4     VALUES
5     <foreach collection="emps" item="emp" separator="," >
6         (#{emp.eName},#{emp.gender},#{emp.email},#
{emp.dept.id})
7     </foreach>
8 </insert>

```

```

1 <!-- 这种方式需要数据库连接属性allowMultiQueries=true的支持
2 如jdbc.url=jdbc:mysql://localhost:3306/mybatis?
allowMultiQueries=true -->
3 <insert id="addEmpsBatch">
4     <foreach collection="emps" item="emp" separator=";" >
5         INSERT INTO emp(ename,gender,email,did)
6         VALUES(#{emp.eName},#{emp.gender},#{emp.email},#
{emp.dept.id})
7     </foreach>
8 </insert>

```

第二种方法：使用ExecutorType.BATCH

- Mybatis内置的ExecutorType有3种，默認為simple，该模式下它为每个语句的执行创建一个新的预处理语句，单条提交sql；而batch模式重复使用已经预处理的语句，并且批量执行所有更新语句，显然batch性能将更优；但batch模式也有自己的问题，比如在Insert操作时，在事务没有提交之前，是没有办法获取到自增的id，在某些情况下不符合业务的需求。

具体用法如下：

```

1 //批量保存方法测试
2 @Test
3 public void testBatch() throws IOException{
4     SqlSessionFactory sqlSessionFactory =
getSqlSessionFactory();
5     //可以执行批量操作的sqlSession
6     SqlSession openSession =
sqlSessionFactory.openSession(ExecutorType.BATCH);
7

```

```

8 //批量保存执行前时间
9 long start = System.currentTimeMillis();
10 try {
11     EmployeeMapper mapper =
12     openSession.getMapper(EmployeeMapper.class);
13     for (int i = 0; i < 1000; i++) {
14         mapper.addEmp(new
15             Employee(UUID.randomUUID().toString().substring(0, 5),
16             "b", "1"));
17     }
18
19     openSession.commit();
20     long end = System.currentTimeMillis();
21     //批量保存执行后的时间
22     System.out.println("执行时长" + (end - start));
23     //批量 预编译sql一次==>设置参数==> 10000次==> 执行1次
24     677
25     //非批量 (预编译=设置参数=执行 ) ==> 10000次 1121
26 }
```

- mapper和mapper.xml如下

```

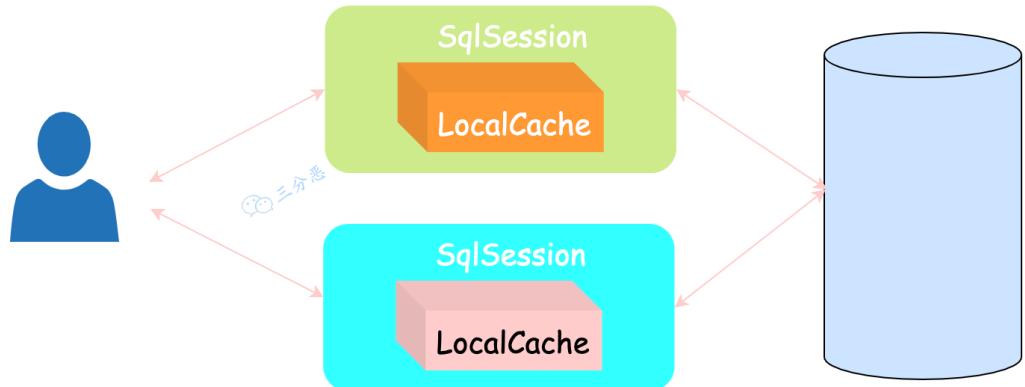
1 public interface EmployeeMapper {
2     //批量保存员工
3     Long addEmp(Employee employee);
4 }
```

```

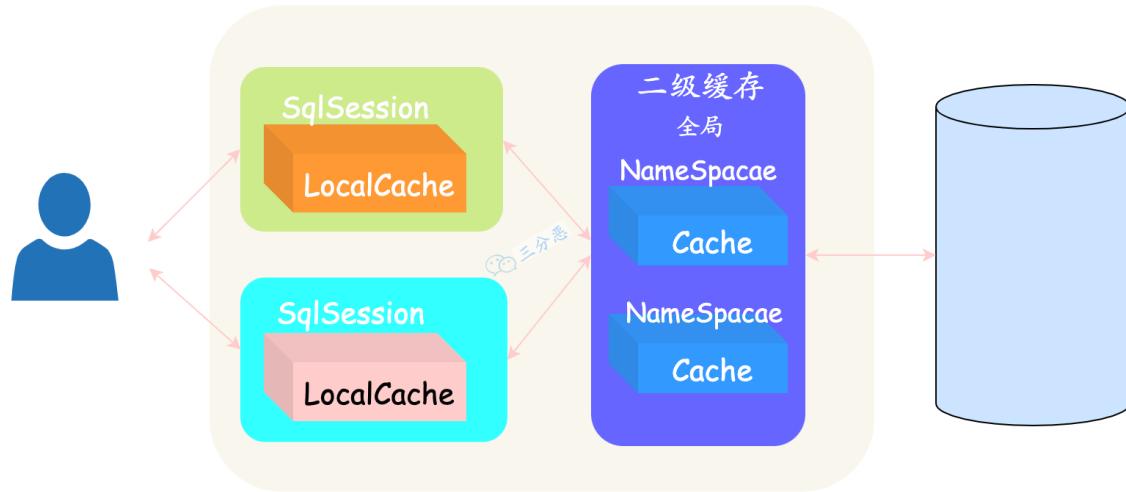
1 <mapper namespace="com.jourwon.mapper.EmployeeMapper"
2     <!--批量保存员工 -->
3     <insert id="addEmp">
4         insert into employee(lastName,email,gender)
5             values(#{$lastName},#{email},#{gender})
6     </insert>
7 </mapper>
```

14. 说说Mybatis的一级、二级缓存？

1. 一级缓存：基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 SqlSession，各个SqlSession之间的缓存相互隔离，当 Session flush 或 close 之后，该 SqlSession 中的所有 Cache 就将清空，MyBatis默认打开一级缓存。



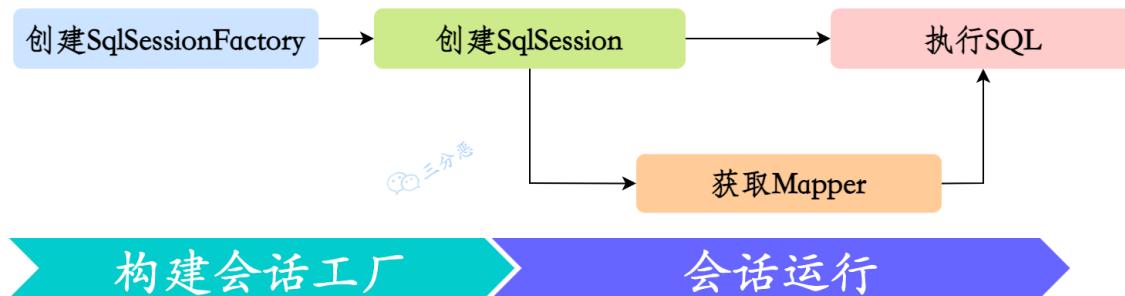
2. 二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同之处在于其存储作用域为 Mapper(Namespace)，可以在多个SqlSession 之间共享，并且可自定义存储源，如 Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现Serializable序列化接口(可用来保存对象的状态)，可在它的映射文件中配置。



原理

15. 能说说MyBatis的工作原理吗？

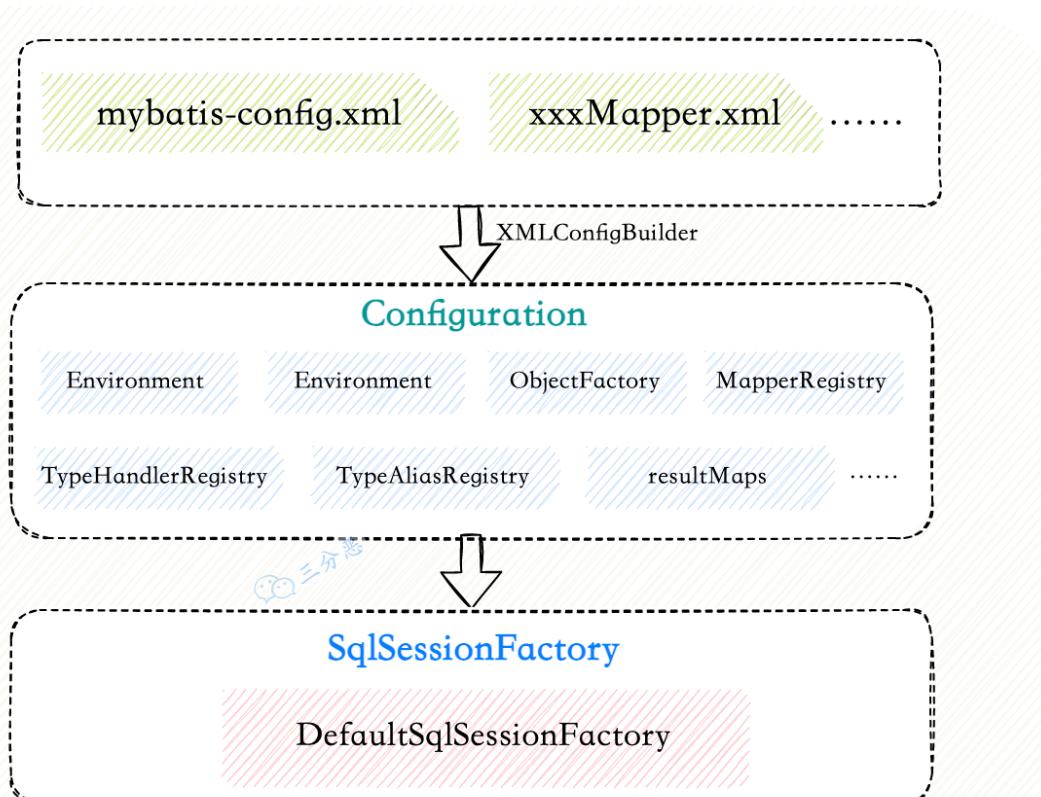
我们已经大概知道了MyBatis的工作流程，按工作原理，可以分为两大步：**生成会话工厂**、**会话运行**。



MyBatis是一个成熟的框架，篇幅限制，这里抓大放小，来看看它的主要工作流程。

构建会话工厂

构造会话工厂也可以分为两步：



- 获取配置

获取配置这一步经过了几步转化，最终由生成了一个配置类Configuration实例，这个配置类实例非常重要，主要作用包括：

- 读取配置文件，包括基础配置文件和映射文件

- 初始化基础配置，比如MyBatis的别名，还有其它的一些重要的类对象，像插件、映射器、ObjectFactory等等
- 提供一个单例，作为会话工厂构建的重要参数
- 它的构建过程也会初始化一些环境变量，比如数据源

```
1 public SqlSessionFactory build(Reader reader, String
2   environment, Properties properties) {
3     SqlSessionFactory var5;
4     //省略异常处理
5     //xml配置构建器
6     XMLConfigBuilder parser = new
7     XMLConfigBuilder(reader, environment, properties);
8     //通过转化的Configuration构建SqlSessionFactory
9     var5 = this.build(parser.parse());
10 }
```

- 构建SqlSessionFactory

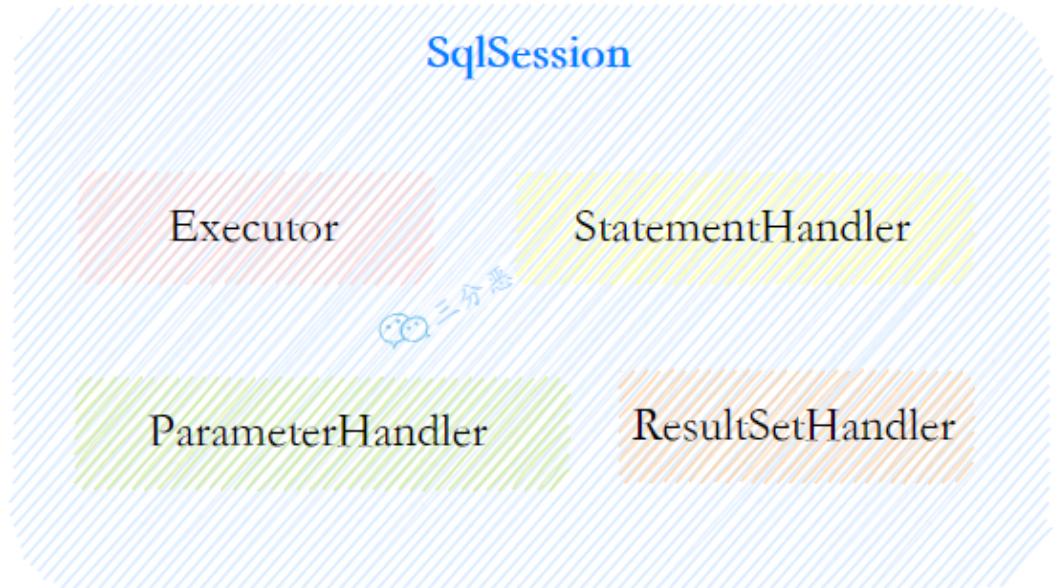
SqlSessionFactory只是一个接口，构建出来的实际上是它的实现类的实例，一般我们用的都是它的实现类DefaultSqlSessionFactory，

```
1 public SqlSessionFactory build(Configuration config) {
2   return new DefaultSqlSessionFactory(config);
3 }
```

会话运行

会话运行是MyBatis最复杂的部分，它的运行离不开四大组件的配合：

SqlSession



- Executor（执行器）

Executor起到了至关重要的作用，SqlSession只是一个门面，相当于客服，真正干活的是Executor，就像是默默无闻的工程师。它提供了相应的查询和更新方法，以及事务方法。

```
1     Environment environment =
2         this.configuration.getEnvironment();
3         TransactionFactory transactionFactory =
4             this.getTransactionFactoryFromEnvironment(environment);
5             tx =
6             transactionFactory.newTransaction(environment.getDataSourc
e(), level, autoCommit);
7             //通过Configuration创建executor
8             Executor executor =
9             this.configuration.newExecutor(tx, execType);
10            var8 = new
11            DefaultSqlSession(this.configuration, executor,
12            autoCommit);
```

- StatementHandler（数据库会话器）

StatementHandler，顾名思义，处理数据库会话的。我们以SimpleExecutor为例，看一下它的查询方法，先生成了一个StatementHandler实例，再拿这个handler去执行query。

```
1 |     public <E> List<E> doQuery(MappedStatement ms, Object
2 |     parameter, RowBounds rowBounds, ResultHandler
3 |     resultHandler, BoundSql boundSql) throws SQLException {
4 |         Statement stmt = null;
5 |
6 |         List var9;
7 |         try {
8 |             Configuration configuration =
9 |             ms.getConfiguration();
10 |             StatementHandler handler =
11 |             configuration.newStatementHandler(this.wrapper, ms,
12 |             parameter, rowBounds, resultHandler, boundSql);
13 |             stmt = this.prepareStatement(handler,
14 |             ms.getStatementLog());
15 |             var9 = handler.query(stmt, resultHandler);
16 |         } finally {
17 |             this.closeStatement(stmt);
18 |         }
19 |     }
20 |
21 |     return var9;
22 | }
```

再以最常用的PreparedStatementHandler看一下它的query方法，其实在上面的

```
prepareStatement
```

已经对参数进行了预编译处理，到了这里，就直接执行sql，使用ResultHandler处理返回结果。

```
1 |     public <E> List<E> query(Statement statement,
2 |     ResultHandler resultHandler) throws SQLException {
3 |         PreparedStatement ps =
4 |         (PreparedStatement)statement;
5 |         ps.execute();
6 |         return this.resultSetHandler.handleResultSets(ps);
7 |     }
```

- ParameterHandler（参数处理器）

PreparedStatementHandler里对sql进行了预编译处理

```
1 |     public void parameterize(Statement statement) throws
2 |     SQLException {
3 |
4 |         this.parameterHandler.setParameters((PreparedStatement)sta
5 |         tement);
6 |     }
```

这里用的就是ParameterHandler， setParameters的作用就是设置预编译SQL语句的参数。

里面还会用到typeHandler类型处理器，对类型进行处理。

```
1 |     public interface ParameterHandler {
2 |         Object getParameterObject();
3 |
4 |         void setParameters(PreparedStatement var1) throws
5 |         SQLException;
6 |     }
```

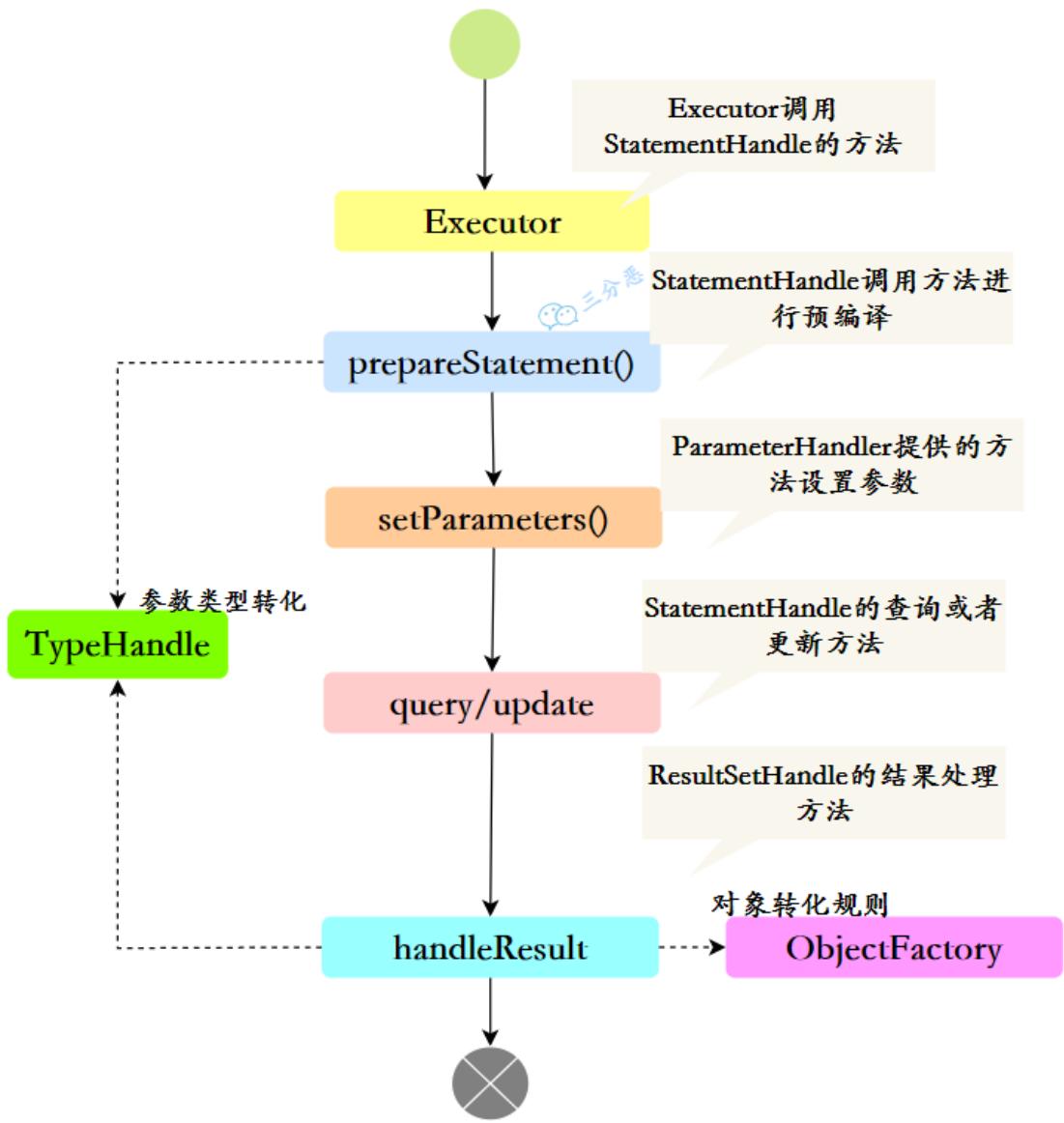
- ResultSetHandler（结果处理器）

我们前面也看到了，最后的结果要通过ResultSetHandler来进行处理， handleResultSets这个方法就是用来包装结果集的。Mybatis为我们提供了一个 DefaultResultSetHandler，通常都是用这个实现类去进行结果的处理的。

```
1 |     public interface ResultSetHandler {
2 |         <E> List<E> handleResultSets(Statement var1) throws
3 |         SQLException;
4 |
5 |         <E> Cursor<E> handleCursorResultSets(Statement var1)
6 |         throws SQLException;
7 |
8 |         void handleOutputParameters(CallableStatement var1)
9 |         throws SQLException;
10 |    }
```

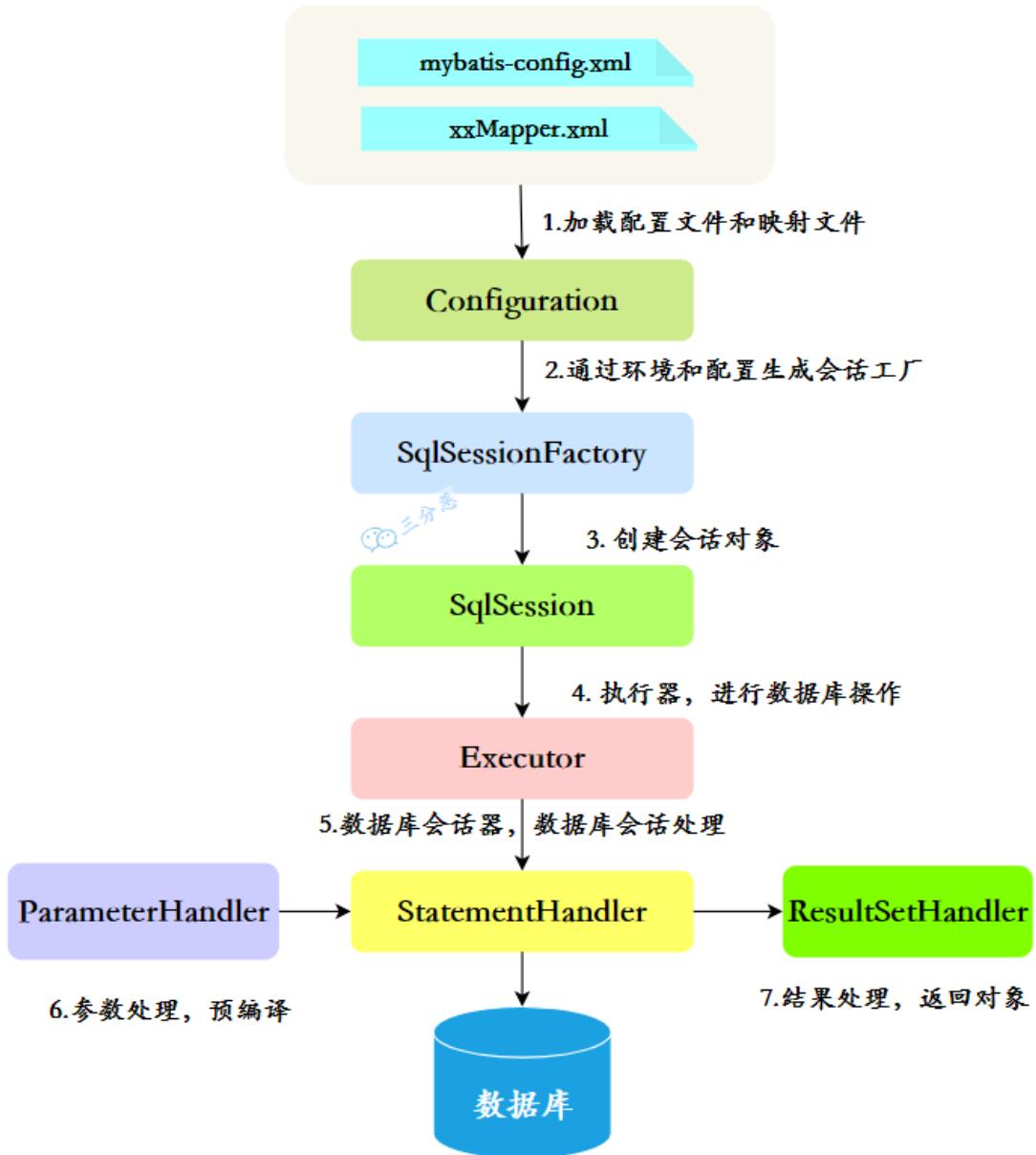
它会使用typeHandle处理类型，然后用ObjectFactory提供的规则组装对象，返回给调用者。

整体上总结一下会话运行：



PS：以上源码分析比较简单，在真正的源码大佬面前可能过不了关，有条件的建议Debug一下MyBatis的源码。

我们最后把整个的工作流程串联起来，简单总结一下：



1. 读取 MyBatis 配置文件——mybatis-config.xml、加载映射文件——映射文件即 SQL 映射文件，文件中配置了操作数据库的 SQL 语句。最后生成一个配置对象。
2. 构造会话工厂：通过 MyBatis 的环境等配置信息构建会话工厂 SqlSessionFactory。
3. 创建会话对象：由会话工厂创建 SqlSession 对象，该对象中包含了执行 SQL 语句的所有方法。
4. Executor 执行器：MyBatis 底层定义了一个 Executor 接口来操作数据库，它将根据 SqlSession 传递的参数动态地生成需要执行的 SQL 语句，同时负责查询缓存的维护。
5. StatementHandler：数据库会话器，串联起参数映射的处理和运行结果映射的处理。
6. 参数处理：对输入参数的类型进行处理，并预编译。
7. 结果处理：对返回结果的类型进行处理，根据对象映射规则，返回相应的对象。

16.MyBatis的功能架构是什么样的？

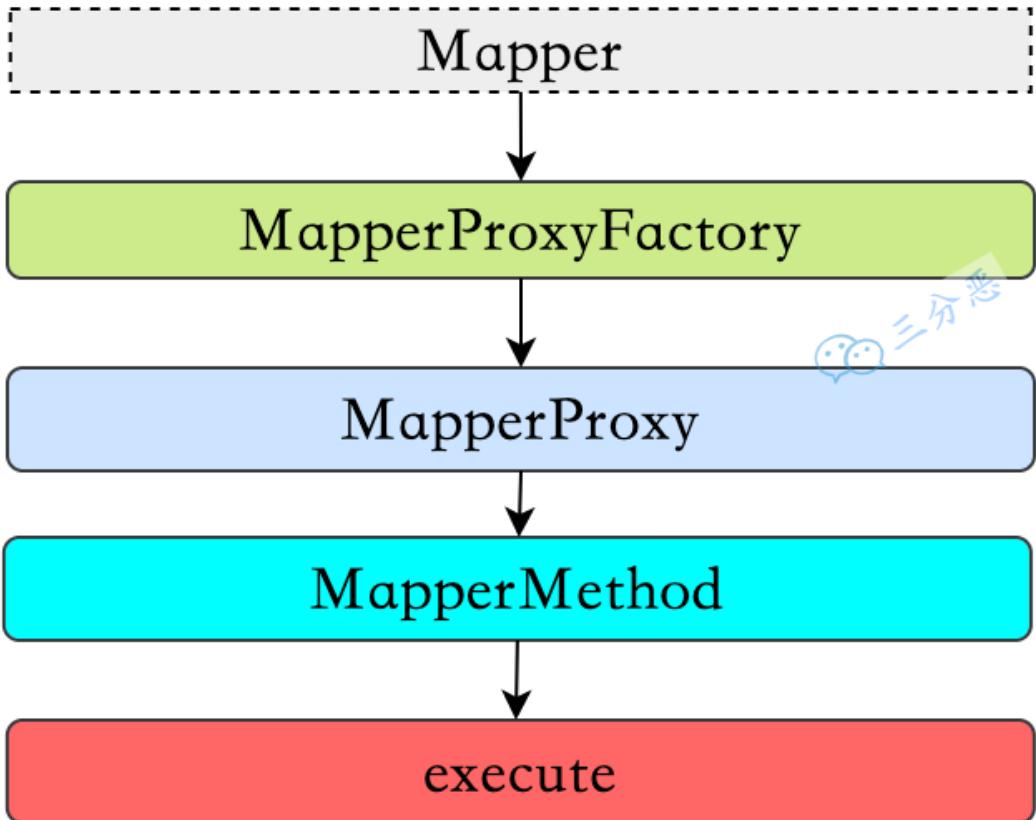


我们一般把Mybatis的功能架构分为三层：

- API接口层：提供给外部使用的接口API，开发人员通过这些本地API来操纵数据库。接口层一接收到调用请求就会调用数据处理层来完成具体的数据处理。
- 数据处理层：负责具体的SQL查找、SQL解析、SQL执行和执行结果映射处理等。它主要的目的是根据调用的请求完成一次数据库操作。
- 基础支撑层：负责最基础的功能支撑，包括连接管理、事务管理、配置加载和缓存处理，这些都是共用的东西，将他们抽取出来作为最基础的组件。为上层的数据处理层提供最基础的支撑。

17.为什么Mapper接口不需要实现类？

四个字回答：**动态代理**，我们来看一下获取Mapper的过程：



- 获取Mapper

我们都知道定义的Mapper接口是没有实现类的，Mapper映射其实是通过**动态代理**实现的。

```
1 | BlogMapper mapper = session.getMapper(BlogMapper.class);
```

七拐八绕地进去看一下，发现获取Mapper的过程，需要先获取MapperProxyFactory——Mapper代理工厂。

```

1   public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
2       MapperProxyFactory<T> mapperProxyFactory =
3           (MapperProxyFactory)this.knownMappers.get(type);
4       if (mapperProxyFactory == null) {
5           throw new BindingException("Type " + type + " "
6               + "is not known to the MapperRegistry.");
7       } else {
8           try {
9               return
10          mapperProxyFactory.newInstance(sqlSession);
11      } catch (Exception var5) {
12          throw new BindingException("Error getting
13          mapper instance. Cause: " + var5, var5);
14      }
15     }
16   }

```

- MapperProxyFactory

MapperProxyFactory的作用是生成MapperProxy（Mapper代理对象）。

```

1  public class MapperProxyFactory<T> {
2      private final Class<T> mapperInterface;
3
4      .....
5      protected T newInstance(MapperProxy<T> mapperProxy) {
6          return
7              Proxy.newProxyInstance(this.mapperInterface.getClassLoader(),
8                  new Class[]{this.mapperInterface}, mapperProxy);
9      }
10
11
12      public T newInstance(SqlSession sqlSession) {
13          MapperProxy<T> mapperProxy = new
14              MapperProxy(sqlSession, this.mapperInterface,
15                  this.methodCache);
16          return this.newInstance(mapperProxy);
17      }
18  }

```

这里可以看到动态代理对接口的绑定，它的作用就是生成动态代理对象（占位），而代理的方法被放到了MapperProxy中。

- MapperProxy

MapperProxy里，通常会生成一个MapperMethod对象，它是通过cachedMapperMethod方法对其进行初始化的，然后执行execute方法。

```

1   public Object invoke(Object proxy, Method method,
2   Object[] args) throws Throwable {
3       try {
4           return
5               Object.class.equals(method.getDeclaringClass()) ?
6               method.invoke(this, args) :
7               this.cachedInvoker(method).invoke(proxy, method, args,
8               this.sqlSession);
9       } catch (Throwable var5) {
10           throw ExceptionUtil.unwrapThrowable(var5);
11       }
12   }

```

- MapperMethod

MapperMethod里的execute方法，会真正去执行sql。这里用到了命令模式，其实绕一圈，最终它还是通过SqlSession的实例去运行对象的sql。

```

1   public Object execute(SqlSession sqlSession, Object[]
2   args) {
3       Object result;
4       Object param;
5       .....
6       case SELECT:
7           if (this.method.returnsVoid() &&
8               this.method.hasResultHandler()) {
9
9       this.executeWithResultHandler(sqlSession, args);
10          result = null;
11      } else if (this.method.returnsMany()) {
12          result =
13      this.executeForMany(sqlSession, args);
14      } else if (this.method.returnsMap()) {
15          result = this.executeForMap(sqlSession,
16          args);
17      } else if (this.method.returnsCursor()) {
18

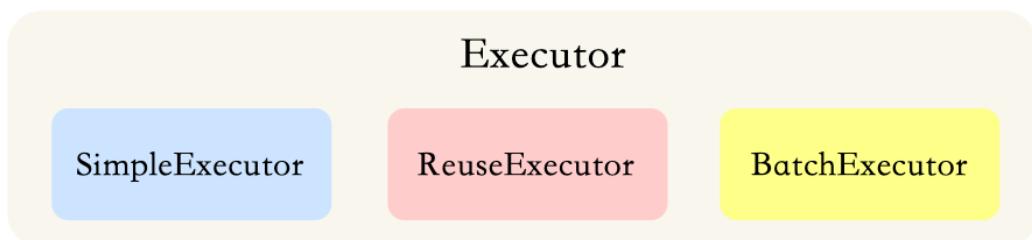
```

```

14         result =
15     this.executeForCursor(sqlSession, args);
16     } else {
17         param =
18         this.method.convertArgsToSqlCommandParam(args);
19         result =
20         sqlSession.selectOne(this.command.getName(), param);
21         if (this.method>ReturnsOptional() &&
22             (result == null ||
23             !this.method.getReturnType().equals(result.getClass())))
24         {
25             result =
26             Optional.ofNullable(result);
27         }
28     }
29     break;
30     .....
31 }

```

18.Mybatis都有哪些Executor执行器？



Mybatis有三种基本的Executor执行器，SimpleExecutor、ReuseExecutor、BatchExecutor。

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

作用范围：Executor的这些特点，都严格限制在SqlSession生命周期范围内。

Mybatis中如何指定使用哪一种Executor执行器？

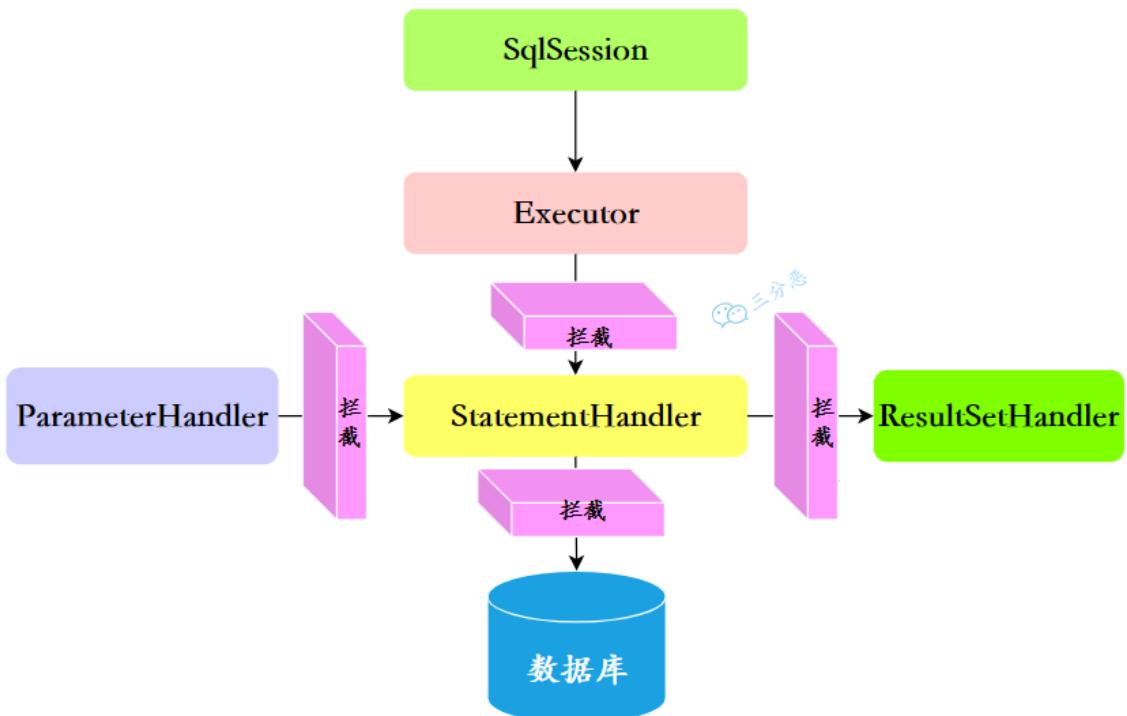
- 在Mybatis配置文件中，在设置（settings）可以指定默认的ExecutorType执行器类型，也可以手动给DefaultSqlSessionFactory的创建SqlSession的方法传递ExecutorType类型参数，如SqlSession openSession(ExecutorType execType)。
- 配置默认的执行器。SIMPLE 就是普通的执行器； REUSE 执行器会重用预处理语句（prepared statements）； BATCH 执行器将重用语句并执行批量更新。

插件

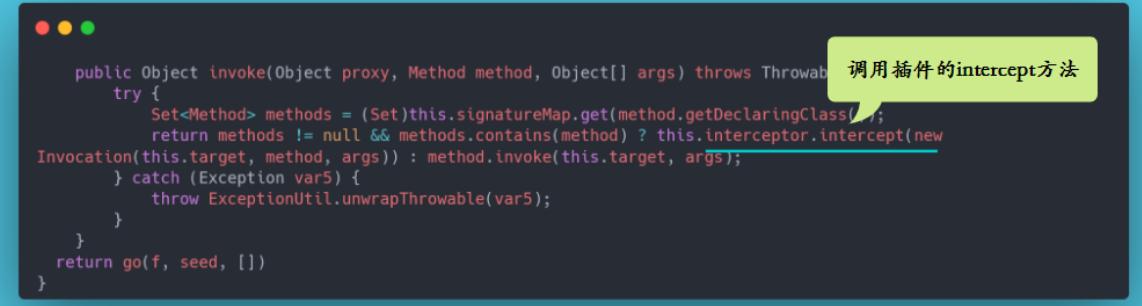
19.说说Mybatis的插件运行原理，如何编写一个插件？

插件的运行原理？

Mybatis会话的运行需要ParameterHandler、ResultSetHandler、StatementHandler、Executor这四大对象的配合，插件的原理就是在这四大对象调度的时候，插入一些我们自己的代码。



Mybatis使用JDK的动态代理，为目标对象生成代理对象。它提供了一个工具类 `Plugin`，实现了 `InvocationHandler` 接口。



```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    try {
        Set<Method> methods = (Set)this.signatureMap.get(method.getDeclaringClass());
        return methods != null && methods.contains(method) ? this.interceptor.intercept(new
Invocation(this.target, method, args)) : method.invoke(this.target, args);
    } catch (Exception var5) {
        throw ExceptionUtil.unwrapThrowable(var5);
    }
}
return go(f, seed, []);
}
```

使用 **Plugin** 生成代理对象，代理对象在调用方法的时候，就会进入`invoke`方法，在`invoke`方法中，如果存在签名的拦截方法，插件的`intercept`方法就会在这里被我们调用，然后就返回结果。如果不存在签名方法，那么将直接反射调用我们要执行的方法。

如何编写一个插件？

我们自己编写MyBatis 插件，只需要实现拦截器接口 `Interceptor` (`org.apache.ibatis.plugin Interceptor`)，在实现类中对拦截对象和方法进行处理。

- 实现Mybatis的`Interceptor`接口并重写`intercept()`方法

这里我们只是在目标对象执行目标方法的前后进行了打印；

```
1 public class MyInterceptor implements Interceptor {
2     Properties props=null;
3
4     @Override
5     public Object intercept(Invocation invocation) throws
Throwable {
6         System.out.println("before.....");
7         //如果当前代理的是一个非代理对象，那么就会调用真实拦截对象的
方法
8         // 如果不是它就会调用下个插件代理对象的invoke方法
9         Object obj=invocation.proceed();
10        System.out.println("after.....");
11        return obj;
12    }
13 }
```

- 然后再给插件编写注解，确定要拦截的对象，要拦截的方法

```
1 @Intercepts({@Signature(
```

```

2     type = Executor.class, //确定要拦截的对象
3     method = "update",      //确定要拦截的方法
4     args = {MappedStatement.class, Object.class}    //拦截方法的参数
5   })
6   public class MyInterceptor implements Interceptor {
7     Properties props=null;
8
9     @Override
10    public Object intercept(Invocation invocation) throws
11      Throwable {
12      System.out.println("before.....");
13      //如果当前代理的是一个非代理对象，那么就会调用真实拦截对象的
14      //如果不是它就会调用下个插件代理对象的invoke方法
15      Object obj=invocation.proceed();
16      System.out.println("after.....");
17      return obj;
18    }
19  }

```

- 最后，再MyBatis配置文件里面配置插件

```

1 <plugins>
2   <plugin interceptor="xxx.MyPlugin">
3     <property name="dbType",value="mysql"/>
4   </plugin>
5 </plugins>

```

20.MyBatis是如何进行分页的？分页插件的原理是什么？

MyBatis是如何分页的？

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

分页插件的原理是什么？

- 分页插件的基本原理是使用Mybatis提供的插件接口，实现自定义插件，拦截Executor的query方法
- 在执行查询的时候，拦截待执行的sql，然后重写sql，根据dialect方言，添加对应的物理分页语句和物理分页参数。
- 举例：select * from student，拦截sql后重写为：select t.* from (select * from student) t limit 0, 10

可以看一下一个大概的MyBatis通用分页拦截器：

```
    @SuppressWarnings({"rawtypes", "unchecked"})
    @Intercepts(
        @Signature(
            type = Executor.class,           //拦截对象
            method = "query",               //拦截方法
            args = {MappedStatement.class, Object.class,
                    RowBounds.class, ResultHandler.class}))}

public class PageInterceptor implements Interceptor {

    private static final List<ResultMapping> EMPTY_RESULTMAPPING = new ArrayList<ResultMapping>();

    //数据库方言接口
    private Dialect dialect;
    private Field additionalParametersField;

    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        //获取拦截方法的参数
        Object[] args = invocation.getArgs();
        MappedStatement ms = (MappedStatement) args[0];
        Object parameterObject = args[1];
        RowBounds rowBounds = (RowBounds) args[2];
        //调用方法判断是否需要进行分页，如果不需要，直接返回结果
        if (!dialect.skip(ms.getId(), parameterObject, rowBounds)) {
            ResultHandler resultHandler = (ResultHandler) args[3];
            //当前的目标对象
            Executor executor = (Executor) invocation.getTarget();
            BoundSql boundSql = ms.getBoundSql(parameterObject);
            //反射获取动态参数
            Map<String, Object> additionalParameters = (Map<String, Object>) additionalParametersField.get(boundSql);
            //判断是否需要进行 count 查询
            if (dialect.beforeCount(ms.getId(), parameterObject, rowBounds)) {
                //根据当前的 ms 创建一个返回值为 Long 类型的 ms
                MappedStatement countMs = new MappedStatement(ms, Long.class);
                //创建count查询的缓存 key
                CacheKey countKey = executor.createCacheKey(
                    countMs,
                    parameterObject,
                    RowBounds.DEFAULT,
                    boundSql);
                //调用方言获取count sql
                String countSql = dialect.getCountSql(
                    boundSql,
                    parameterObject,
                    rowBounds,
                    countKey);
                BoundSql countBoundSql = new BoundSql(
                    ms.getConfiguration(),
                    countSql,
                    boundSql.getParameterMappings(),
                    parameterObject);
                //当使用动态SQL时，可能会产生临时的参数
                //这些参数需要手动设置到新的 BoundSql 中
                for (String key : additionalParameters.keySet()) {
                    countBoundSql.setAdditionalParameter(
                        key, additionalParameters.get(key));
                }
                //执行 count 查询
                Object countResultList = executor.query(
                    countMs,
                    parameterObject,
                    RowBounds.DEFAULT,
                    resultHandler,
                    countKey,
                    countBoundSql);
                Long count = (Long) ((List) countResultList).get(0);
                //处理查询总数
                dialect.afterCount(count, parameterObject, rowBounds);
                if (count == 0L) {
                    //当查询总数为 0 时，直接返回空的结果
                    return dialect.afterPage(
                        new ArrayList<>());
                }
            }
        }
    }
}
```

```

        new ArrayList(),
        parameterObject,
        rowBounds
    );
}
// 判断是否需要进行分页查询
if (dialect.beforePage(ms.getId(), parameterObject, rowBounds)) {
    //生成分页的缓存 key
    CacheKey pageKey = executor.createCacheKey(
        ms,
        parameterObject,
        rowBounds,
        boundSql);
    //调用方言获取分页 sql
    String pageSql = dialect.getPageSql(
        boundSql,
        parameterObject,
        rowBounds,
        pageKey);
    BoundSql pageBoundSql = new BoundSql(
        ms.getConfiguration(),
        pageSql,
        boundSql.getParameterMappings(),
        parameterObject);
    //设置动态参数
    for (String key : additionalParameters.keySet()) {
        pageBoundSql.setAdditionalParameter(
            key, additionalParameters.get(key));
    }
    //执行分页查询
    List resultList = executor.query(
        ms,
        parameterObject,
        RowBounds.DEFAULT,
        resultHandler,
        pageKey,
        pageBoundSql);
    return dialect.afterPage(resultList, parameterObject, rowBounds);
}
}
//返回默认查询
return invocation.proceed();
}

/**
 * 根据现有的 ms 创建一个新的返回值类型，使用新的返回值类型
 *
 * @param ms
 * @param resultType
 * @return
 */
private MappedStatement newMappedStatement(MappedStatement ms, Class<?> resultType) {
    MappedStatement.Builder builder = new MappedStatement.Builder(
        ms.getConfiguration(),
        ms.getId() + "_Count",
        ms.getSqlSource(),
        ms.getSqlCommandType());
    builder.resource(ms.getResource());
    builder.fetchSize(ms.getFetchSize());
    builder.statementType(ms.getStatementType());
    builder.keyGenerator(ms.getKeyGenerator());
    if (ms.getKeyProperties() != null)
        if (ms.getKeyProperties().length != 0) {
            StringBuilder keyProperties = new StringBuilder();
            for (String keyProperty : ms.getKeyProperties()) {
                keyProperties.append(keyProperty).append(",");
            }
            keyProperties.delete(keyProperties.length() - 1, keyProperties.length());
            builder.keyProperty(keyProperties.toString());
        }
    builder.timeout(ms.getTimeout());
    builder.parameterMap(ms.getParameterMap());
    //count 查询返回值 int
    List<ResultMap> resultMaps = new ArrayList<ResultMap>();
    ResultMap resultMap = new ResultMap.Builder(
        ms.getConfiguration()

```

```
        ms.getConfiguration(),
        ms.getId(),
        resultType,
        EMPTY_RESULTMAPPING).build();
    resultMaps.add(resultMap);
    builder.resultMaps(resultMaps);
    builder.resultSetType(ms.getResultsetType());
    builder.cache(ms.getCache());
    builder.flushCacheRequired(ms.isFlushCacheRequired());
    builder.useCache(ms.isUseCache());
    return builder.build();
}

@Override
public Object plugin(Object target) {
    return Plugin.wrap(target, this);
}

@Override
public void setProperties(Properties properties) {
    String dialectClass = properties.getProperty("dialect");
    try {
        dialect = (Dialect) Class.forName(dialectClass).newInstance();
    } catch (Exception e) {
        throw new RuntimeException("使用PageInterceptor分页插件时，必须设置dialect属性");
    }
    dialect.setProperties(properties);
    try {
        //反射获取BoundSql中的additionalParameters属性
        additionalParametersField = BoundSql.class.getDeclaredField("additionalParameters");
    } catch (NoSuchFieldException e) {
        throw new RuntimeException(e);
    }
}
}
```

参考

- [1]. [MyBatis面试题（2020最新版）](#)
 - [2]. [mybatis官网](#)
 - [3]. 《深入浅出MyBatis基础原理与实战》
 - [4]. [聊聊MyBatis缓存机制](#)
 - [5]. 《MyBatis从入门到精通》
-

关注公众号：三分恶

手册更新动态

即刻送达



添加个人微信：ThirdFighter

技术交流

加大佬云集微信群



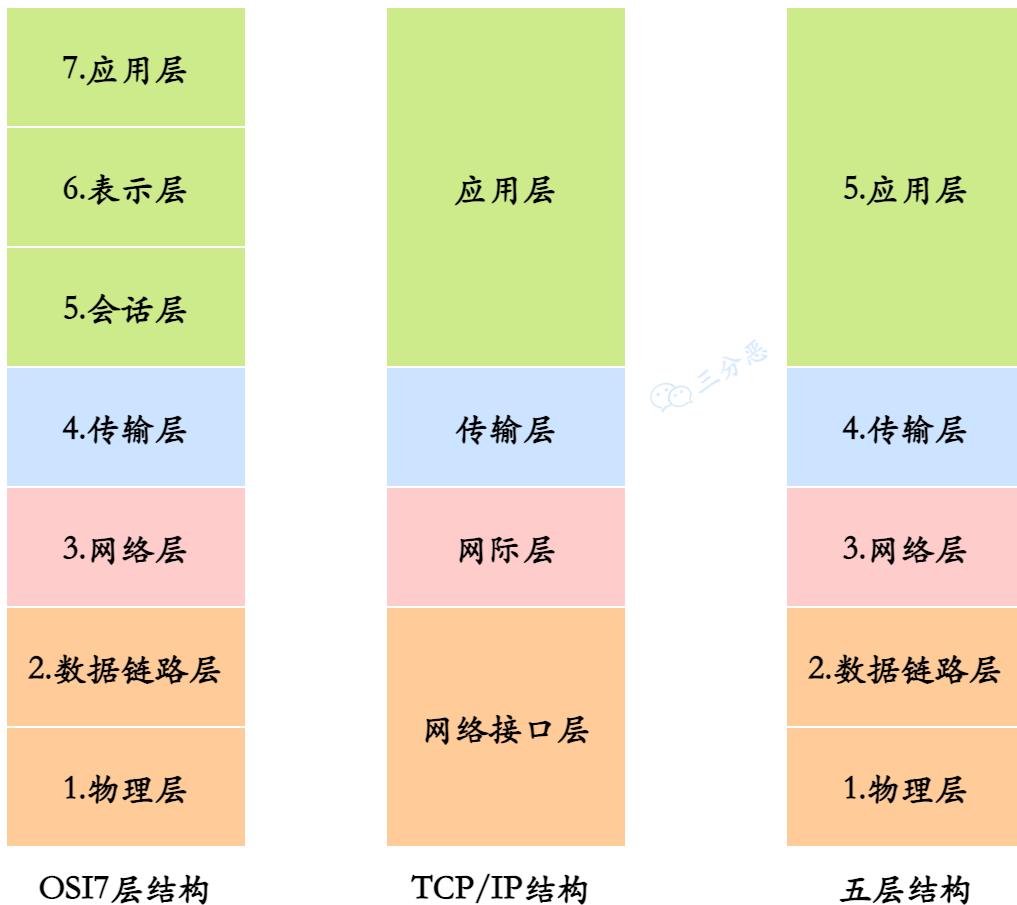
第三部分：计算机基础

一、计算机网路

基础

1.说下计算机网络体系结构

计算机网络体系结构，一般有三种：OSI 七层模型、TCP/IP 四层模型、五层结构。



简单说，OSI是一个理论上的网络通信模型，TCP/IP是实际上的网络通信模型，五层结构就是为了介绍网络原理而折中的网络通信模型。

OSI 七层模型

OSI 七层模型是国际标准化组织（International Organization for Standardization）制定的一个用于计算机或通信系统间互联的标准体系。

- **应用层：**通过应用进程之间的交互来完成特定网络应用，应用层协议定义的是应用进程间通信和交互的规则，常见的协议有： HTTP FTP SMTP SNMP DNS .
- **表示层：**数据的表示、安全、压缩。确保一个系统的应用层所发送的信息可以被另一个系统的应用层读取。
- **会话层：**建立、管理、终止会话，是用户应用程序和网络之间的接口。
- **运输层：**提供源端与目的端之间提供可靠的透明数据传输，传输层协议为不同主机上运行的进程提供逻辑通信。
- **网络层：**将网络地址翻译成对应的物理地址，实现不同网络之间的路径选择，协议有 ICMP IGMP IP 等 .
- **数据链路层：**在物理层提供比特流服务的基础上，建立相邻结点之间的数据链路。
- **物理层：**建立、维护、断开物理连接。

TCP/IP 四层模型

- 应用层：对应于 OSI 参考模型的（应用层、表示层、会话层）。
- 传输层：对应 OSI 的传输层，为应用层实体提供端到端的通信功能，保证了数据包的顺序传送及数据的完整性。
- 网际层：对应于 OSI 参考模型的网络层，主要解决主机到主机的通信问题。
- 网络接口层：与 OSI 参考模型的数据链路层、物理层对应。

五层体系结构

- 应用层：对应于 OSI 参考模型的（应用层、表示层、会话层）。
- 传输层：对应 OSI 参考模型的的传输层
- 网络层：对应 OSI 参考模型的的网络层
- 数据链路层：对应 OSI 参考模型的的数据链路层
- 物理层：对应 OSI 参考模型的物理层。

2.说一下每一层对应的网络协议有哪些？

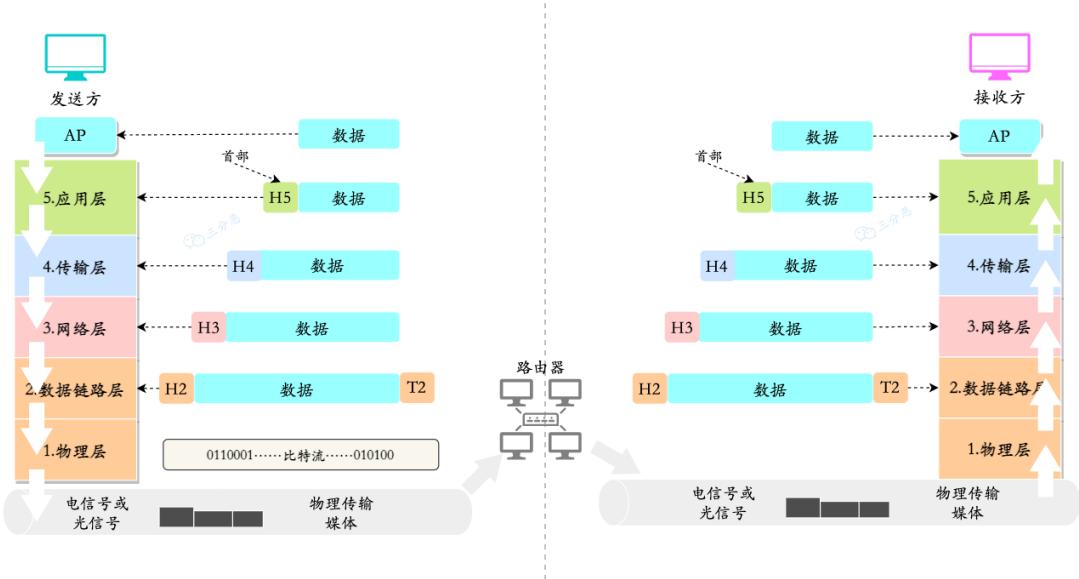
一张表格总结常见网络协议：

OSI七层网络模型	TCP/IP四层模型	对应的网络协议
应用层	应用层	HTTP、DNS、FTP、NFS、WAIS、SMIP、Telnet、SNMP
表示层		TIFF、GIF、JPEG、PICT
会话层		RPC、SQL、NFS、NetBIOS、names、AppleTalk
传输层	传输层	TCP、UDP
网络层	网络层	IP、ICMP、ARP、RAPP、RIP、IPX
数据链路层	网络接口层	FDDI、Frame Relay、HDLC、PPP
物理层		EIA/TIA-232、EIA/TIA-499

3.那么数据在各层之间是怎么传输的呢？

对于发送方而言，从上层到下层层层包装，对于接收方而言，从下层到上层，层层解开包装。

- 发送方的应用进程向接收方的应用进程传送数据
- AP先将数据交给本主机的应用层，应用层加上本层的控制信息H5就变成了下一层的数据单元
- 传输层收到这个数据单元后，加上本层的控制信息H4，再交给网络层，成为网络层的数据单元
- 到了数据链路层，控制信息被分成两部分，分别加到本层数据单元的首部（H2）和尾部（T2）
- 最后的物理层，进行比特流的传输



这个过程类似写信，写一封信，每到一层，就加一个信封，写一些地址的信息。到了目的地之后，又一层层解封，传向下一个目的地。

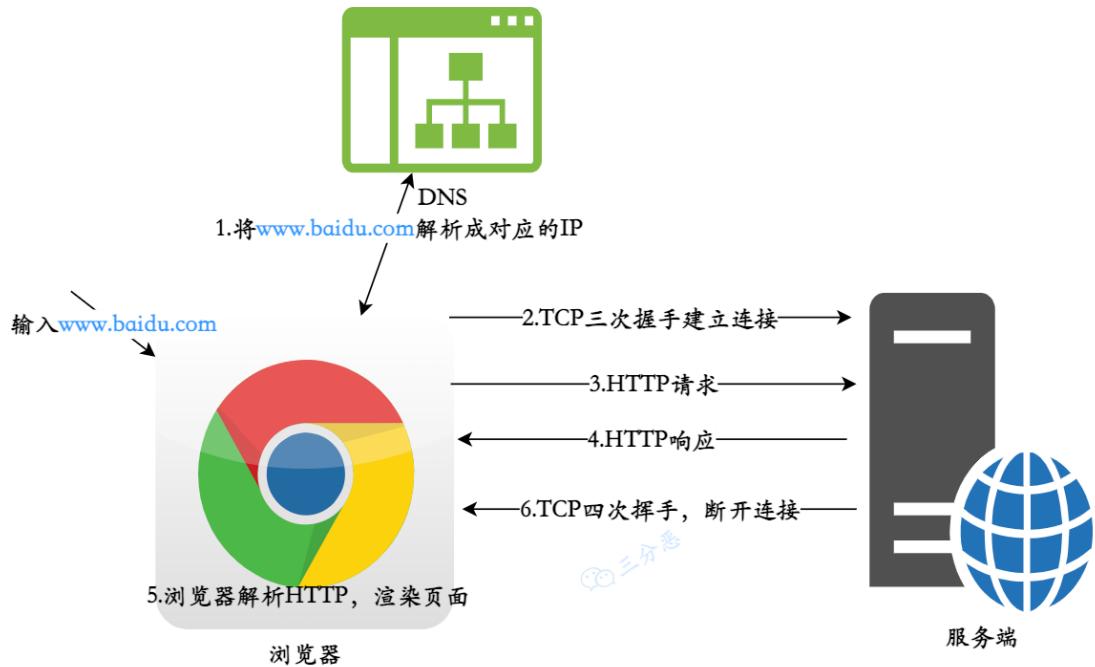
网络综合

4.从浏览器地址栏输入 url 到显示主页的过程？

这道题，大概的过程比较简单，但是有很多点可以细挖：DNS解析、TCP三次握手、HTTP报文格式、TCP四次挥手等等。

1. DNS 解析：将域名解析成对应的 IP 地址。
2. TCP连接：与服务器通过三次握手，建立 TCP 连接
3. 向服务器发送 HTTP 请求
4. 服务器处理请求，返回HTTP响应
5. 浏览器解析并渲染页面
6. 断开连接：TCP 四次挥手，连接结束

我们以输入 www.baidu.com 为例：



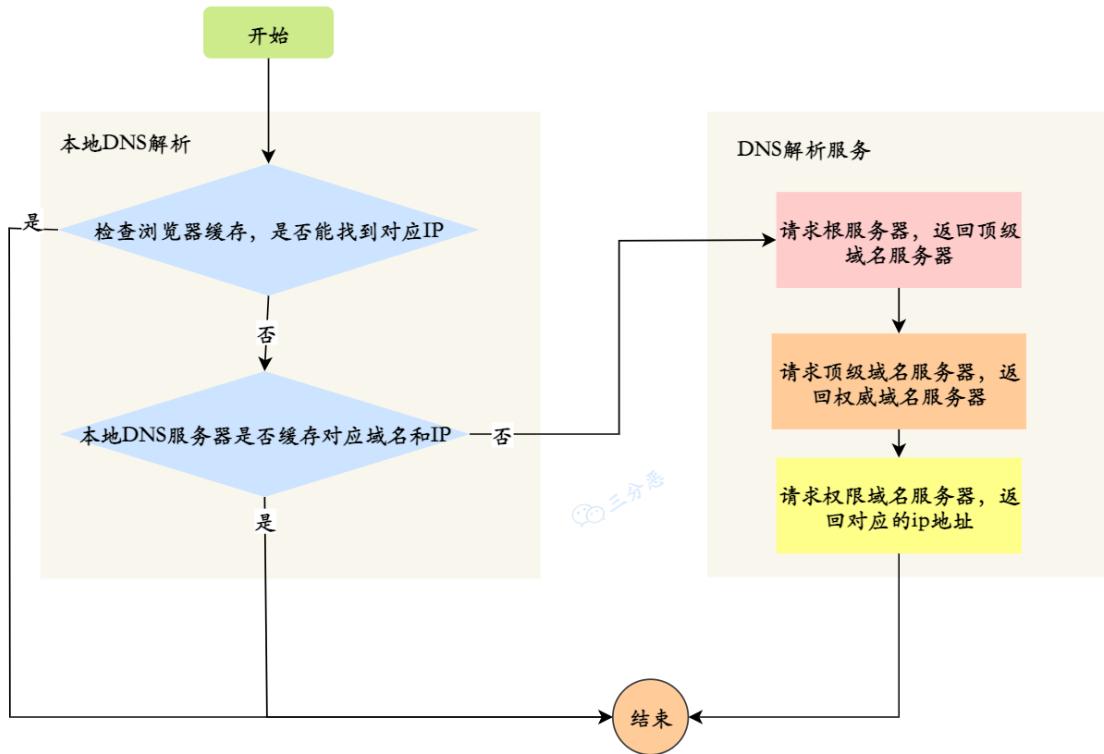
各个过程都使用了哪些协议？

过程	使用的协议
1. 浏览器查找域名的IP地址 (DNS查找过程：浏览器缓存、本地DNS路由缓存、DNS解析服务)	DNS: 获取域名对应的IP
2. 浏览器和服务端TCP三次握手，建立连接	TCP: 与服务端建立连接和断开连接
3. 浏览器向服务端发起一个HTTP请求	IP: 使用TCP协议时，网络层需要使用IP协议。 OSPF: IP数据包在路由器之间，路由选择使用OSPF协议
4. 服务端处理请求，返回HTTP响应	ARP: 路由器再与服务器通信时，需要将IP地址转换为MAC地址，需要使用ARP协议
5. 浏览器解析并渲染页面	HTTP: TCP连接建立完成之后，使用HTTP协议传递HTTP报文
6. 断开连接	

5. 说说 DNS 的解析过程？

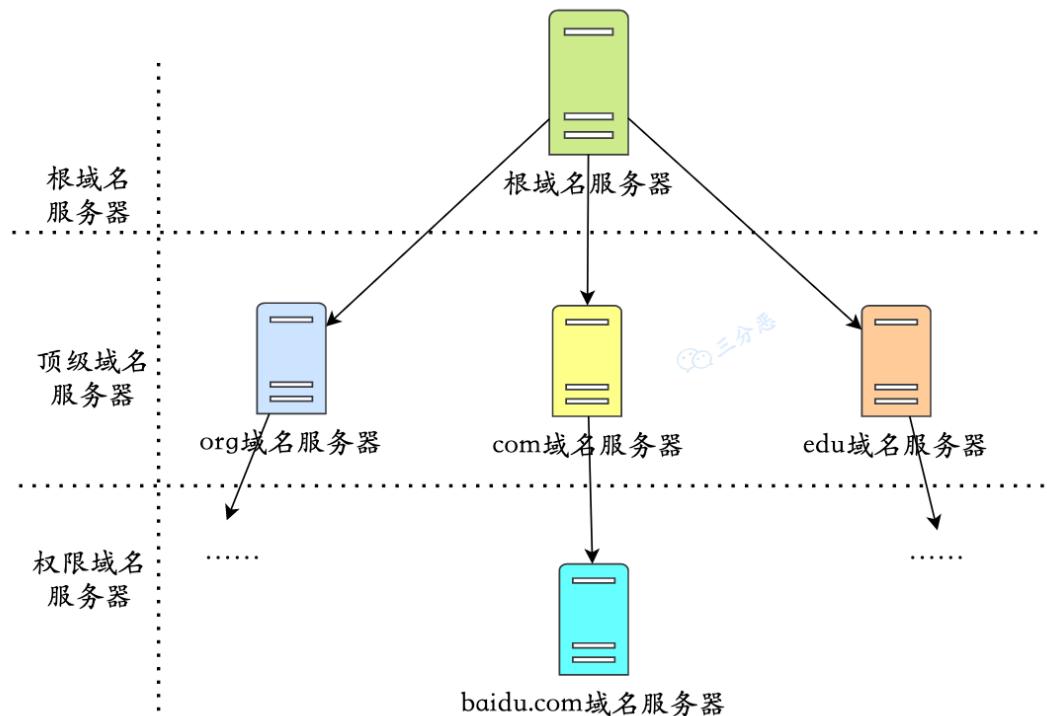
DNS，英文全称是 **domain name system**，域名解析系统，它的作用也很明确，就是域名和 IP 相互映射。

DNS 的解析过程如下图：



假设你要查询 www.baidu.com 的 IP 地址：

- 首先会查找浏览器的缓存,看看是否能找到 www.baidu.com 对应的IP地址, 找到就直接返回; 否则进行下一步。
- 将请求发往给本地DNS服务器, 如果查找到也直接返回, 否则继续进行下一步;



- 本地DNS服务器向 根域名服务器 发送请求, 根域名服务器返回负责 com 的顶级域名服务器的IP地址的列表。

- 本地DNS服务器再向其中一个负责 `com` 的顶级域名服务器发送一个请求，返回负责 `baidu.com` 的权限域名服务器的IP地址列表。
- 本地DNS服务器再向其中一个权限域名服务器发送一个请求，返回 www.baidu.com 所对应的IP地址。

6.说说 WebSocket 与 Socket 的区别？

- Socket 其实就是等于 IP 地址 + 端口 + 协议。

具体来说，Socket 是一套标准，它完成了对 TCP/IP 的高度封装，屏蔽网络细节，以方便开发者更好地进行网络编程。

- WebSocket 是一个持久化的协议，它是伴随 H5 而出的协议，用来解决 http 不支持持久化连接的问题。
- Socket 一个是网编编程的标准接口，而 WebSocket 则是应用层通信协议。

7.说一下你了解的端口及对应的服务？

端口	服务
21	FTP(文件传输协议)
22	SSH
23	Telnet(远程登录服务)
53	DNS域名解析服务
80	HTTP超文本传输协议
443	HTTPS
1080	Sockets
3306	MySQL默认端口号

HTTP

8. 说说 HTTP 常用的状态码及其含义？

HTTP状态码首先应该知道个大概的分类：

- 1XX：信息性状态码
- 2XX：成功状态码
- 3XX：重定向状态码

- 4XX: 客户端错误状态码
- 5XX: 服务端错误状态码

几个常用的，面试之外，也应该记住：

状态码	含义
101	切换请求协议
200	请求成功
301	请求资源永久移动，返回新URI
302	请求资源临时移动，浏览器重定向到新的URI
400	客户端请求的语法错误，服务端无法理解
401	当前请求需要认证
403	服务端理解请求，但拒绝执行
500	服务器内部错误

之前写过一篇：[程序员五一被拉去相亲，结果彻底搞懂了HTTP常用状态码](#)，还比较有意思，可以看看。

说一下301和302的区别？

- 301: 永久性移动，请求的资源已被永久移动到新位置。服务器返回此响应时，会返回新的资源地址。
- 302: 临时性移动，服务器从另外的地址响应资源，但是客户端还应该使用这个地址。

用一个比喻，301就是嫁人的新垣结衣，302就是有男朋友的长泽雅美。

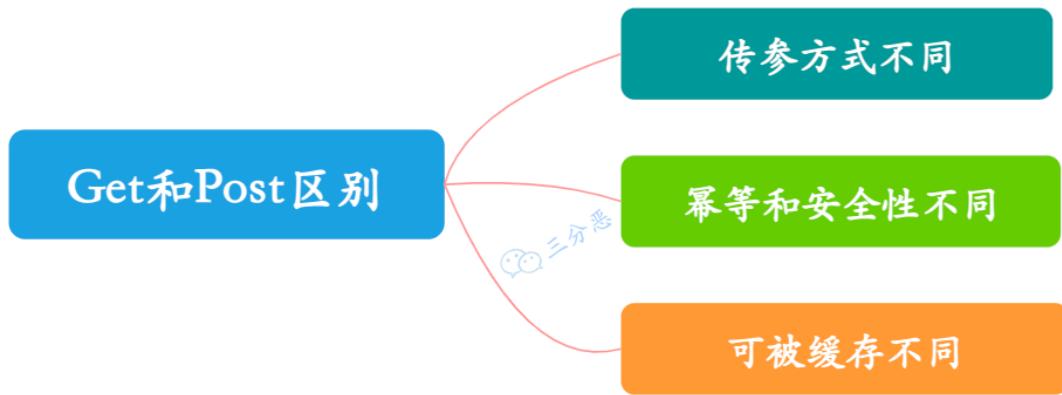
9.HTTP 有哪些请求方式？

请求方式	描述
GET	对服务器获取资源的简单请求
POST	向服务器提交数据请求
PUT	修改指定资源
DELETE	删除URL标记的指定资源
CONNECT	用于代理服务器
TRACE	主要用于回环测试
OPTIONS	返回所有可用的方法
HEAD	获取URL标记资源的首部

其中，POST、DELETE、PUT、GET的含义分别对应我们最熟悉的增、删、改、查。

10.说一下 GET 和 POST 的区别？

可以从以下几个方面来说明GET和POST的区别：



1. 从 HTTP 报文层面来看，GET 请求将信息放在 URL，POST 将请求信息放在请求体中。这一点使得 GET 请求携带的数据量有限，因为 URL 本身是有长度限制的，而 POST 请求的数据存放在报文体中，因此对大小没有限制。而且从形式上看，GET 请求把数据放 URL 上不太安全，而 POST 请求把数据放在请求体里想比较而言安全一些。
2. 从数据库层面来看，GET 符合幂等性和安全性，而 POST 请求不符合。这个其实和 GET/POST 请求的作用有关。按照 HTTP 的约定，GET 请求用于查看信息，不会改变服务器上的信息；而 POST 请求用来改变服务器上的信息。正因为 GET 请求只查看信息，不改变信息，对数据库的一次或多次操作获得的结果是一致的，认为它符合幂等性。安全性是指对数据库操作没有改变数据库中的数据。
3. 从其他层面来看，GET 请求能够被缓存，GET 请求能够保存在浏览器的浏览记录里，GET 请求的 URL 能够保存为浏览器书签。这些都是 POST 请求所不具备的。缓存是 GET 请求被广泛应用的根本，他能够被缓存也是因为它的幂等性和安全性，除了返回结果没有其他多余的动作，因此绝大部分的 GET 请求都被 CDN 缓存起来了，大大减少了 Web 服务器的负担。

11.GET 的长度限制是多少？

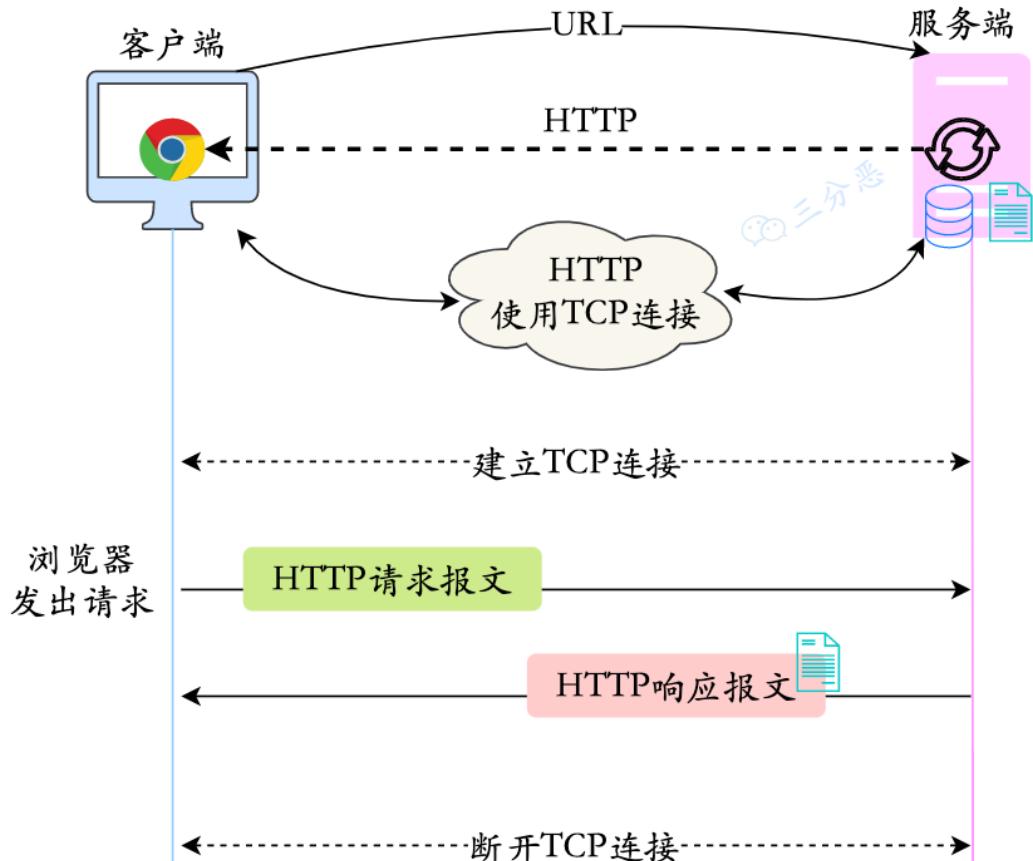
HTTP中的GET方法是通过URL传递数据的，但是URL本身其实并没有对数据的长度进行限制，真正限制GET长度的是浏览器。

例如IE浏览器对URL的最大限制是2000多个字符，大概2kb左右，像Chrome、Firefox等浏览器支持的URL字符数更多，其中FireFox中URL的最大长度限制是65536个字符，Chrome则是8182个字符。

这个长度限制也不是针对数据部分，而是针对整个URL。

12.HTTP 请求的过程与原理？

HTTP协议定义了浏览器怎么向服务器请求文档，以及服务器怎么把文档传给浏览器。



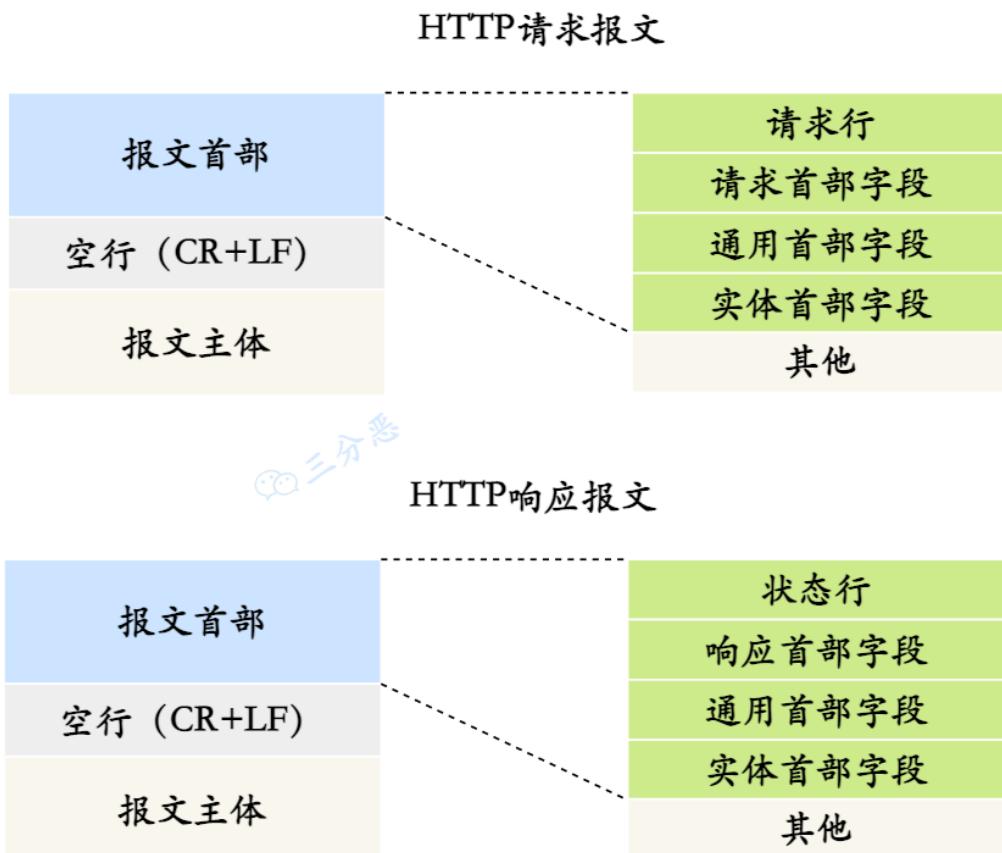
- 每个服务器都有一个进程，它不断监听TCP的端口80，以便发现是否有浏览器向它发出连接建立请求
- 监听到连接请求，就会建立TCP连接
- 浏览器向服务器发出浏览某个页面的请求，服务器接着就返回所请求的页面作为响应
- 最后，释放TCP连接

在浏览器和服务器之间的请求和响应的交互，必须按照规定的格式和遵循一定的规则，这些格式和规则就是超文本传输协议HTTP。

PS:这道题和上面浏览器输入网址发生了什么那道题大差不差。

13.说一下HTTP的报文结构？

HTTP报文有两种，HTTP请求报文和HTTP响应报文：



HTTP请求报文

HTTP 请求报文的格式如下：

```
1 | GET / HTTP/1.1
2 | User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5)
3 | Accept: */*
```

HTTP 请求报文的第一行叫做请求行，后面的行叫做首部行，首部行后还可以跟一个实体主体。请求首部之后有一个空行，这个空行不能省略，它用来划分首部与实体。

请求行包含三个字段：

- 方法字段：包括POST、GET等请方法。
- URL 字段
- HTTP 版本字段。

HTTP 响应报文

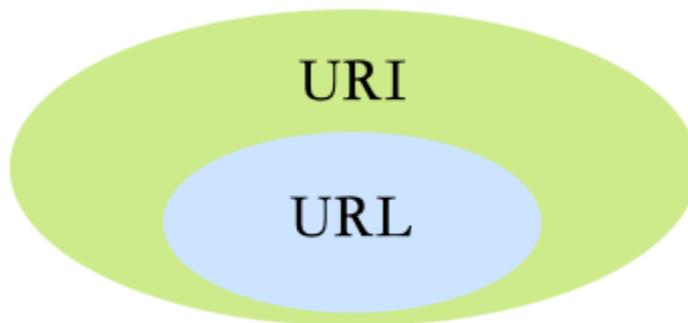
HTTP 响应报文的格式如下：

```
1 HTTP/1.0 200 OK
2 Content-Type: text/plain
3 Content-Length: 137582
4 Expires: Thu, 05 Dec 1997 16:00:00 GMT
5 Last-Modified: Wed, 5 August 1996 15:55:28 GMT
6 Server: Apache 0.84
7 <html>
8   <body>Hello World</body>
9 </html>
```

HTTP 响应报文的第一行叫做**状态行**，后面的行是**首部行**，最后是**实体主体**。

- **状态行**包含了三个字段：协议版本字段、状态码和相应的状态信息。
- **实体部分**是报文的主要部分，它包含了所请求的对象。
- **首部行**首部可以分为四种首部，请求首部、响应首部、通用首部和实体首部。
通用首部和实体首部在请求报文和响应报文中都可以设置，区别在于请求首部和响应首部。
 - 常见的请求首部有 Accept 可接收媒体资源的类型、Accept-Charset 可接收的字符集、Host 请求的主机名。
 - 常见的响应首部有 ETag 资源的匹配信息，Location 客户端重定向的 URI。
 - 常见的通用首部有 Cache-Control 控制缓存策略、Connection 管理持久连接。
 - 常见的实体首部有 Content-Length 实体主体的大小、Expires 实体主体的过期时间、Last-Modified 资源的最后修改时间。

14.URI 和 URL 有什么区别?



- URI，统一资源标识符(Uniform Resource Identifier，URI)，标识的是Web上每一种可用的资源，如 HTML文档、图像、视频片段、程序等都是由一个URI进行标识的。

- URL，统一资源定位符（Uniform Resource Location），它是URI的一种子集，主要作用是提供资源的路径。

它们的主要区别在于，URL除了提供了资源的标识，还提供了资源访问的方式。这么比喻，URI 像是身份证，可以唯一标识一个人，而 URL 更像一个住址，可以通过 URL 找到这个人——人类住址协议://地球/中国/北京市/海淀区/xx职业技术学院/14号宿舍楼/525号寝/张三.男。

15.说下 HTTP/1.0, 1.1, 2.0 的区别？

关键需要记住 **HTTP/1.0** 默认是短连接，可以强制开启，HTTP/1.1 默认长连接，HTTP/2.0 采用**多路复用**。

HTTP/1.0

- 默认使用 短连接，每次请求都需要建立一个 TCP 连接。它可以设置 `Connection: keep-alive` 这个字段，强制开启长连接。

HTTP/1.1

- 引入了持久连接，即 TCP 连接默认不关闭，可以被多个请求复用。
- 分块传输编码，即服务端每产生一块数据，就发送一块，用“流模式”取代“缓存模式”。
- 管道机制，即在同一个 TCP 连接里面，客户端可以同时发送多个请求。

HTTP/2.0

- 二进制协议，1.1 版本的头信息是文本（ASCII 编码），数据体可以是文本或者二进制；2.0 中，头信息和数据体都是二进制。
- 完全多路复用，在一个连接里，客户端和浏览器都可以同时发送多个请求或回应，而且不用按照顺序一一对应。
- 报头压缩，HTTP 协议不带有状态，每次请求都必须附上所有信息。Http/2.0 引入了头信息压缩机制，使用 gzip 或 compress 压缩后再发送。
- 服务端推送，允许服务器未经请求，主动向客户端发送资源。

16.HTTP/3了解吗？

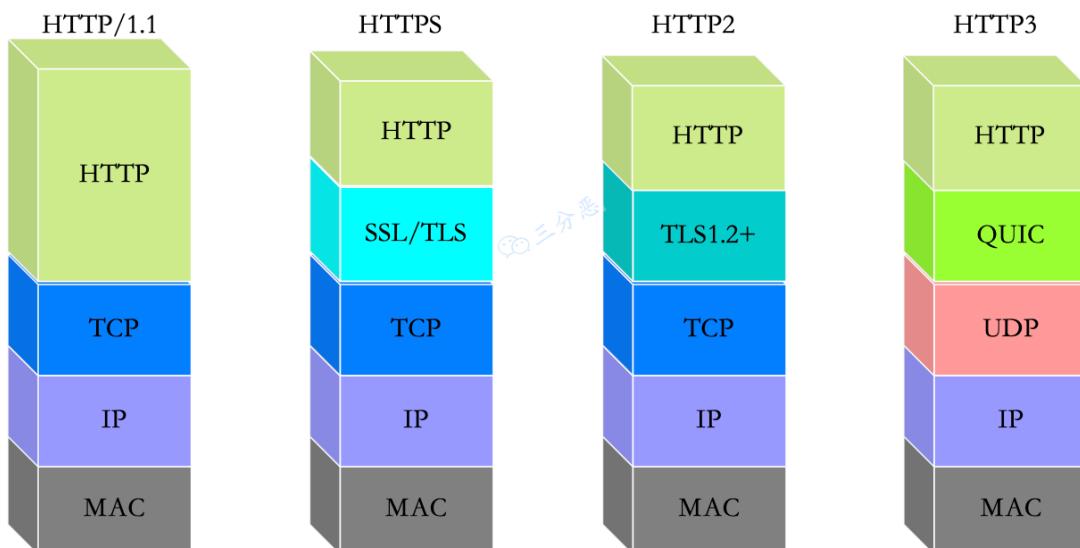
HTTP/3主要有两大变化，**传输层基于UDP**、使用**QUIC保证UDP可靠性**。

HTTP/2存在的一些问题，比如重传等等，都是由于TCP本身的特性导致的，所以HTTP/3在QUIC的基础上进行发展而来，QUIC（Quick UDP Connections）直译为快速UDP网络连接，底层使用UDP进行数据传输。

HTTP/3主要有这些特点：

- 使用UDP作为传输层进行通信
- 在UDP的基础上QUIC协议保证了HTTP/3的安全性，在传输的过程中就完成了TLS加密握手
- HTTPS要建立一个连接，要花费6次交互，先是建立三次握手，然后是TLS/1.3的三次握手。QUIC直接把以往的TCP和TLS/1.3的6次交互合并成了3次，减少了交互次数。
- QUIC有自己的一套机制可以保证传输的可靠性的。当某个流发生丢包时，只会阻塞这个流，其他流不会受到影响。

我们拿一张图看一下HTTP协议的变迁：



17. HTTP 如何实现长连接？在什么时候会超时？

什么是 HTTP 的长连接？

1. HTTP分为长连接和短连接，本质上说的是TCP的长短连接。TCP连接是一个双向的通道，它是可以保持一段时间不关闭的，因此TCP连接才具有真正的长连接和短连接这一说法。
2. TCP长连接可以复用一个TCP连接，来发起多次的HTTP请求，这样就可以减少资源消耗，比如一次请求HTML，如果是短连接的话，可能还需要请求后续的JS/CSS。

如何设置长连接？

通过在头部（请求和响应头）设置 **Connection** 字段指定为 **keep-alive**，HTTP/1.0 协议支持，但是是默认关闭的，从 HTTP/1.1 以后，连接默认都是长连接。

在什么时候会超时呢？

- HTTP 一般会有 httpd 守护进程，里面可以设置 keep-alive timeout，当 tcp 连接闲置超过这个时间就会关闭，也可以在 HTTP 的 header 里面设置超时时间
- TCP 的 keep-alive 包含三个参数，支持在系统内核的 net.ipv4 里面设置；当 TCP 连接之后，闲置了 `tcp_keepalive_time`，则会发生侦测包，如果没有收到对方的 ACK，那么会每隔 `tcp_keepalive_intvl` 再发一次，直到发送了 `tcp_keepalive_probes`，就会丢弃该连接。

```
1 | 1. tcp_keepalive_intvl = 15  
2 | 2. tcp_keepalive_probes = 5  
3 | 3. tcp_keepalive_time = 1800
```

18. 说说HTTP与HTTPS有哪些区别？

1. HTTP 是超文本传输协议，信息是明文传输，存在安全风险的问题。HTTPS 则解决 HTTP 不安全的缺陷，在TCP 和 HTTP 网络层之间加入了 SSL/TLS 安全协议，使得报文能够加密传输。
2. HTTP 连接建立相对简单，TCP 三次握手之后便可进行 HTTP 的报文传输。而 HTTPS 在 TCP 三次握手之后，还需进行 SSL/TLS 的握手过程，才可进入加密报文传输。
3. HTTP 的端口号是 80，HTTPS 的端口号是 443。
4. HTTPS 协议需要向 CA（证书权威机构）申请数字证书，来保证服务器的身份是可信的。

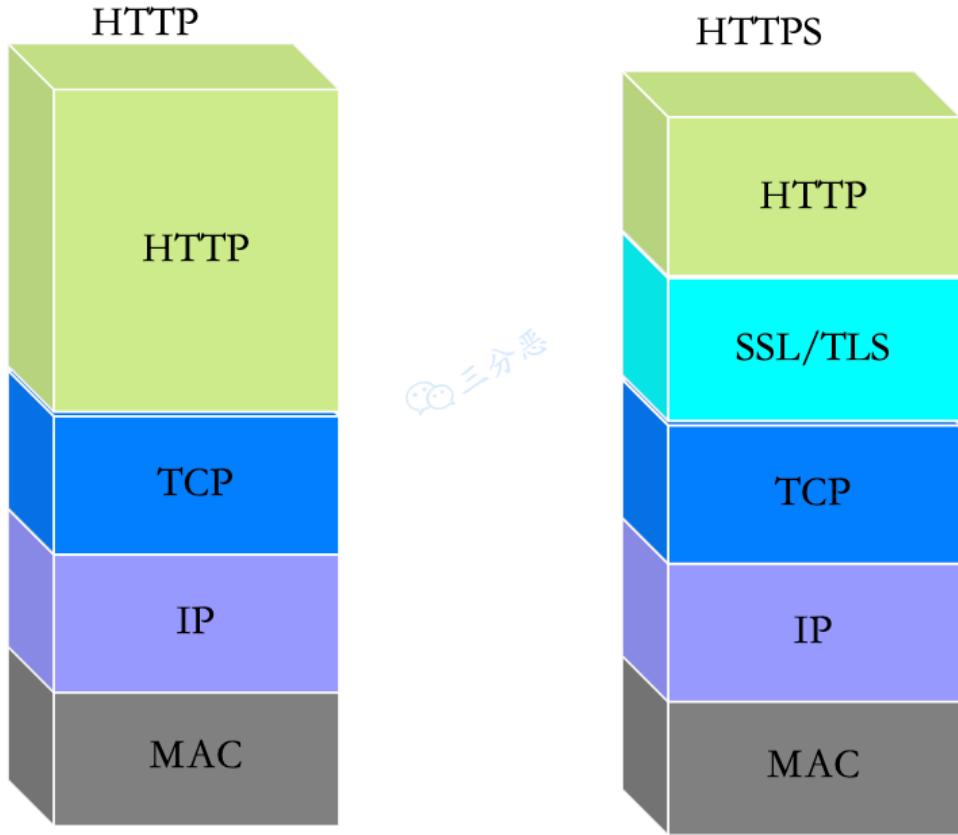
19. 为什么要用HTTPS？解决了哪些问题？

因为HTTP 是明文传输，存在安全上的风险：

窃听风险，比如通信链路上可以获取通信内容，用户账号被盗。

篡改风险，比如强制植入垃圾广告，视觉污染。

冒充风险，比如冒充淘宝网站，用户金钱损失。



所以引入了HTTPS，HTTPS 在 HTTP 与 TCP 层之间加入了 SSL/TLS 协议，可以很好的解决了这些风险：

- 信息加密：交互信息无法被窃取。
- 校验机制：无法篡改通信内容，篡改了就不能正常显示。
- 身份证书：能证明淘宝是真淘宝。

所以SSL/TLS 协议是能保证通信是安全的。

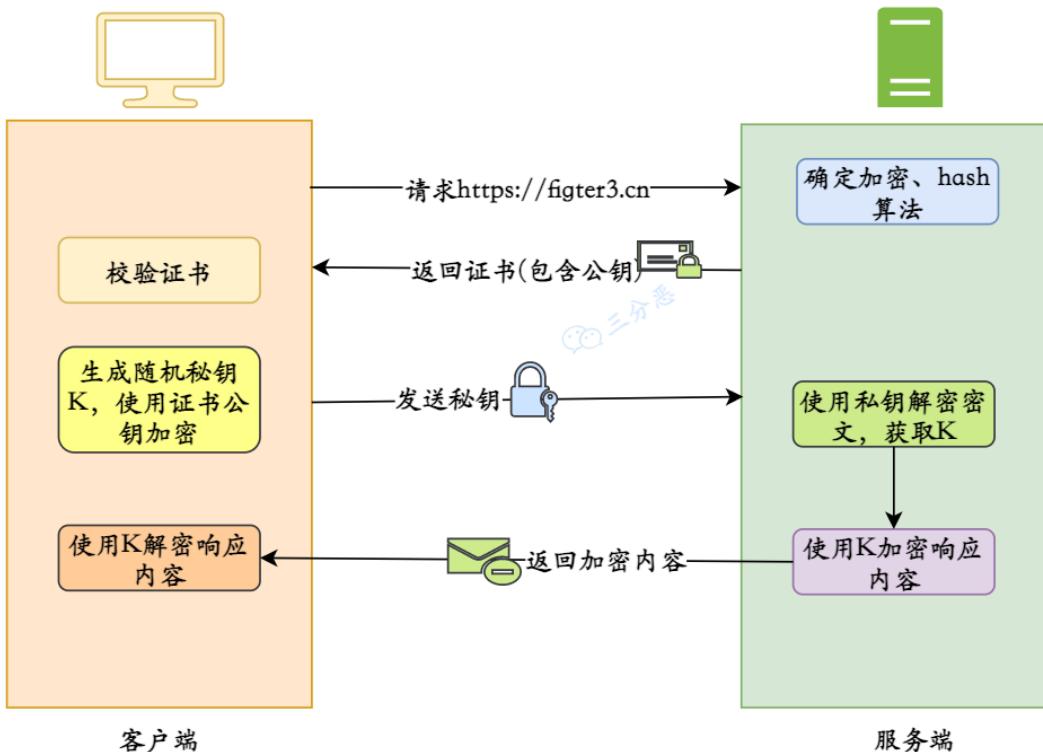
20. HTTPS工作流程是怎样的？

这道题有几个要点：**公私钥、数字证书、加密、对称加密、非对称加密**。

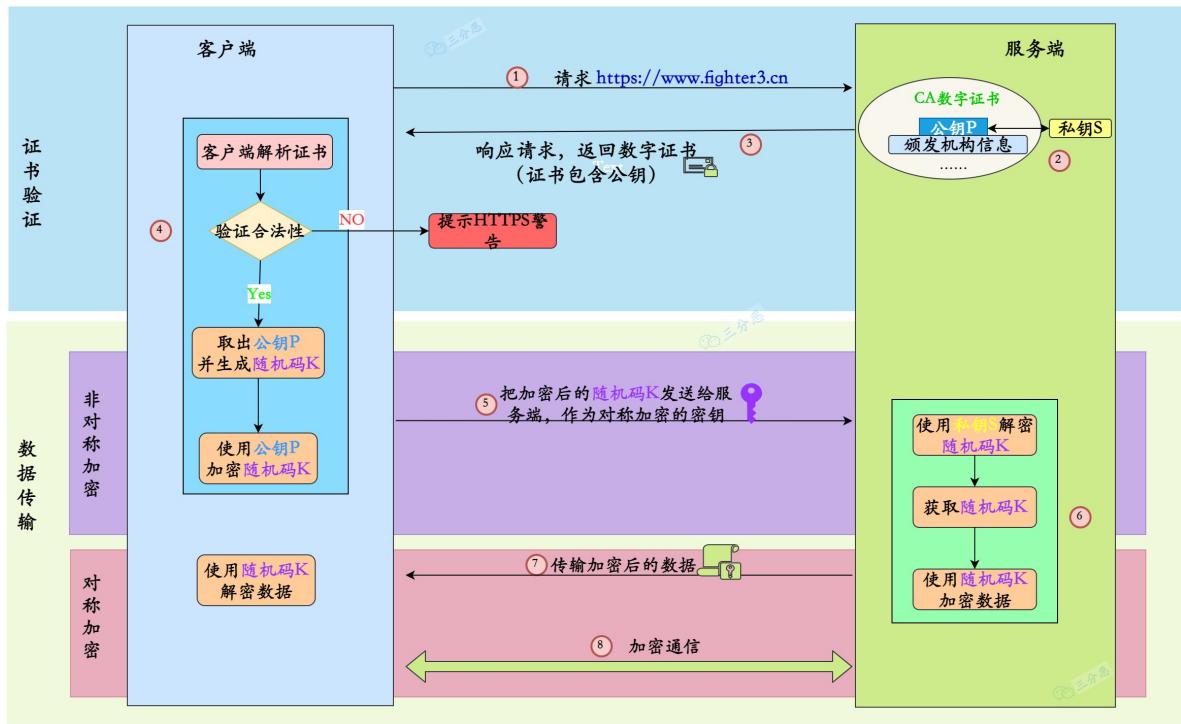
HTTPS 主要工作流程：

1. 客户端发起 HTTPS 请求，连接到服务端的 443 端口。
2. 服务端有一套数字证书（证书内容有公钥、证书颁发机构、失效日期等）。
3. 服务端将自己的数字证书发送给客户端（公钥在证书里面，私钥由服务器持有）。

4. 客户端收到数字证书之后，会验证证书的合法性。如果证书验证通过，就会生成一个随机的对称密钥，用证书的公钥加密。
5. 客户端将公钥加密后的密钥发送到服务器。
6. 服务器接收到客户端发来的密文密钥之后，用自己之前保留的私钥对其进行非对称解密，解密之后就得到客户端的密钥，然后用客户端密钥对返回数据进行对称加密，酱紫传输的数据都是密文啦。
7. 服务器将加密后的密文返回到客户端。
8. 客户端收到后，用自己的密钥对其进行对称解密，得到服务器返回的数据。



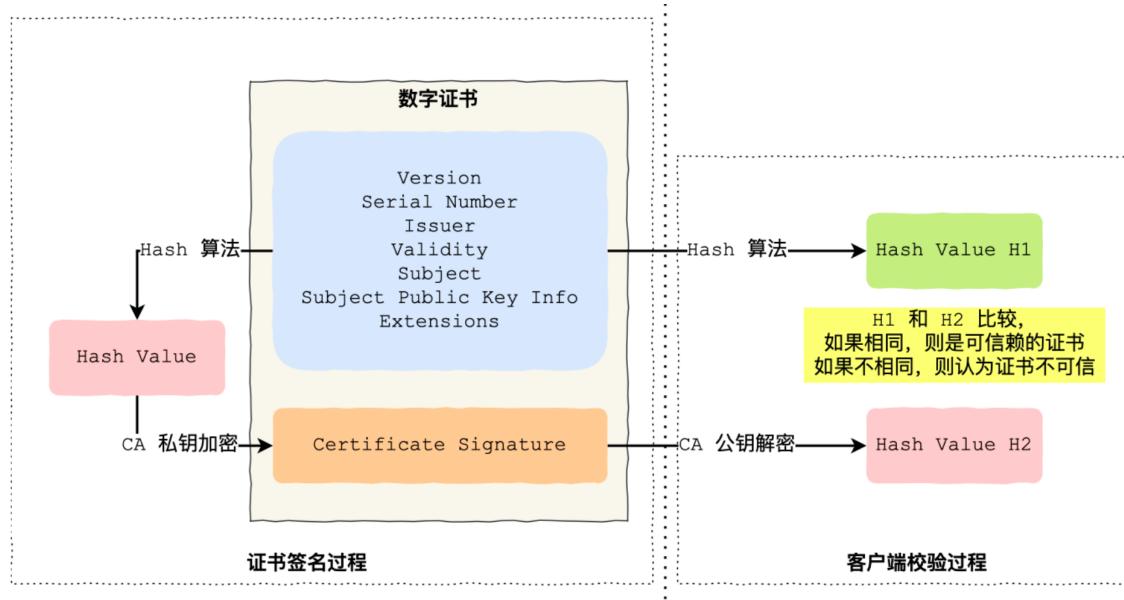
这里还画了一张更详尽的图：



21. 客户端怎么去校验证书的合法性？

首先，服务端的证书从哪来的呢？

为了让服务端的公钥被大家信任，服务端的证书都是由 CA（**Certificate Authority**，证书认证机构）签名的，CA就是网络世界里的公安局、公证中心，具有极高的可信度，所以由它来给各个公钥签名，信任的一方签发的证书，那必然证书也是被信任的。



CA 签发证书的过程，如上图左边部分：

- 首先 CA 会把持有者的公钥、用途、颁发者、有效时间等信息打成一个包，然后对这些信息进行 Hash 计算，得到一个 Hash 值；

- 然后 CA 会使用自己的私钥将该 Hash 值加密，生成 Certificate Signature，也就是 CA 对证书做了签名；
- 最后将 Certificate Signature 添加在文件证书上，形成数字证书；

客户端校验服务端的数字证书的过程，如上图右边部分：

- 首先客户端会使用同样的 Hash 算法获取该证书的 Hash 值 H1；
- 通常浏览器和操作系统中集成了 CA 的公钥信息，浏览器收到证书后可以使用 CA 的公钥解密 Certificate
- Signature 内容，得到一个 Hash 值 H2；
- 最后比较 H1 和 H2，如果值相同，则为可信赖的证书，否则则认为证书不可信。

假如在HTTPS的通信过程中，中间人篡改了证书原文，由于他没有CA机构的私钥，所以CA公钥解密的内容就不一致。

22.如何理解 HTTP 协议是无状态的？

这个 **无状态** 的 **状态** 值的是什么？是客户端的状态，所以字面意思，就是HTTP协议中服务端不会保存客户端的任何信息。

比如当浏览器第一次发送请求给服务器时，服务器响应了；如果同个浏览器发起第二次请求给服务器时，它还是会响应，但是呢，服务器不知道你就是刚才的那个浏览器。

那有什么办法记录状态呢？

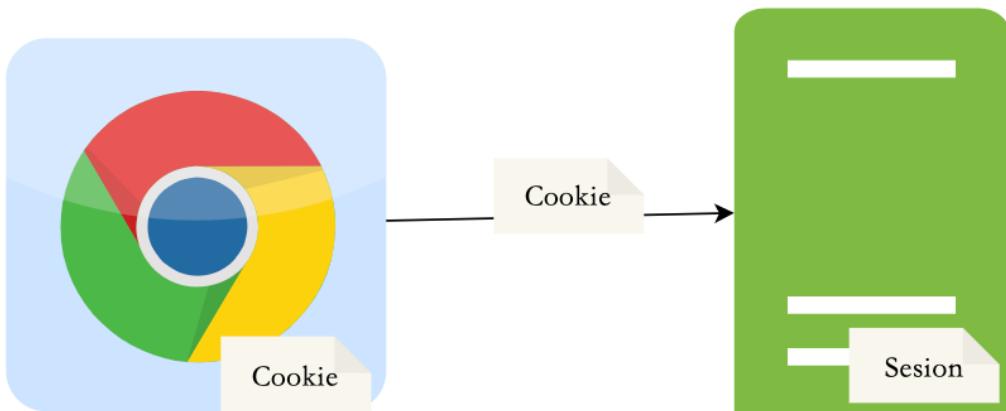
主要有两个办法，Session和Cookie。

23.说说Session 和 Cookie 有什么联系和区别？

先来看看什么是 Session 和 Cookie：

- Cookie 是保存在客户端的一小块文本串的数据。客户端向服务器发起请求时，服务端会向客户端发送一个 Cookie，客户端就把 Cookie 保存起来。在客户端下次向同一服务器再发起请求时，Cookie 被携带发送到服务器。服务端可以根据这个Cookie判断用户的身份和状态。
- Session 指的就是服务器和客户端一次会话的过程。它是另一种记录客户状态的机制。不同的是cookie保存在客户端浏览器中，而session保存在服务器上。客户

端浏览器访问服务器的时候，服务器把客户端信息以某种形式记录在服务器上，这就是session。客户端浏览器再次访问时只需要从该session中查找用户的状态。

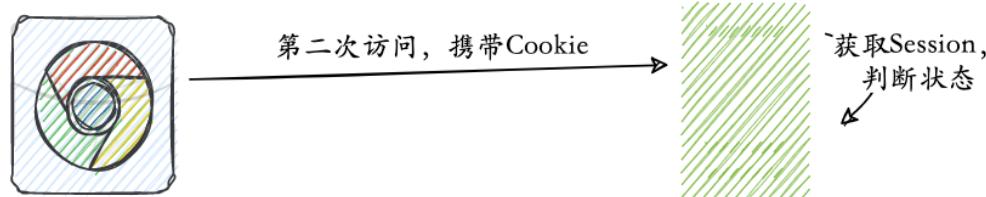
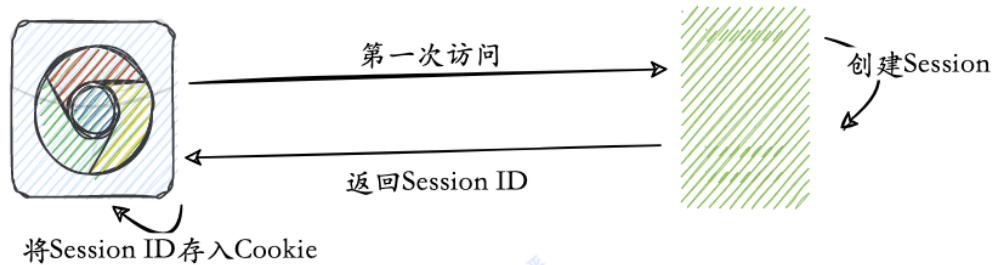


Session 和 Cookie 到底有什么不同呢？

- 存储位置不一样，Cookie 保存在客户端，Session 保存在服务器端。
- 存储数据类型不一样，Cookie 只能保存ASCII，Session可以存任意数据类型，一般情况下我们可以在 Session 中保持一些常用变量信息，比如说 UserId 等。
- 有效期不同，Cookie 可设置为长时间保持，比如我们经常使用的默认登录功能，Session 一般有效时间较短，客户端关闭或者 Session 超时都会失效。
- 隐私策略不同，Cookie 存储在客户端，比较容易遭到不法获取，早期有人将用户的登录名和密码存储在 Cookie 中导致信息被窃取；Session 存储在服务端，安全性相对 Cookie 要好一些。
- 存储大小不同，单个Cookie保存的数据不能超过4K，Session可存储数据远高于Cookie。

Session 和 Cookie有什么关联呢？

可以使用Cookie记录Session的标识。



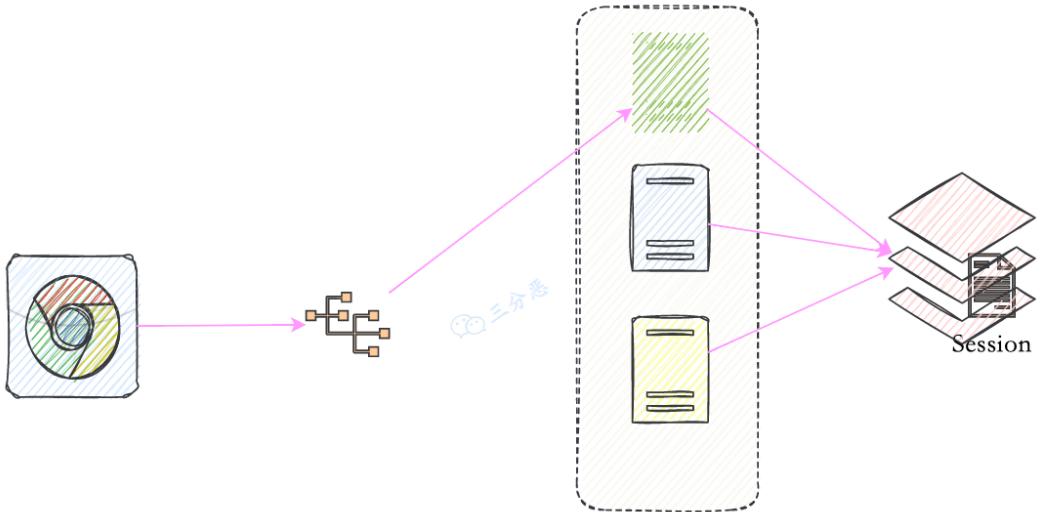
- 用户第一次请求服务器时，服务器根据用户提交的信息，创建对应的 Session，请求返回时将此 Session 的唯一标识信息 SessionID 返回给浏览器，浏览器接收到服务器返回的 SessionID 信息后，会将此信息存入 Cookie 中，同时 Cookie 记录此 SessionID 是属于哪个域名。
- 当用户第二次访问服务器时，请求会自动判断此域名下是否存在 Cookie 信息，如果存在，则自动将 Cookie 信息也发送给服务端，服务端会从 Cookie 中获取 SessionID，再根据 SessionID 查找对应的 Session 信息，如果没有找到，说明用户没有登录或者登录失效，如果找到 Session 证明用户已经登录可执行后面操作。

分布式环境下Session怎么处理呢？

分布式环境下，客户端请求经过负载均衡，可能会分配到不同的服务器上，假如一个用户的请求两次没有落到同一台服务器上，那么在新的服务器上就没有记录用户状态的Session。

这时候怎么办呢？

可以使用Redis等分布式缓存来存储Session，在多台服务器之间共享。



客户端无法使用Cookie怎么办？

有可能客户端无法使用Cookie，比如浏览器禁用Cookie，或者客户端是安卓、IOS等等。

这时候怎么办？SessionID怎么存？怎么传给服务端呢？

首先是SessionID的存储，可以使用客户端的本地存储，比如浏览器的sessionStorage。

接下来怎么传呢？

- 拼接到URL里：直接把SessionID作为URL的请求参数
- 放到请求头里：把SessionID放到请求的Header里，比较常用。

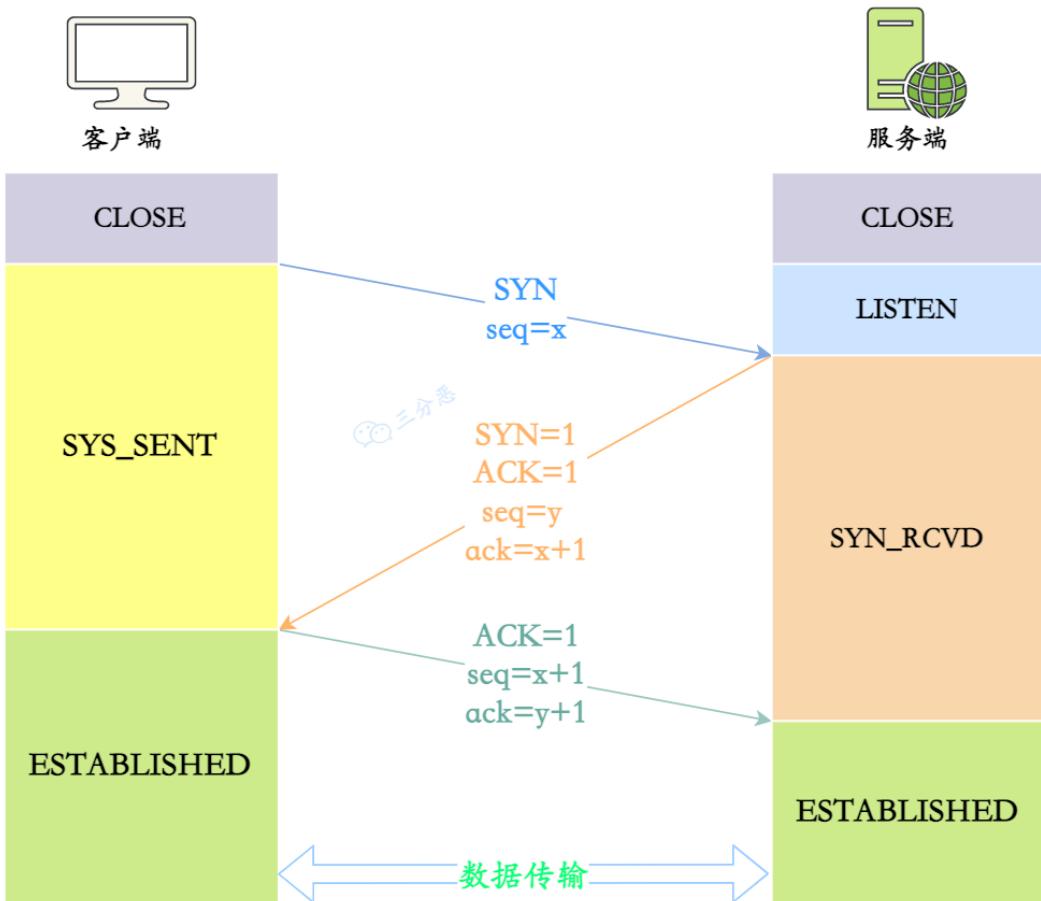
TCP

24. 详细说一下 TCP 的三次握手机制

PS:TCP三次握手是最重要的知识点，一定要熟悉到问到即送分。

TCP提供面向连接的服务，在传送数据前必须建立连接，TCP连接是通过三次握手建立的。

三次握手



三次握手的过程：

- 最开始，客户端和服务端都处于CLOSE状态，服务端监听客户端的请求，进入LISTEN状态
- 客户端发送连接请求，第一次握手 ($SYN=1, seq=x$)，发送完毕后，客户端就进入 **SYN_SENT** 状态
- 服务端确认连接，第二次握手 ($SYN=1, ACK=1, seq=y, ACKnum=x+1$)，发送完毕后，服务器端就进入 **SYN_RCV** 状态。
- 客户端收到服务端的确认之后，再次向服务端确认，这就是第三次握手 ($ACK=1, ACKnum=y+1$)，发送完毕后，客户端进入 **ESTABLISHED** 状态，当服务器端接收到这个包时，也进入 **ESTABLISHED** 状态。

TCP三次握手通俗比喻：

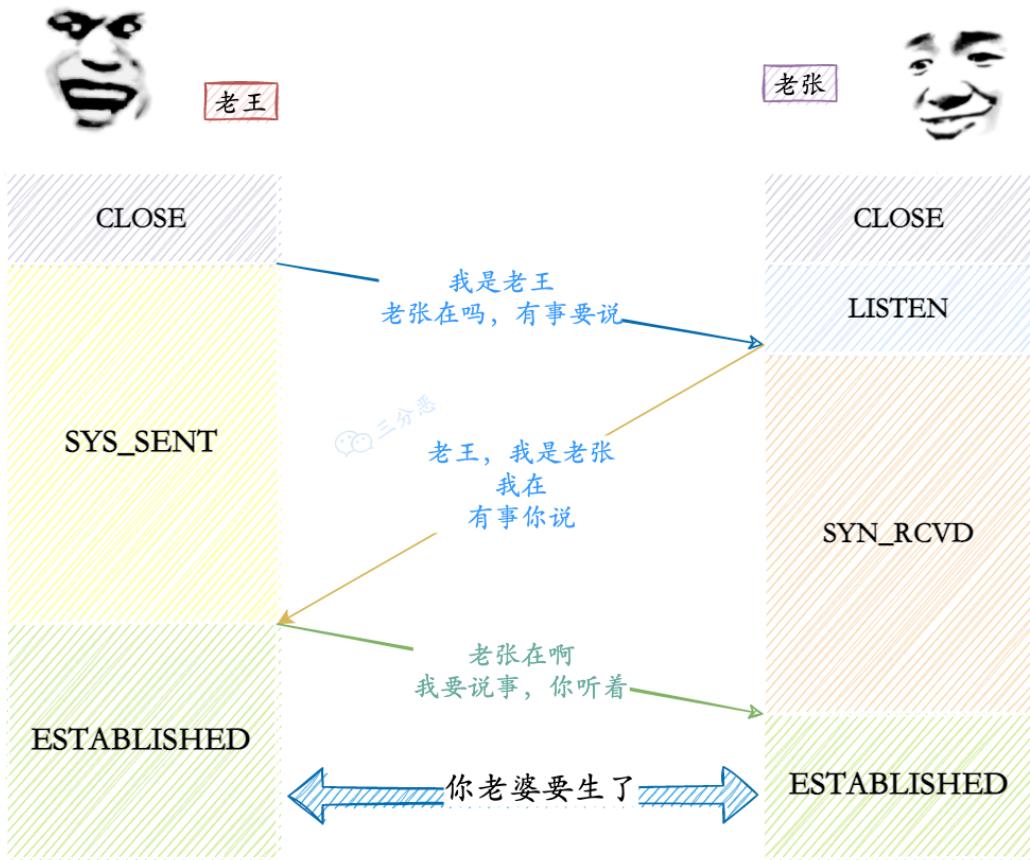
在二十年前的农村，电话没有普及，手机就更不用说了，所以，通信基本靠吼。

老张和老王是邻居，这天老张下地了，结果家里有事，热心的邻居老王赶紧跑到村口，开始叫唤老王。

- 老王：老张唉！我是老王，你能听到吗？

- 老张一听，是老王的声音：老王老王，我是老张，我能听到，你能听到吗？
 - 老王一听，嗯，没错，是老张：老张，我听到了，我有事要跟你说。
- "你老婆要生了，赶紧回家吧！"

老张风风火火地赶回家，老婆顺利地生了个带把的大胖小子。握手的故事充满了幸福和美满。



25.TCP 握手为什么是三次，为什么不能是两次？不能是四次？

为什么不能是两次？

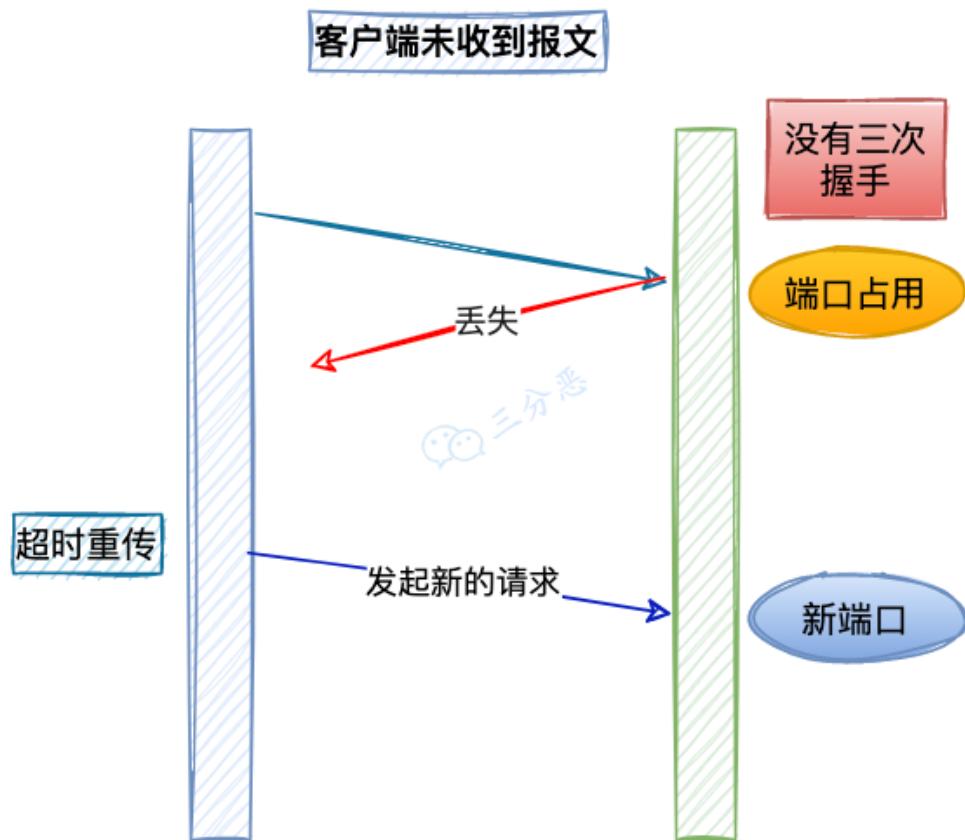
- 为了防止服务器端开启一些无用的连接增加服务器开销
- 防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误。

由于网络传输是有延时的(要通过网络光纤和各种中间代理服务器)，在传输的过程中，比如客户端发起了 $SYN=1$ 的第一次握手。

如果服务器端就直接创建了这个连接并返回包含 SYN 、 ACK 和 Seq 等内容的数据包给客户端，这个数据包因为网络传输的原因丢失了，丢失之后客户端就一直没有接收到服务器返回的数据包。

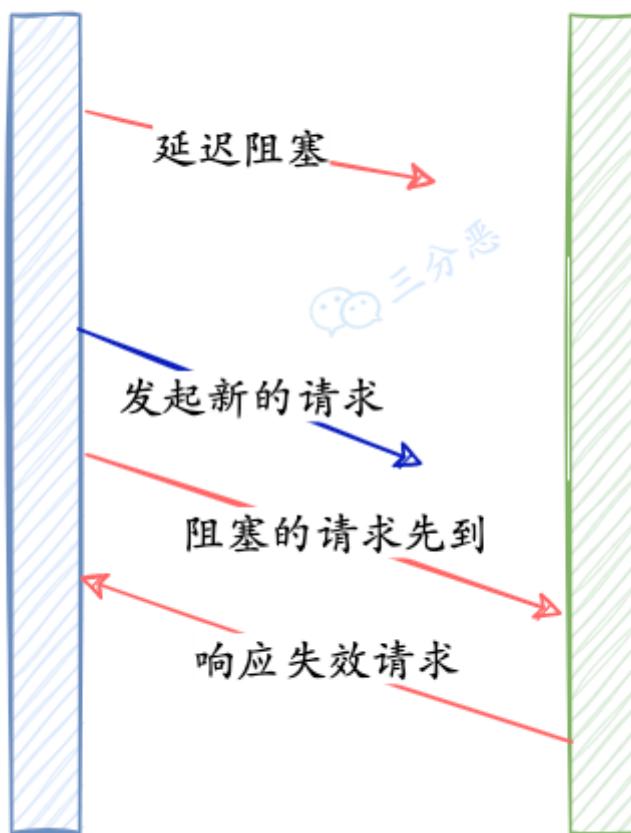
如果没有第三次握手告诉服务器端客户端收的到服务器端传输的数据的话，服务器端是不知道客户端有没有接收到服务器端返回的信息的。

服务端就认为这个连接是可用的，端口就一直开着，等到客户端因超时重新发出请求时，服务器就会重新开启一个端口连接。这样一来，就会有很多无效的连接端口白白地开着，导致资源的浪费。



还有一种情况是已经失效的客户端发出的请求信息，由于某种原因传输到了服务器端，服务器端以为是客户端发出的有效请求，接收后产生错误。

响应失效请求



所以我们需要“第三次握手”来确认这个过程：

通过第三次握手的数据告诉服务端，客户端有没有收到服务器“第二次握手”时传过去的数据，以及这个连接的序号是不是有效的。若发送的这个数据是“**收到且没有问题**”的信息，接收后服务器就正常建立 TCP 连接，否则建立 TCP 连接失败，服务器关闭连接端口。由此减少服务器开销和接收到失效请求发生的错误。

为什么不是四次？

简单说，就是三次挥手已经足够创建可靠的连接，没有必要再多一次握手导致花费更多的时间建立连接。

26.三次握手中每一次没收到报文会发生什么情况？

- 第一次握手服务端未收到SYN报文

服务端不会进行任何的动作，而客户端由于一段时间内没有收到服务端发来的确认报文，等待一段时间后会重新发送SYN报文，如果仍然没有回应，会重复这个过程，直到发送次数超过最大重传次数限制，就会返回连接建立失败。

- 第二次握手客户端未收到服务端响应的ACK报文

客户端会继续重传，直到次数限制；而服务端此时会阻塞在accept()处，等待客户端发送ACK报文

- 第三次握手服务端为收到客户端发送过来的ACK报文

服务端同样会采用类似客户端的超时重传机制，如果重试次数超过限制，则accept()调用返回-1，服务端建立连接失败；而此时客户端认为自己已经建立连接成功，因此开始向服务端发送数据，但是服务端的accept()系统调用已经返回，此时不在监听状态，因此服务端接收到客户端发送来的数据时会发送RST报文给客户端，消除客户端单方面建立连接的状态。

27.第二次握手传回了 ACK，为什么还要传回 SYN？

ACK是为了告诉客户端传来的数据已经接收无误。

而传回SYN是为了告诉客户端，服务端响应的确实是客户端发送的报文。

28.第3次握手可以携带数据吗？

第3次握手是可以携带数据的。

此时客户端已经处于ESTABLISHED状态。对于客户端来说，它已经建立连接成功，并且确认服务端的接收和发送能力是正常的。

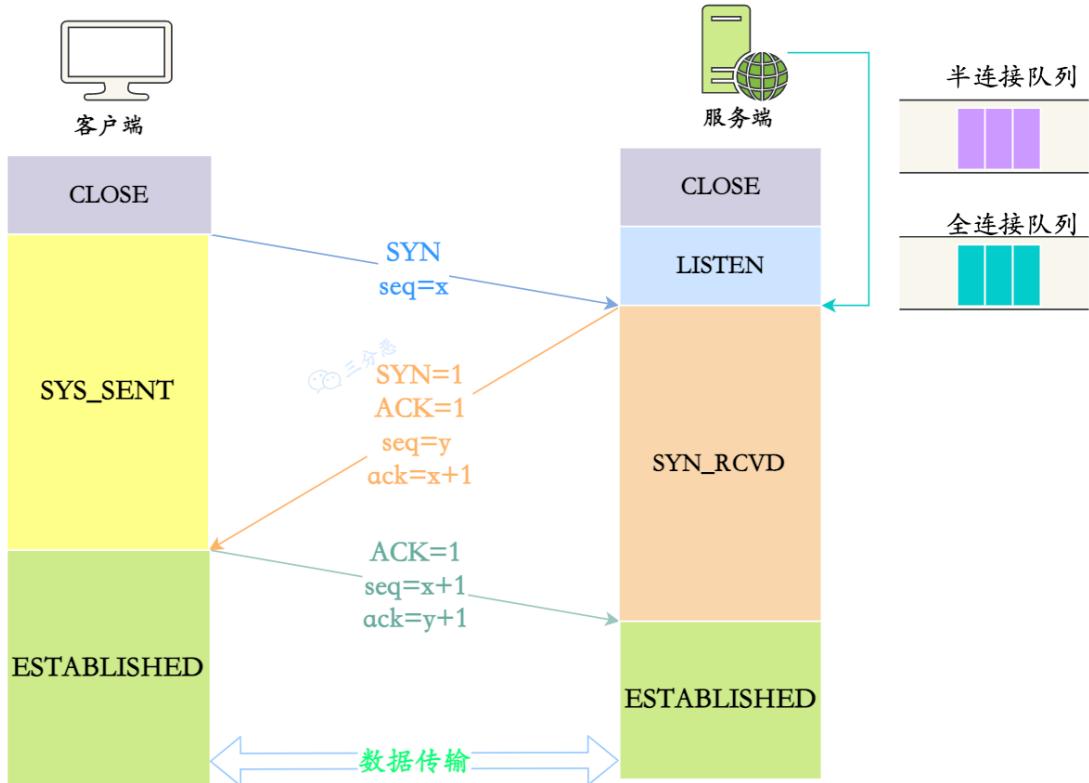
第一次握手不能携带数据是出于安全的考虑，因为如果允许携带数据，攻击者每次在SYN报文中携带大量数据，就会导致服务端消耗更多的时间和空间去处理这些报文，会造成CPU和内存的消耗。

29.说说半连接队列和 SYN Flood 攻击的关系？

什么是半连接队列？

TCP 进入三次握手前，服务端会从 **CLOSED** 状态变为 **LISTEN** 状态，同时在内部创建了两个队列：半连接队列（SYN 队列）和全连接队列（ACCEPT 队列）。

三次握手

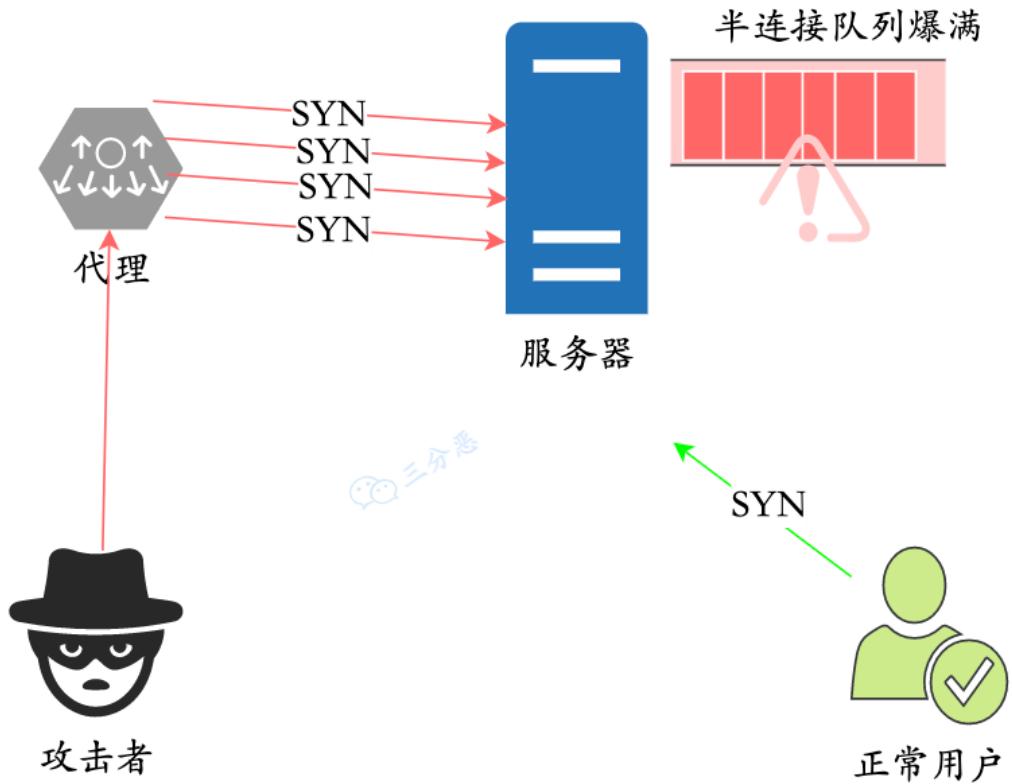


顾名思义，半连接队列存放的是三次握手未完成的连接，全连接队列存放的是完成三次握手的连接。

- TCP 三次握手时，客户端发送 SYN 到服务端，服务端收到之后，便回复 ACK 和 SYN，状态由 LISTEN 变为 SYN_RECV，此时这个连接就被推入了 SYN 队列，即半连接队列。
- 当客户端回复 ACK，服务端接收后，三次握手就完成了。这时连接会等待被具体的应用取走，在被取走之前，它被推入 ACCEPT 队列，即全连接队列。

什么是SYN Flood ?

SYN Flood 是一种典型的 DDos 攻击，它在短时间内，伪造 **不存在的 IP 地址**，向服务器发送大量SYN 报文。当服务器回复 SYN+ACK 报文后，不会收到 ACK 回应报文，那么SYN队列里的连接旧不会出对队，久而久之就会占满服务端的 **SYN** 接收队列（半连接队列），使得服务器不能为正常用户提供服务。



那有什么应对方案呢？

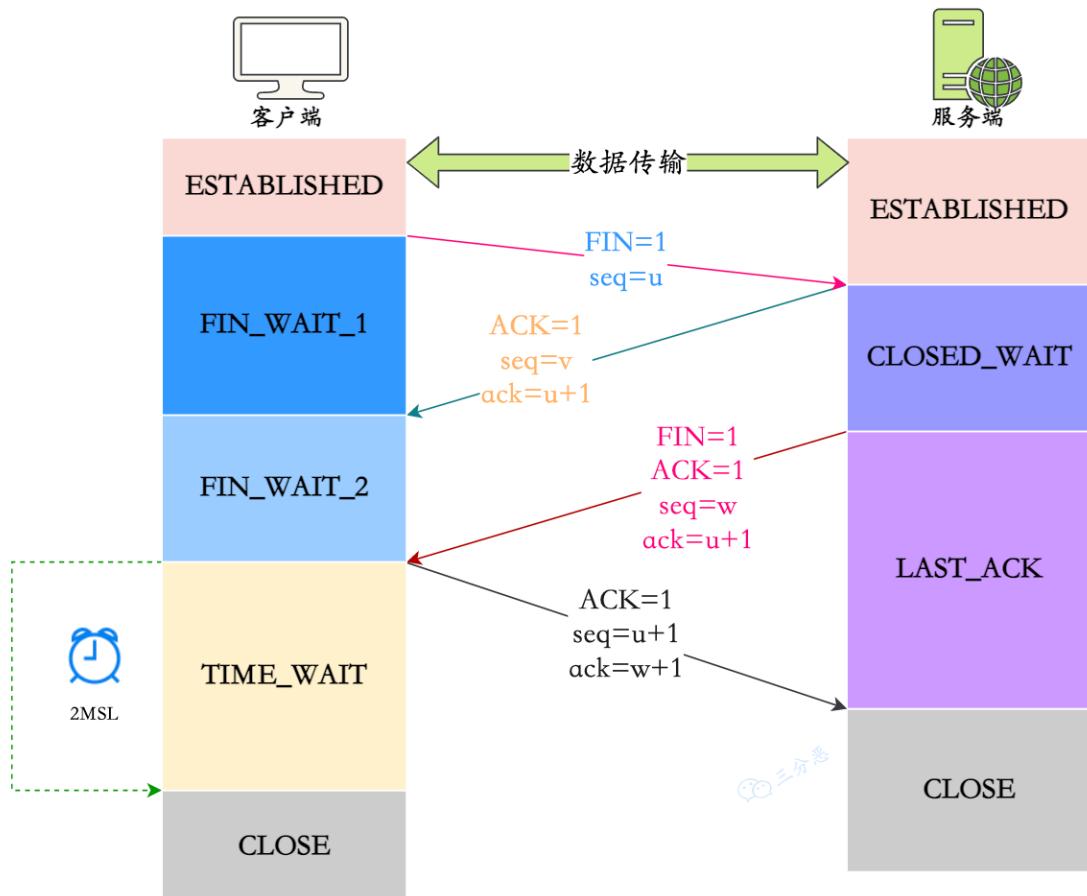
主要有 **syn cookie** 和 **SYN Proxy 防火墙** 等。

- **syn cookie**：在收到 SYN 包后，服务器根据一定的方法，以数据包的源地址、端口等信息为参数计算出一个 cookie 值作为自己的 SYNACK 包的序列号，回复 SYN+ACK 后，服务器并不立即分配资源进行处理，等收到发送方的 ACK 包后，重新根据数据包的源地址、端口计算该包中的确认序列号是否正确，如果正确则建立连接，否则丢弃该包。
- **SYN Proxy 防火墙**：服务器防火墙会对收到的每一个 SYN 报文进行代理和回应，并保持半连接。等发送方将 ACK 包返回后，再重新构造 SYN 包发到服务器，建立真正的 TCP 连接。

30. 说说 TCP 四次挥手的过程？

PS：问完三次握手，常常也会顺道问问四次挥手，所以也是必须掌握知识点。

四次挥手



TCP 四次挥手过程:

- 数据传输结束之后，通信双方都可以主动发起断开连接请求，这里假定客户端发起
- 客户端发送释放连接报文，第一次挥手 ($\text{FIN}=1, \text{seq}=u$)，发送完毕后，客户端进入 `FIN_WAIT_1` 状态。
- 服务端发送确认报文，第二次挥手 ($\text{ACK}=1, \text{ack}=u+1, \text{seq}=v$)，发送完毕后，服务器端进入 `CLOSE_WAIT` 状态，客户端接收到这个确认包之后，进入 `FIN_WAIT_2` 状态。
- 服务端发送释放连接报文，第三次挥手 ($\text{FIN}=1, \text{ACK}=1, \text{seq}=w, \text{ack}=u+1$)，发送完毕后，服务器端进入 `LAST_ACK` 状态，等待来自客户端的最后一个 ACK。
- 客户端发送确认报文，第四次挥手 ($\text{ACK}=1, \text{seq}=u+1, \text{ack}=w+1$)，客户端接收到来自服务器端的关闭请求，发送一个确认包，并进入 `TIME_WAIT` 状态，等待了某个固定时间（两个最大段生命周期， 2MSL ， 2 Maximum Segment Lifetime）之后，没有收到服务器端的 ACK，认为服务器端已经正常关闭连接，于是自己也关闭连接，进入 `CLOSE` 状态。服务器端接收到这个确认包之后，关闭连接，进入 `CLOSE` 状态。

大白话说四次挥手：

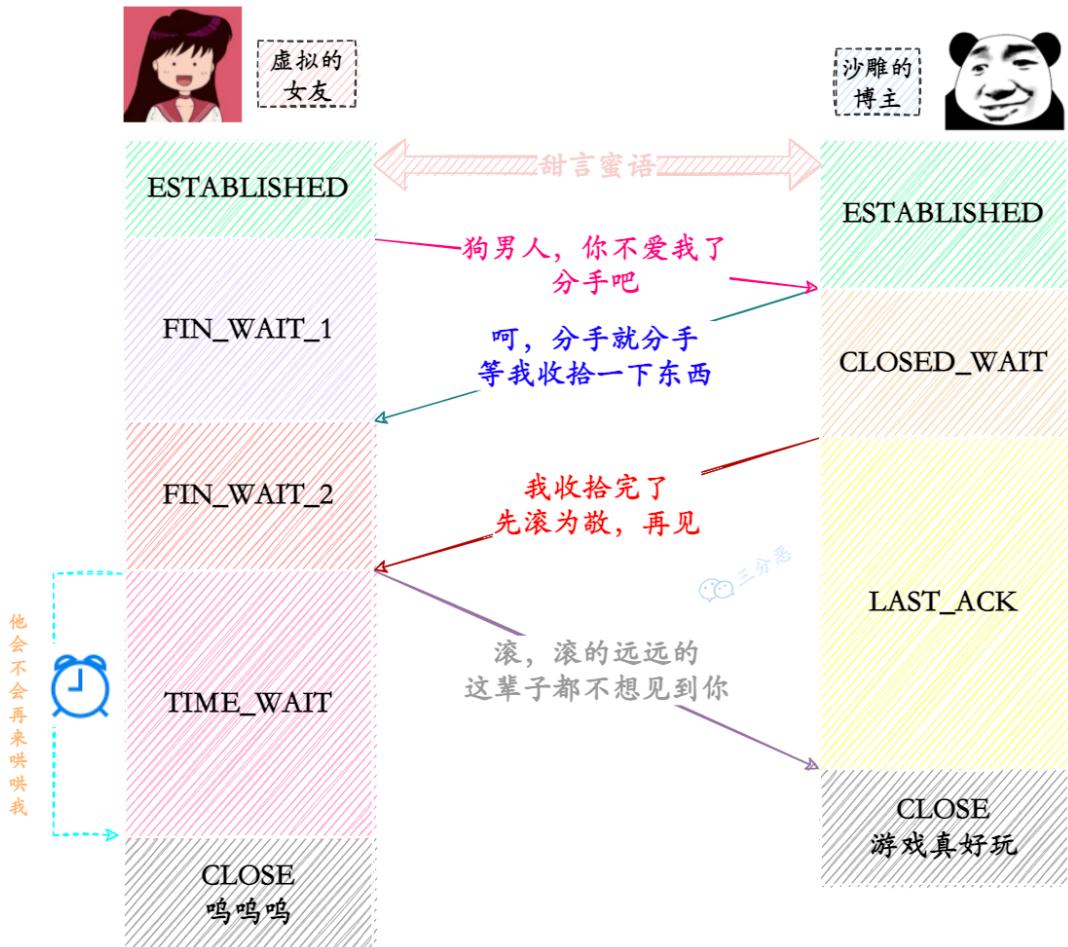
假如单身狗博主有一个女朋友—由于博主上班九九六，下班肝博客，导致没有时间陪女朋友，女朋友忍无可忍。

- 女朋友：狗男人，最近你都不理我，你是不是不爱我了？你是不是外面有别的狗子了？我要和你分手？
- 沙雕博主一愣，怒火攻心：分手就分手，不陪你闹了，等我把东西收拾收拾。

沙雕博主小心翼翼地装起了自己的青轴机械键盘。

- 哼，蠢女人，我已经收拾完了，我先滚为敬，再见！
- 女朋友：滚，滚的远远的，越远越好，我一辈子都不想再见到你。

挥手的故事总充满了悲伤和遗憾！



31.TCP 挥手为什么需要四次呢？

再来看看四次挥手双方发 **FIN** 包的过程，就能理解为什么需要四次了。

- 关闭连接时，客户端向服务端发送 **FIN** 时，仅仅表示客户端不再发送数据了但是还能接收数据。

- 服务端收到客户端的 **FIN** 报文时，先回一个 **ACK** 应答报文，而服务端可能还有数据需要处理和发送，等服务端不再发送数据时，才发送 **FIN** 报文给客户端来表示同意现在关闭连接。

从上面过程可知，服务端通常需要等待完成数据的发送和处理，所以服务端的 **ACK** 和 **FIN** 一般都会分开发送，从而比三次握手导致多了一次。

32.TCP 四次挥手过程中，为什么需要等待 **2MSL**，才进入 **CLOSED** 关闭状态？

为什么需要等待？

1. 为了保证客户端发送的最后一个 **ACK** 报文段能够到达服务端。这个 **ACK** 报文段有可能丢失，因而使处在 **LAST-ACK** 状态的服务端就收不到对已发送的 **FIN + ACK** 报文段的确认。服务端会超时重传这个 **FIN+ACK** 报文段，而客户端就能在 **2MSL** 时间内（**超时 + 1MSL 传输**）收到这个重传的 **FIN+ACK** 报文段。接着客户端重传一次确认，重新启动 **2MSL** 计时器。最后，客户端和服务器都正常进入到 **CLOSED** 状态。

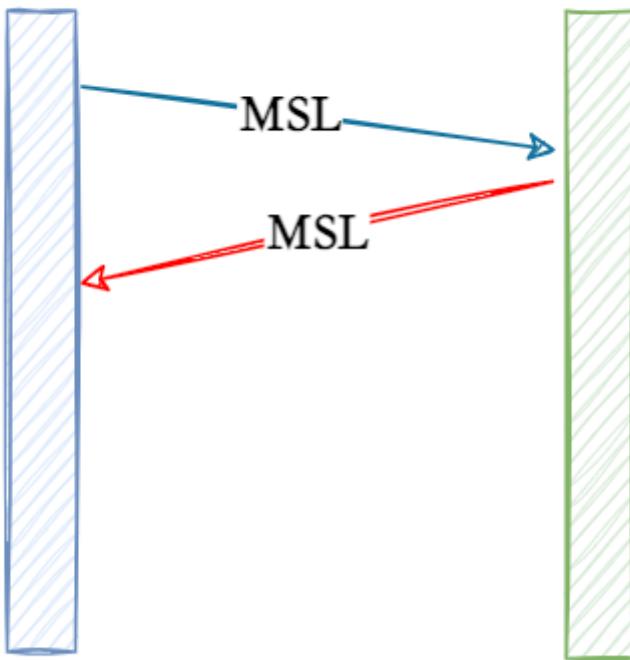
2. 防止已失效的连接请求报文段出现在本连接中。客户端在发送完最后一个 **ACK** 报文段后，再经过时间 **2MSL**，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失。这样就可以使下一个连接中不会出现这种旧的连接请求报文段。

为什么等待的时间是**2MSL**？

MSL 是 Maximum Segment Lifetime，报文最大生存时间，它是任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃。

TIME_WAIT 等待 2 倍的 **MSL**，比较合理的解释是：网络中可能存在来自发送方的数据包，当这些发送方的数据包被接收方处理后又会向对方发送响应，所以一来一回需要等待 **2** 倍的时间。

一个来回



比如如果被动关闭方没有收到断开连接的最后的 ACK 报文，就会触发超时重发 Fin 报文，另一方接收到 FIN 后，会重发 ACK 给被动关闭方，一来一去正好 2 个 MSL。

33.保活计时器有什么用？

除时间等待计时器外，TCP 还有一个保活计时器（keepalive timer）。

设想这样的场景：客户已主动与服务器建立了 TCP 连接。但后来客户端的主机突然发生故障。显然，服务器以后就不能再收到客户端发来的数据。因此，应当有措施使服务器不要再白白等待下去。这就需要使用保活计时器了。

服务器每收到一次客户端的数据，就重新设置保活计时器，时间的设置通常是两个小时。若两个小时都没有收到客户端的数据，服务端就发送一个探测报文段，以后则每隔 75 秒钟发送一次。若连续发送 10 个探测报文段后仍然无客户端的响应，服务端就认为客户端出了故障，接着就关闭这个连接。

34.CLOSE-WAIT 和 TIME-WAIT 的状态和意义？

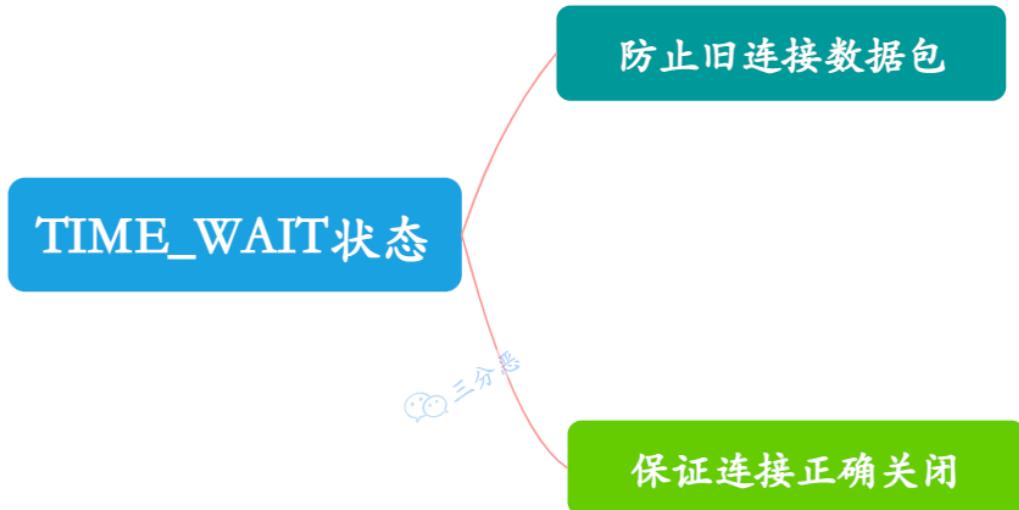
CLOSE-WAIT 状态有什么意义？

服务端收到客户端关闭连接的请求并确认之后，就会进入CLOSE-WAIT状态。此时服务端可能还有一些数据没有传输完成，因此不能立即关闭连接，而CLOSE-WAIT状态就是为了保证服务端在关闭连接之前将待发送的数据处理完。

TIME-WAIT有什么意义？

TIME-WAIT状态发生在第四次挥手，当客户端向服务端发送ACK确认报文后进入TIME-WAIT状态。

它存在的意义主要是两个：



- 防止旧连接的数据包

如果客户端收到服务端的FIN报文之后立即关闭连接，但是此时服务端对应的端口并没有关闭，如果客户端在相同端口建立新的连接，可能会导致新连接收到旧连接残留的数据包，导致不可预料的异常发生。

- 保证连接正确关闭

假设客户端最后一次发送的ACK包在传输的时候丢失了，由于TCP协议的超时重传机制，服务端将重发FIN报文，如果客户端没有维持TIME-WAIT状态而直接关闭的话，当收到服务端重新发送的FIN包时，客户端就会使用RST包来响应服务端，导致服务端以为有错误发生，然而实际关闭连接过程是正常的。

35.TIME_WAIT 状态过多会导致什么问题？怎么解决？

TIME_WAIT 状态过多会导致什么问题？

如果服务器有处于 TIME-WAIT 状态的 TCP，则说明是由服务器方主动发起的断开请求。

过多的 TIME-WAIT 状态主要的危害有两种：

第一是内存资源占用；

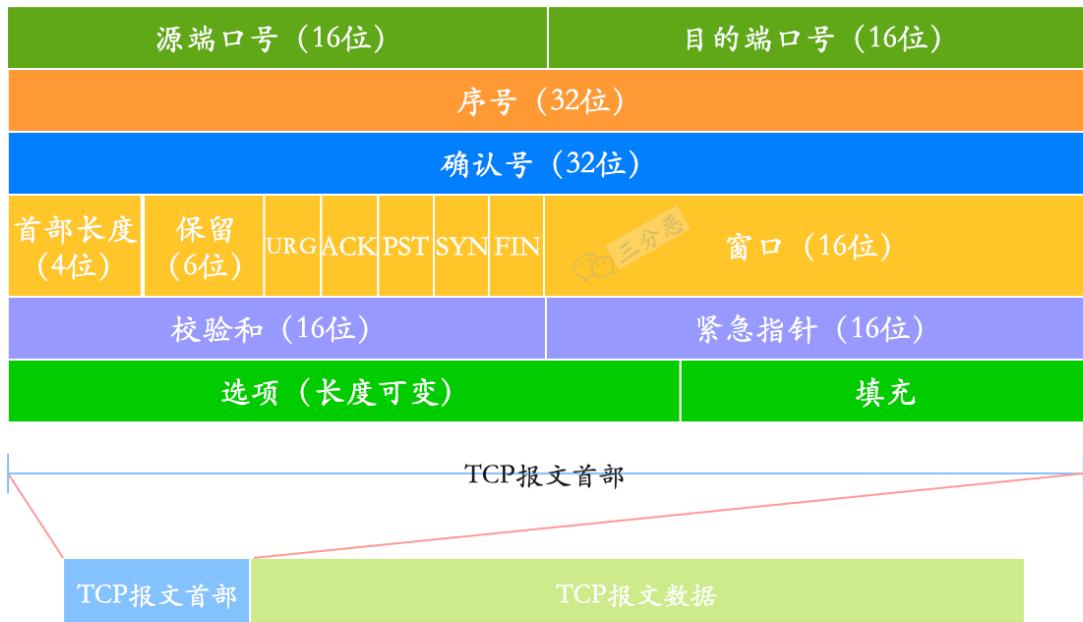
第二是对端口资源的占用，一个 TCP 连接至少消耗一个本地端口；

怎么解决TIME_WAIT 状态过多？

- 服务器可以设置SO_REUSEADDR套接字来通知内核，如果端口被占用，但是TCP连接位于TIME_WAIT 状态时可以重用端口。
- 还可以使用长连接的方式来减少TCP的连接和断开，在长连接的业务里往往不需要考虑TIME_WAIT状态。

36.说说 TCP 报文头部的格式？

看一下TCP报文头部的格式：

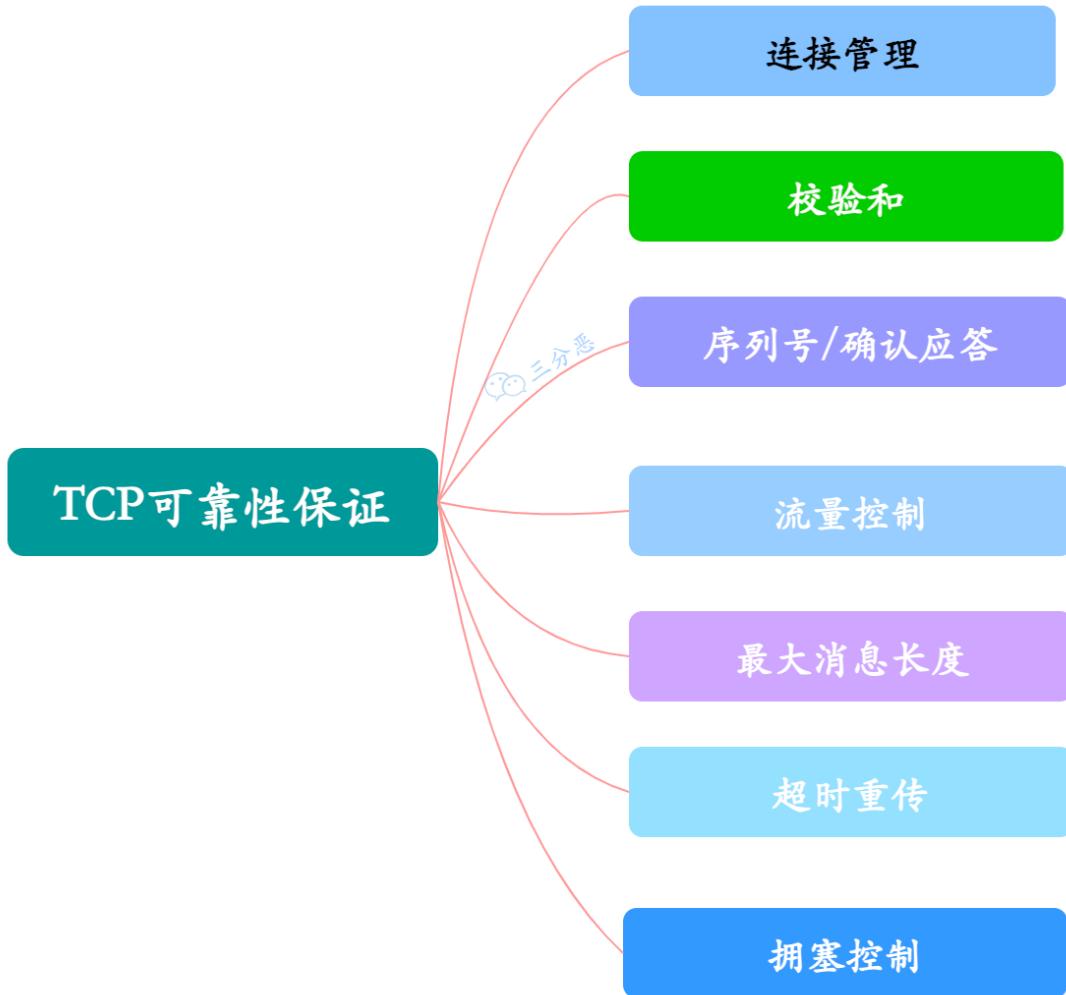


- 16 位端口号：源端口号，主机该报文段是来自哪里；目标端口号，要传给哪个上层协议或应用程序
- 32 位序号：一次 TCP 通信（从 TCP 连接建立到断开）过程中某一个传输方向上的字节流的每个字节的编号。
- 32 位确认号：用作对另一方发送的 tcp 报文段的响应。其值是收到的 TCP 报文段的序号值加 1。

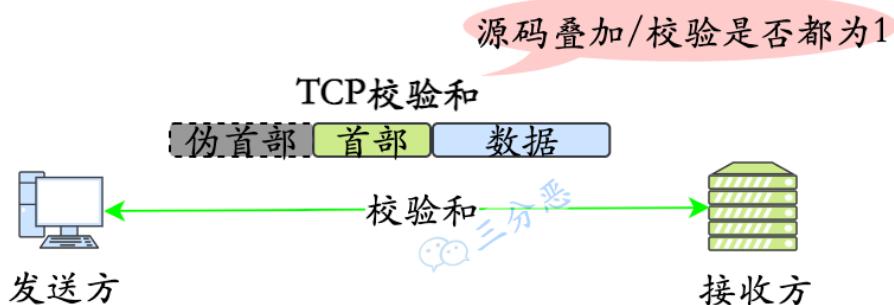
- 4 位首部长度：表示 tcp 头部有多少个 32bit 字（4 字节）。因为 4 位最大能标识 15，所以 TCP 头部最长是 60 字节。
- 6 位标志位：URG(紧急指针是否有效), ACK (表示确认号是否有效) , PSH (缓冲区尚未填满) , RST (表示要求对方重新建立连接) , SYN (建立连接消息标志接) , FIN (表示告知对方本端要关闭连接了)
- 16 位窗口大小：是 TCP 流量控制的一个手段。这里说的窗口，指的是接收通告窗口。它告诉对方本端的 TCP 接收缓冲区还能容纳多少字节的数据，这样对方就可以控制发送数据的速度。
- 16 位校验和：由发送端填充，接收端对 TCP 报文段执行 CRC 算法以检验 TCP 报文段在传输过程中是否损坏。注意，这个校验不仅包括 TCP 头部，也包括数据部分。这也是 TCP 可靠传输的一个重要保障。
- 16 位紧急指针：一个正的偏移量。它和序号字段的值相加表示最后一个紧急数据的下一字节的序号。因此，确切地说，这个字段是紧急指针相对当前序号的偏移，不妨称之为紧急偏移。TCP 的紧急指针是发送端向接收端发送紧急数据的方法。

37.TCP 是如何保证可靠性的？

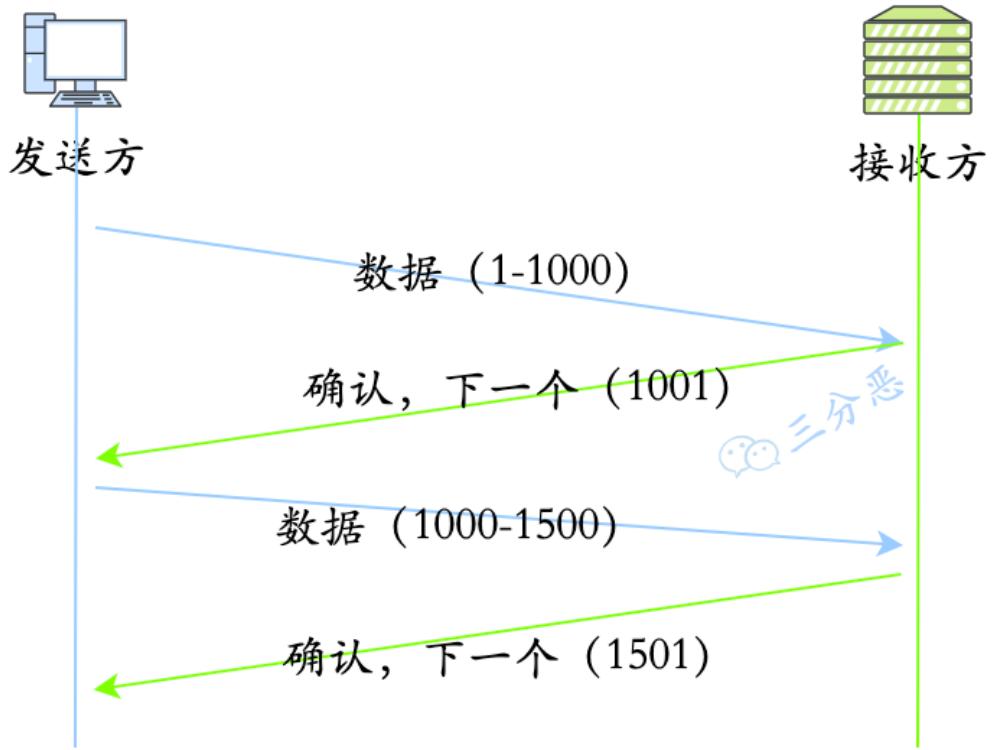
TCP 主要提供了检验和、序列号/确认应答、超时重传、最大消息长度、滑动窗口控制等方法实现了可靠性传输。



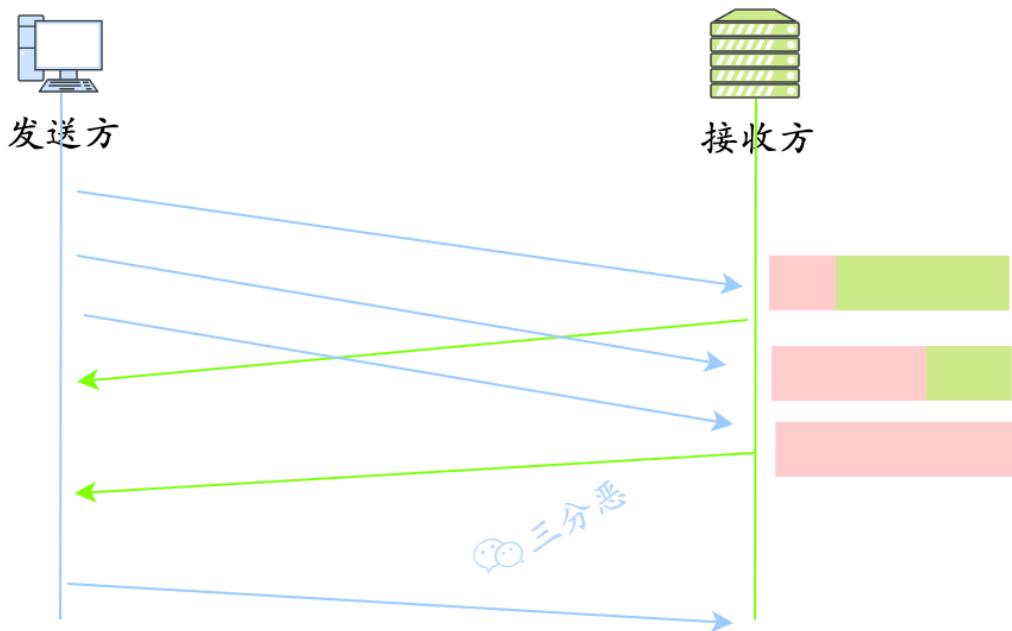
1. 连接管理：TCP使用三次握手和四次挥手保证可靠地建立连接和释放连接，这里就不用多说了。
2. 校验和：TCP将保持它首部和数据的校验和。这是一个端到端的校验和，目的是检测数据在传输过程中的任何变化。如果接收端的校验和有差错，TCP将丢弃这个报文段和不确认收到此报文段。



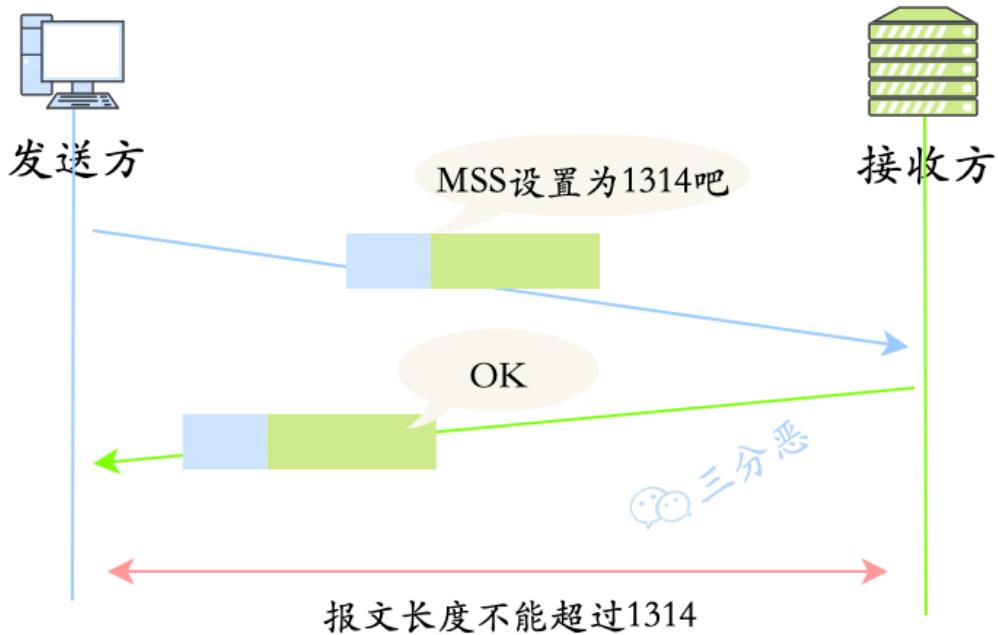
3. 序列号/确认应答：TCP给发送的每一个包进行编号，接收方会对收到的包进行应答，发送方就会知道接收方是否收到对应的包，如果发现没有收到，就会重发，这样就能保证数据的完整性。就像老师上课，会问一句，这一章听懂了吗？没听懂再讲一遍。



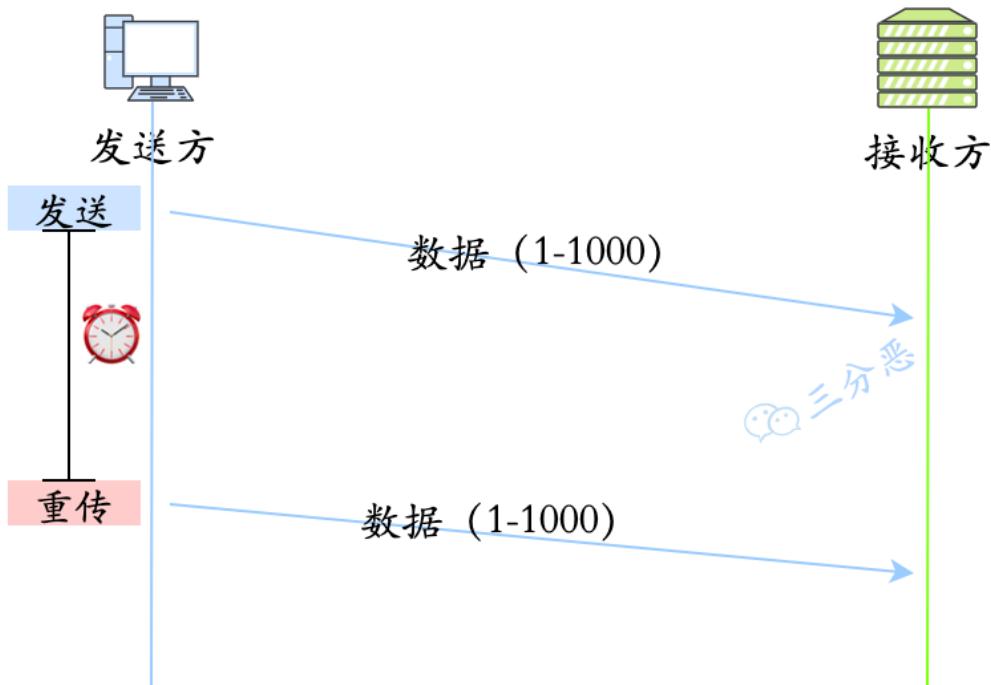
4. 流量控制：TCP 连接的每一方都有固定大小的缓冲空间，TCP的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。TCP 使用的流量控制协议是可变大小的滑动窗口协议。（TCP 利用滑动窗口实现流量控制）



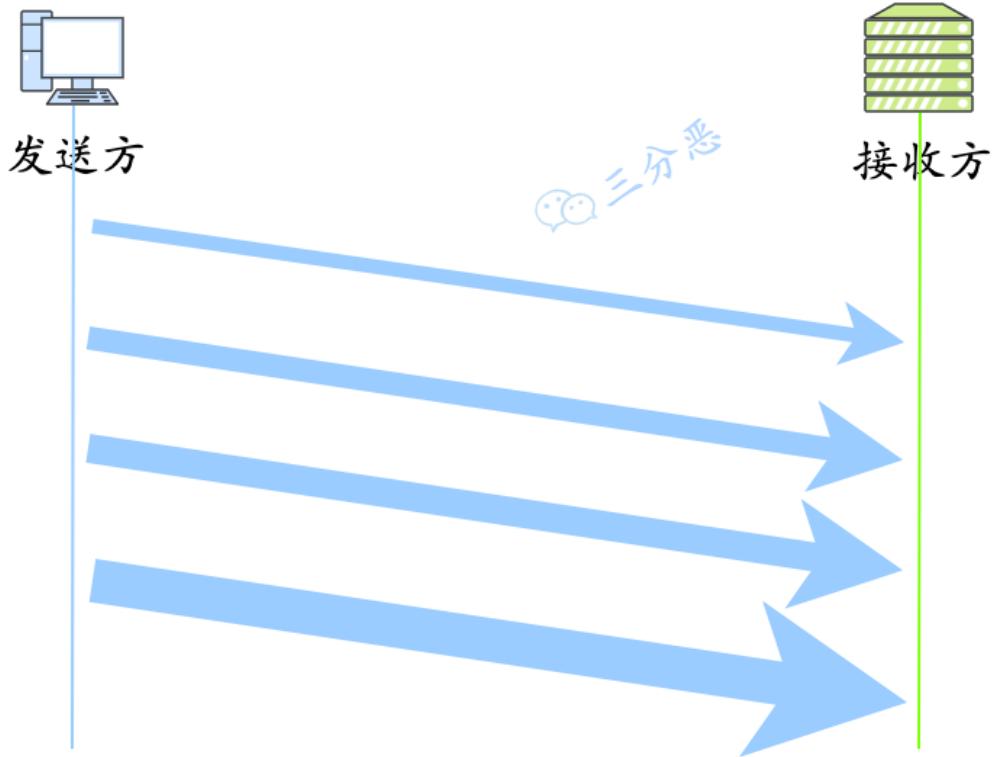
5. 最大消息长度：在建立TCP连接的时候，双方约定一个最大的长度（MSS）作为发送的单位，重传的时候也是以这个单位来进行重传。理想的情况下是该长度的数据刚好不被网络层分块。



6. 超时重传：超时重传是指发送出去的数据包到接收到确认包之间的时间，如果超过了这个时间会被认为是丢包了，需要重传。



7. 拥塞控制：如果网络非常拥堵，此时再发送数据就会加重网络负担，那么发送的数据段很可能超过了最大生存时间也没有到达接收方，就会产生丢包问题。为此TCP引入慢启动机制，先发出少量数据，就像探路一样，先摸清当前的网络拥堵状态后，再决定按照多大的速度传送数据。

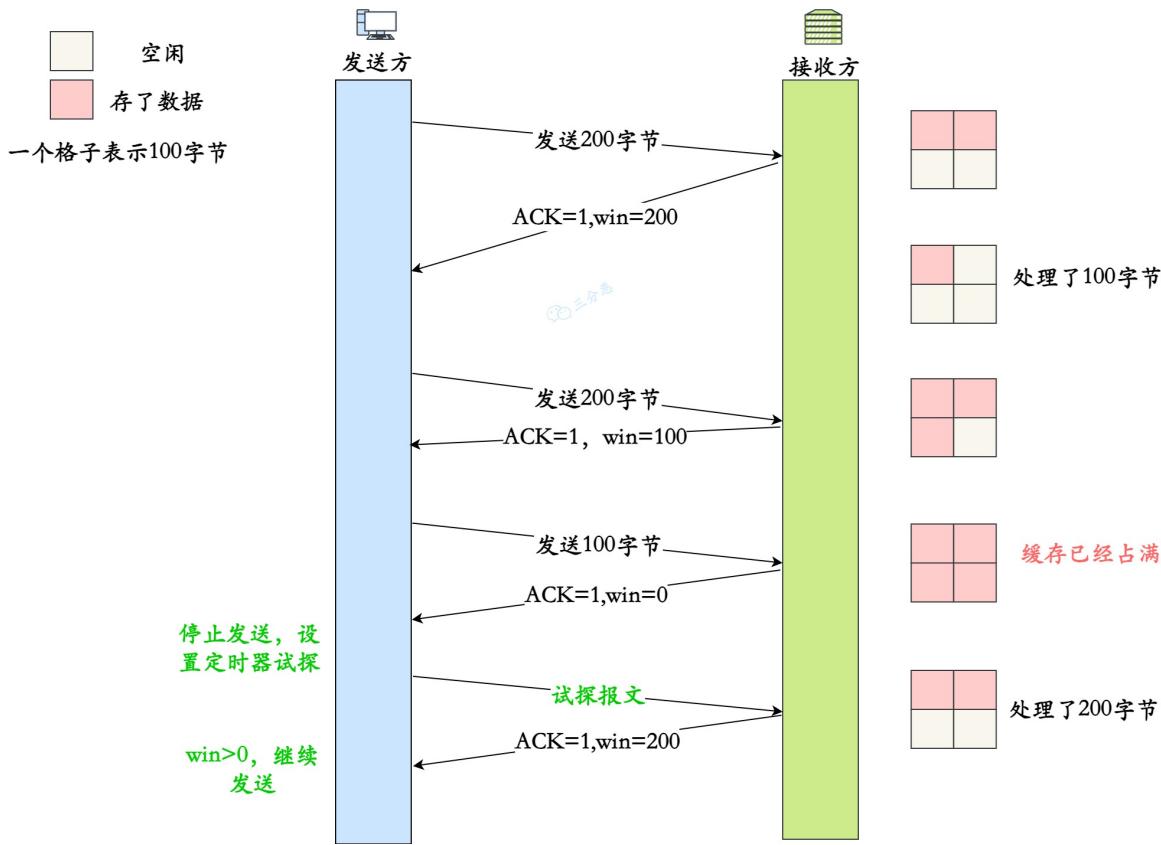


38. 说说 TCP 的流量控制？

TCP 提供了一种机制，可以让发送端根据接收端的实际接收能力控制发送的数据量，这就是**流量控制**。

TCP 通过**滑动窗口**来控制流量，我们看下简要流程：

- 首先双方三次握手，初始化各自的窗口大小，均为 400 个字节。



- 假如当前发送方给接收方发送了 200 个字节，那么，发送方的 `SND.NXT` 会右移 200 个字节，也就是说当前的可用窗口减少了 200 个字节。
- 接受方收到后，放到缓冲队列里面， $REV.WND = 400 - 200 = 200$ 字节，所以 $win = 200$ 字节返回给发送方。接收方会在 ACK 的报文首部带上缩小后的滑动窗口 200 字节
- 发送方又发送 200 字节过来，200 字节到达，继续放到缓冲队列。不过这时候，由于大量负载的原因，接受方处理不了这么多字节，只能处理 100 字节，剩余的 100 字节继续放到缓冲队列。这时候， $REV.WND = 400 - 200 - 100 = 100$ 字节，即 $win = 100$ 返回发送方。
- 发送方继续发送 100 字节过来，这时候，接收窗口 win 变为 0。
- 发送方停止发送，开启一个定时任务，每隔一段时间，就去询问接受方，直到 win 大于 0，才继续开始发送。

39. 详细说说 TCP 的滑动窗口？

TCP 发送一个数据，如果需要收到确认应答，才会发送下一个数据。这样的话就会有个缺点：效率会比较低。

“用一个比喻，我们在微信上聊天，你打完一句话，我回复一句之后，你才能打下一句。假如我没有及时回复呢？你是把话憋着不说吗？然后傻傻等到我回复之后再接着发下一句？”

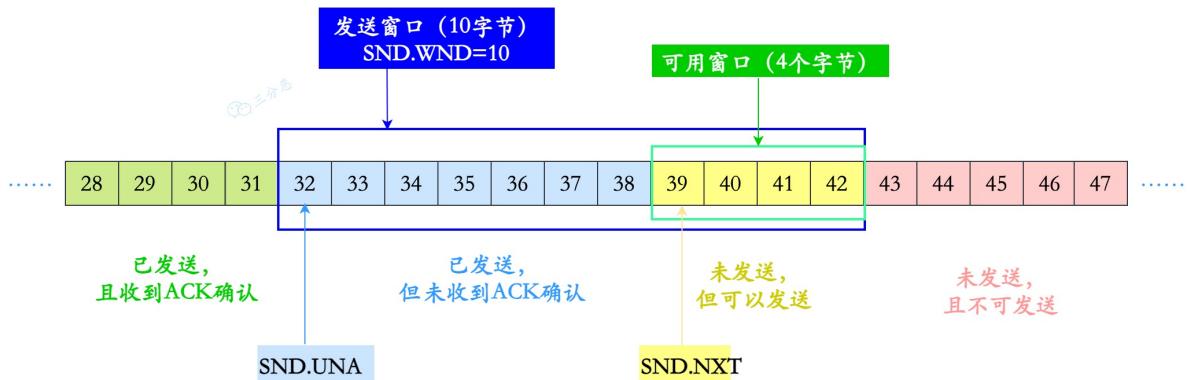
为了解决这个问题，TCP 引入了**窗口**，它是操作系统开辟的一个缓存空间。窗口大小值表示无需等待确认应答，而可以继续发送数据的最大值。

TCP 头部有个字段叫 win，也即那个**16 位的窗口大小**，它告诉对方本端的 TCP 接收缓冲区还能容纳多少字节的数据，这样对方就可以控制发送数据的速度，从而达到**流量控制**的目的。

“通俗点讲，就是接受方每次收到数据包，在发送确认报文的时候，同时告诉发送方，自己的缓存区还有多少空余空间，缓冲区的空余空间，我们就称之为接受窗口大小。这就是 win。”

TCP 滑动窗口分为两种：发送窗口和接收窗口。**发送端的滑动窗口**包含四大部分，如下：

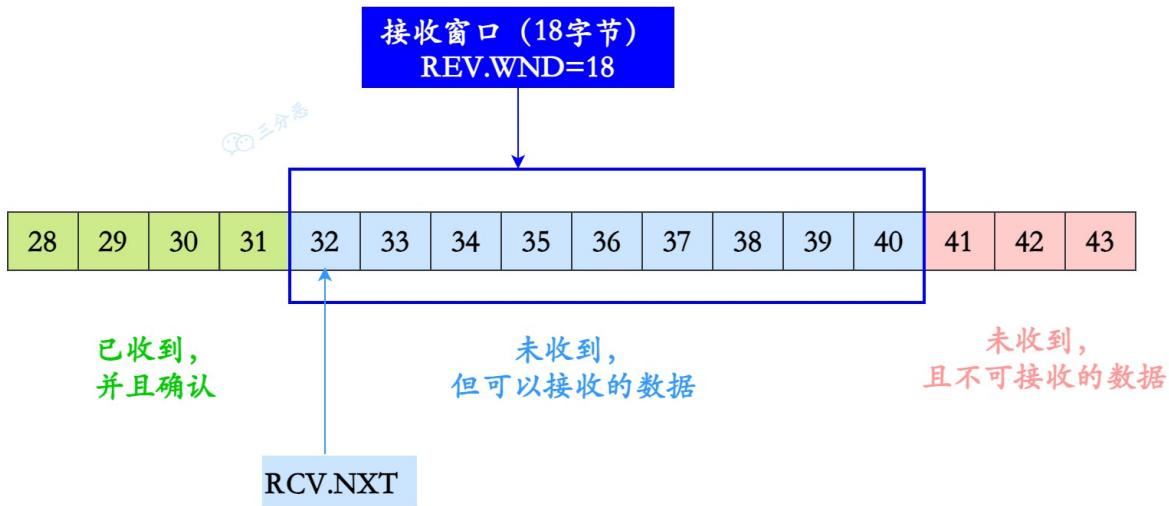
- 已发送且已收到 ACK 确认
- 已发送但未收到 ACK 确认
- 未发送但可以发送
- 未发送也不可以发送



- 深蓝色框里就是发送窗口。
- SND.WND: 表示发送窗口的大小，上图虚线框的格子数是 10 个，即发送窗口大小是 10。
- SND.NXT: 下一个发送的位置，它指向未发送但可以发送的第一个字节的序列号。
- SND.UNA: 一个绝对指针，它指向的是已发送但未确认的第一个字节的序列号。

接收方的滑动窗口包含三大部分，如下：

- 已成功接收并确认
- 未收到数据但可以接收
- 未收到数据并不可以接收的数据

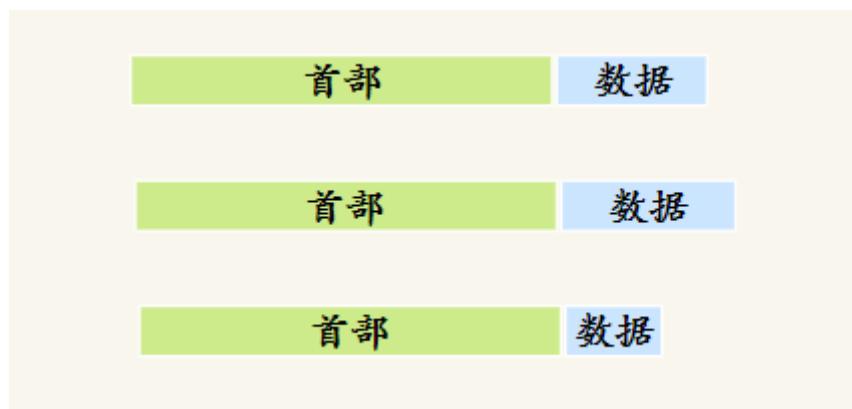


- 蓝色框内，就是接收窗口。
- REV.WND: 表示接收窗口的大小, 上图虚线框的格子就是 9 个。
- RCV.NXT: 下一个接收的位置，它指向未收到但可以接收的第一个字节的序列号。

40. 了解Nagle 算法和延迟确认吗？

Nagle 算法和延迟确认是干什么的？

当我们 TCP 报文的承载的数据非常小的时候，例如几个字节，那么整个网络的效率是很低的，因为每个 TCP 报文中都会有 20 个字节的 TCP 头部，也会有 20 个字节的 IP 头部，而数据只有几个字节，所以在整个报文中有效数据占有的比例就会非常低。



这就好像快递员开着大货车送一个小包裹一样浪费。

那么就出现了常见的两种策略，来减少小报文的传输，分别是：

- Nagle 算法
- 延迟确认

Nagle 算法

Nagle 算法：任意时刻，最多只能有一个未被确认的小段。所谓“小段”，指的是小于 MSS 尺寸的数据块，所谓“未被确认”，是指一个数据块发送出去后，没有收到对方发送的 ACK 确认该数据已收到。

Nagle 算法的策略：

- 没有已发送未确认报文时，立刻发送数据。
- 存在未确认报文时，直到「没有已发送未确认报文」或「数据长度达到 MSS 大小」时，再发送数据。

只要没满足上面条件中的一条，发送方一直在囤积数据，直到满足上面的发送条件。

延迟确认

事实上当没有携带数据的 ACK，它的网络效率也是很低的，因为它也有 40 个字节的 IP 头和 TCP 头，但却没有携带数据报文。

为了解决 ACK 传输效率低问题，所以就衍生出了 TCP 延迟确认。

TCP 延迟确认的策略：

- 当有响应数据要发送时，ACK 会随着响应数据一起立刻发送给对方
- 当没有响应数据要发送时，ACK 将会延迟一段时间，以等待是否有响应数据可以一起发送
- 如果在延迟等待发送 ACK 期间，对方的第二个数据报文又到达了，这时就会立刻发送 ACK

一般情况下，Nagle 算法和延迟确认不能一起使用，Nagle 算法意味着延迟发，延迟确认意味着延迟接收，两个凑在一起就会造成更大的延迟，会产生性能问题。

41. 说说 TCP 的拥塞控制？

什么是拥塞控制？不是有了流量控制吗？

前面的流量控制是避免发送方的数据填满接收方的缓存，但是并不知道整个网络之中发生了什么。

一般来说，计算机网络都处在一个共享的环境。因此也有可能会因为其他主机之间的通信使得网络拥堵。

在网络出现拥堵时，如果继续发送大量数据包，可能会导致数据包时延、丢失等，这时 **TCP** 就会重传数据，但是一重传就会导致网络的负担更重，于是会导致更大的延迟以及更多的丢包，这个情况就会进入恶性循环被不断地放大....

所以，TCP 不能忽略整个网络中发生的事，它被设计成一个无私的协议，当网络发送拥塞时，TCP 会自我牺牲，降低发送的数据流。

于是，就有了拥塞控制，控制的目的就是避免发送方的数据填满整个网络。

就像是一个水管，不能让太多的水（数据流）流入水管，如果超过水管的承受能力，水管会被撑爆（丢包）。



发送方维护一个**拥塞窗口 cwnd (congestion window)** 的变量，调节所要发送数据的量。

什么是拥塞窗口？和发送窗口有什么关系呢？

拥塞窗口 **cwnd** 是发送方维护的一个的状态变量，它会根据网络的拥塞程度动态变化的。

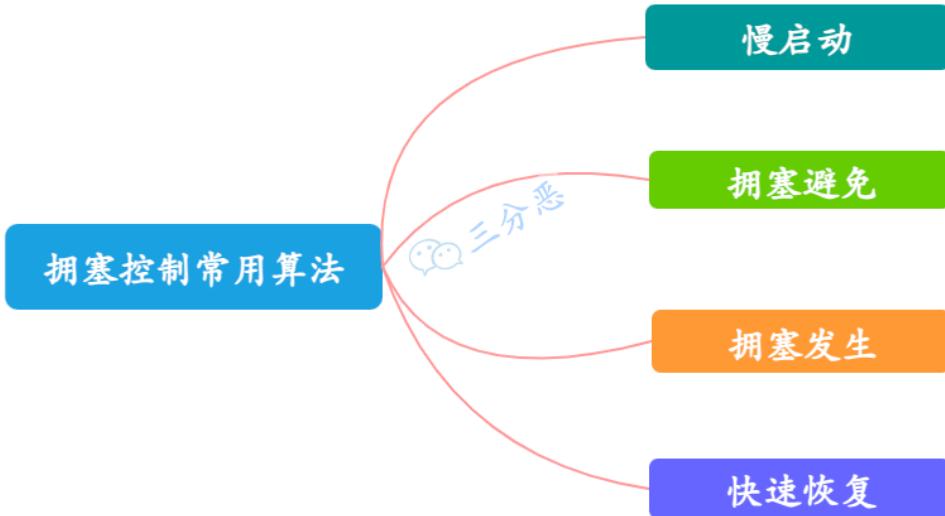
发送窗口 **swnd** 和接收窗口 **rwnd** 是约等于的关系，那么由于加入了拥塞窗口的概念后，此时发送窗口的值是 $\text{swnd} = \min(\text{cwnd}, \text{rwnd})$ ，也就是拥塞窗口和接收窗口中的最小值。

拥塞窗口 **cwnd** 变化的规则：

- 只要网络中没有出现拥塞，**cwnd** 就会增大；
- 但网络中出现了拥塞，**cwnd** 就减少；

拥塞控制有哪些常用算法？

拥塞控制主要有这几种常用算法：



- 慢启动
- 拥塞避免
- 拥塞发生
- 快速恢复

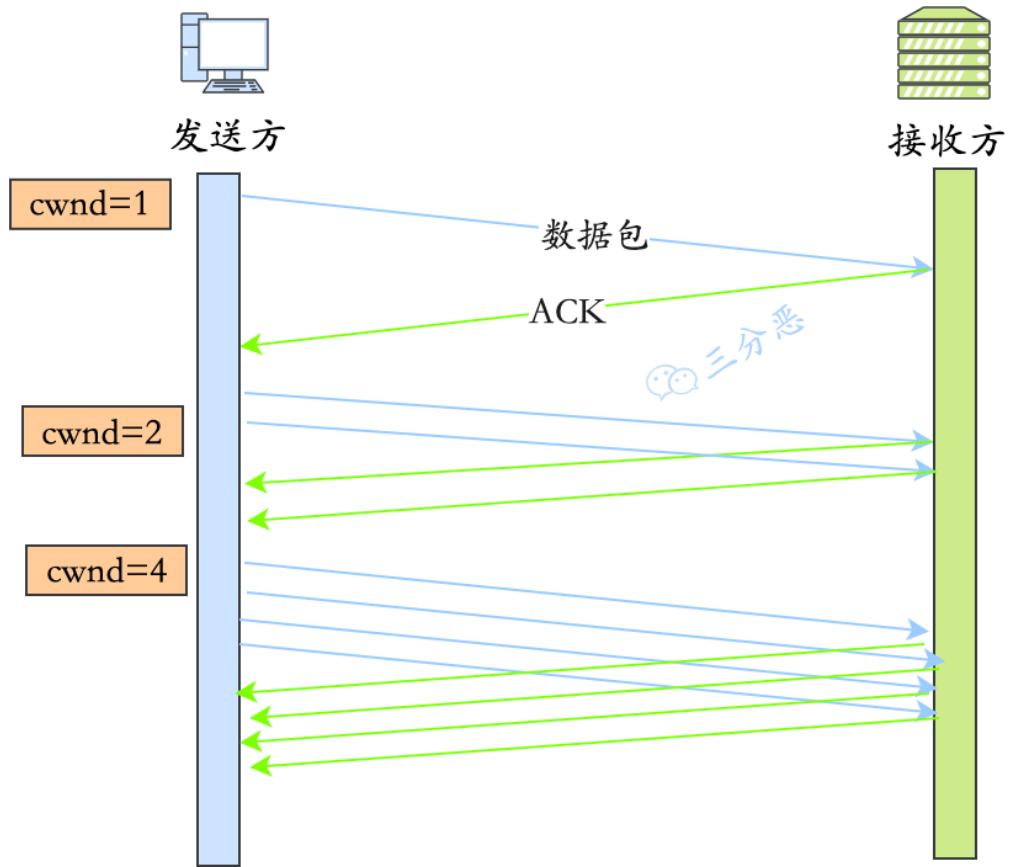
H5 慢启动算法

慢启动算法，慢慢启动。

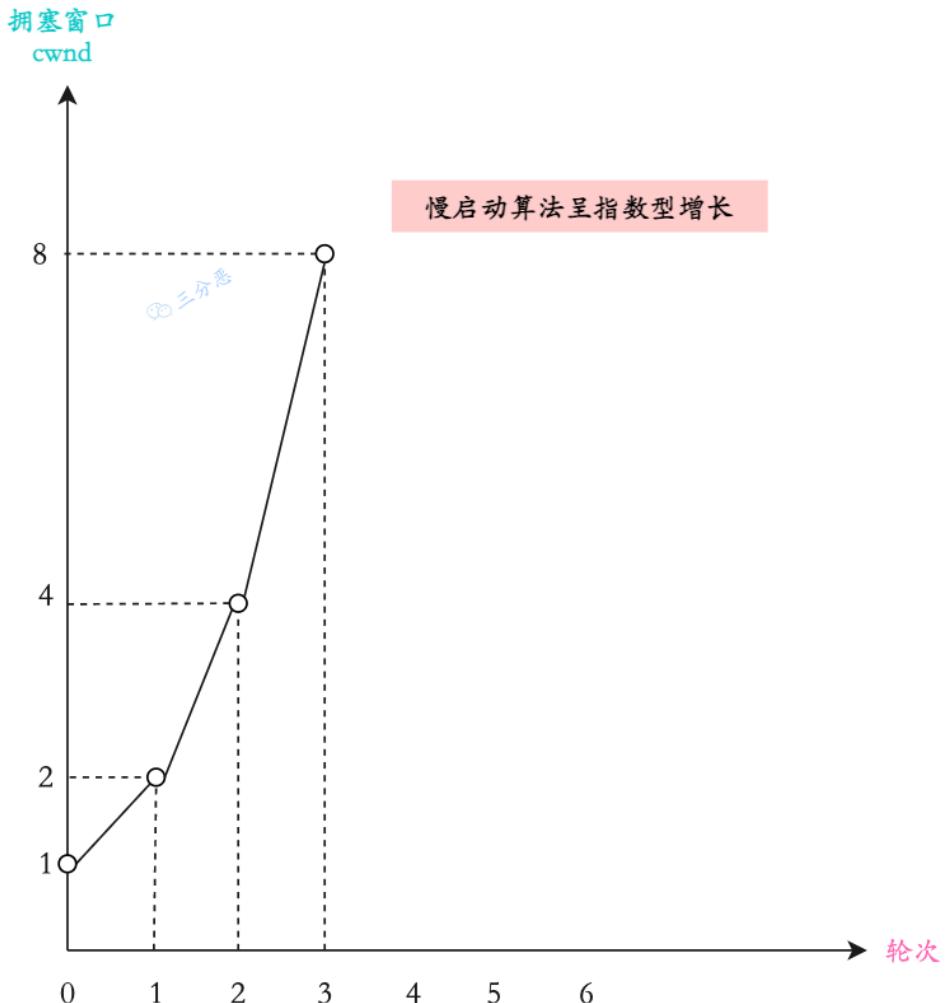
它表示 TCP 建立连接完成后，一开始不要发送大量的数据，而是先探测一下网络的拥塞程度。由小到大逐渐增加拥塞窗口的大小，如果没有出现丢包，**每收到一个 ACK，就将拥塞窗口 cwnd 大小就加 1（单位是 MSS）**。每轮次发送窗口增加一倍，呈指数增长，如果出现丢包，拥塞窗口就减半，进入拥塞避免阶段。

举个例子：

- 连接建立完成后，一开始初始化 $cwnd = 1$ ，表示可以传一个 MSS 大小的数据。
- 当收到一个 ACK 确认应答后， $cwnd$ 增加 1，于是一次能够发送 2 个
- 当收到 2 个的 ACK 确认应答后， $cwnd$ 增加 2，于是就可以比之前多发 2 个，所以这一次能够发送 4 个
- 当这 4 个的 ACK 确认到来的时候，每个确认 $cwnd$ 增加 1，4 个确认 $cwnd$ 增加 4，于是就可以比之前多发 4 个，所以这一次能够发送 8 个。



发包的个数是指数性的增长。



为了防止 cwnd 增长过大引起网络拥塞，还需设置一个**慢启动阀值 ssthresh**（slow start threshold）状态变量。当 **cwnd** 到达该阀值后，就好像水管被关小了水龙头一样，减少拥塞状态。即当 **cwnd > ssthresh** 时，进入了**拥塞避免**算法。

H5 拥塞避免算法

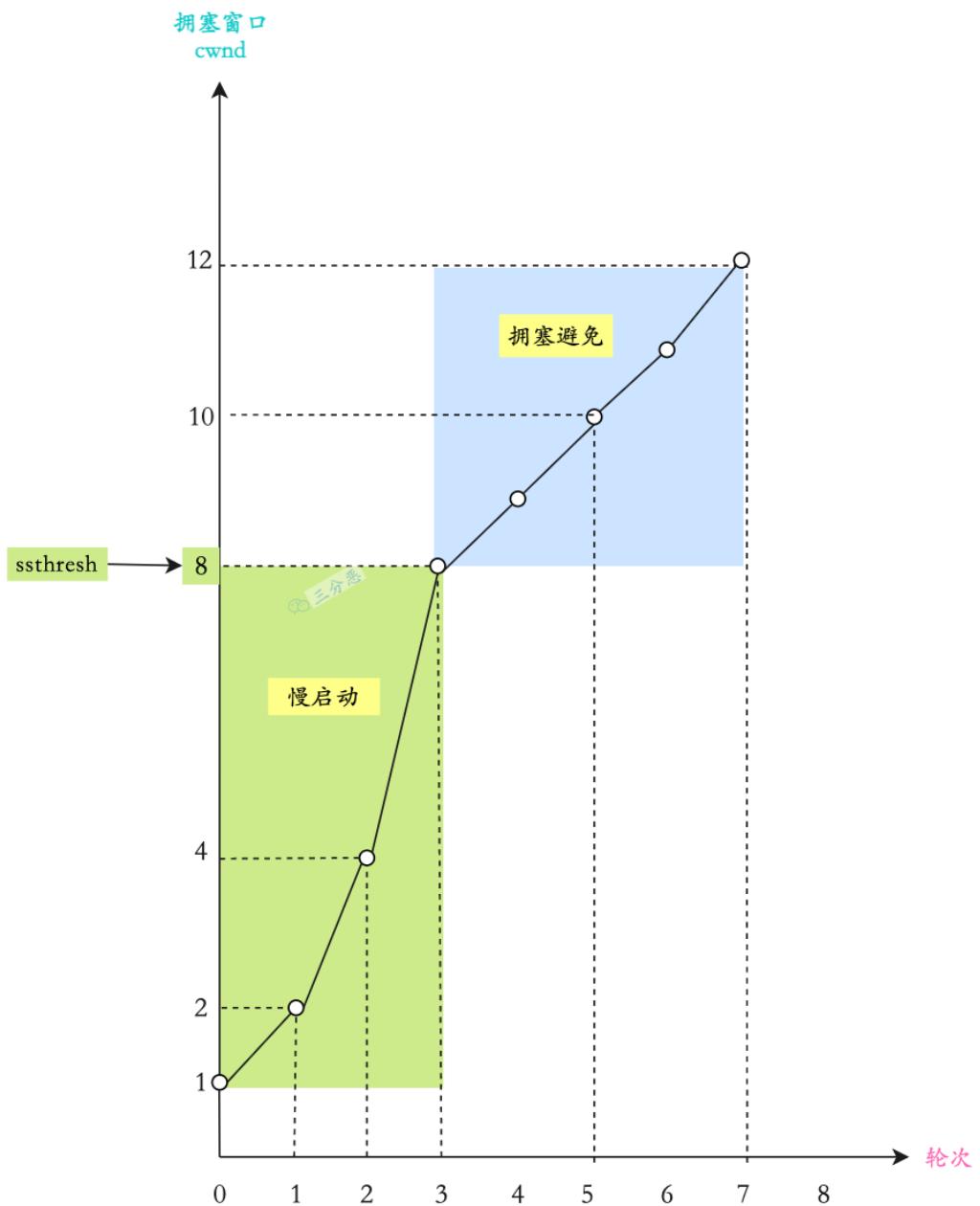
一般来说，慢启动阀值 ssthresh 是 65535 字节，**cwnd** 到达**慢启动阀值**后

- 每收到一个 ACK 时， $cwnd = cwnd + 1/cwnd$
- 当每过一个 RTT 时， $cwnd = cwnd + 1$

显然这是一个线性上升的算法，避免过快导致网络拥塞问题。

接着上面慢启动的例子，假定 ssthresh 为 8 :

- 当 8 个 ACK 应答确认到来时，每个确认增加 $1/8$ ，8 个 ACK 确认 cwnd 一共增加 1，于是这一次能够发送 9 个 MSS 大小的数据，变成了线性增长。



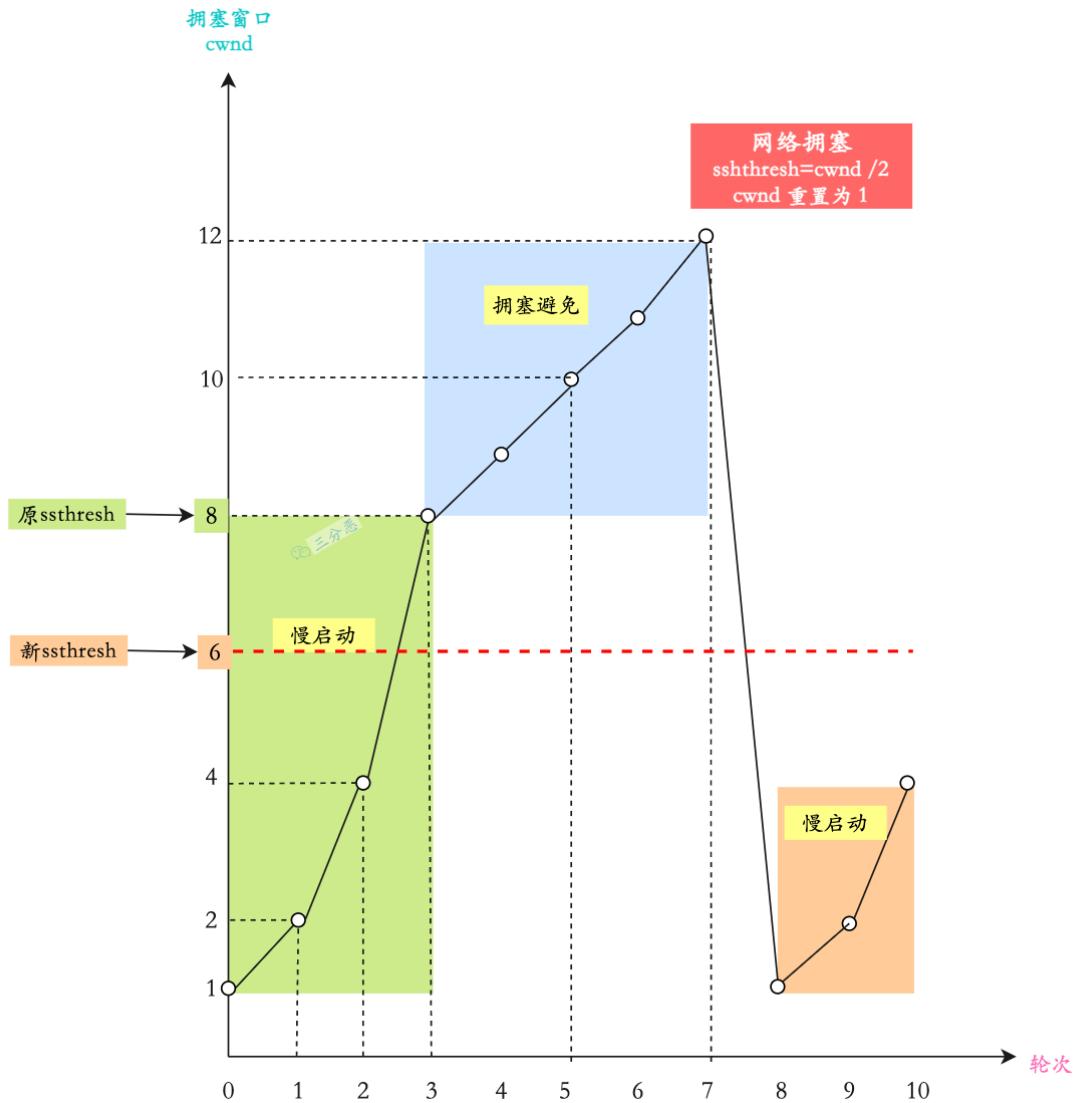
H5 拥塞发生

当网络拥塞发生 **丢包** 时，会有两种情况：

- RTO 超时重传
- 快速重传

如果是发生了 **RTO 超时重传**，就会使用拥塞发生算法

- 慢启动阀值 $ssthresh = cwnd / 2$
- $cwnd$ 重置为 1
- 进入新的慢启动过程



这种方式就像是飙车的时候急刹车，还飞速倒车，这。。。

其实还有更好的处理方式，就是**快速重传**。发送方收到3个连续重复的ACK时，就会快速地重传，不必等待**RTO超时**再重传。

发生快速重传的拥塞发生算法：

- 拥塞窗口大小 $cwnd = cwnd/2$
- 慢启动阀值 $ssthresh = cwnd$
- 进入快速恢复算法

H5 快速恢复

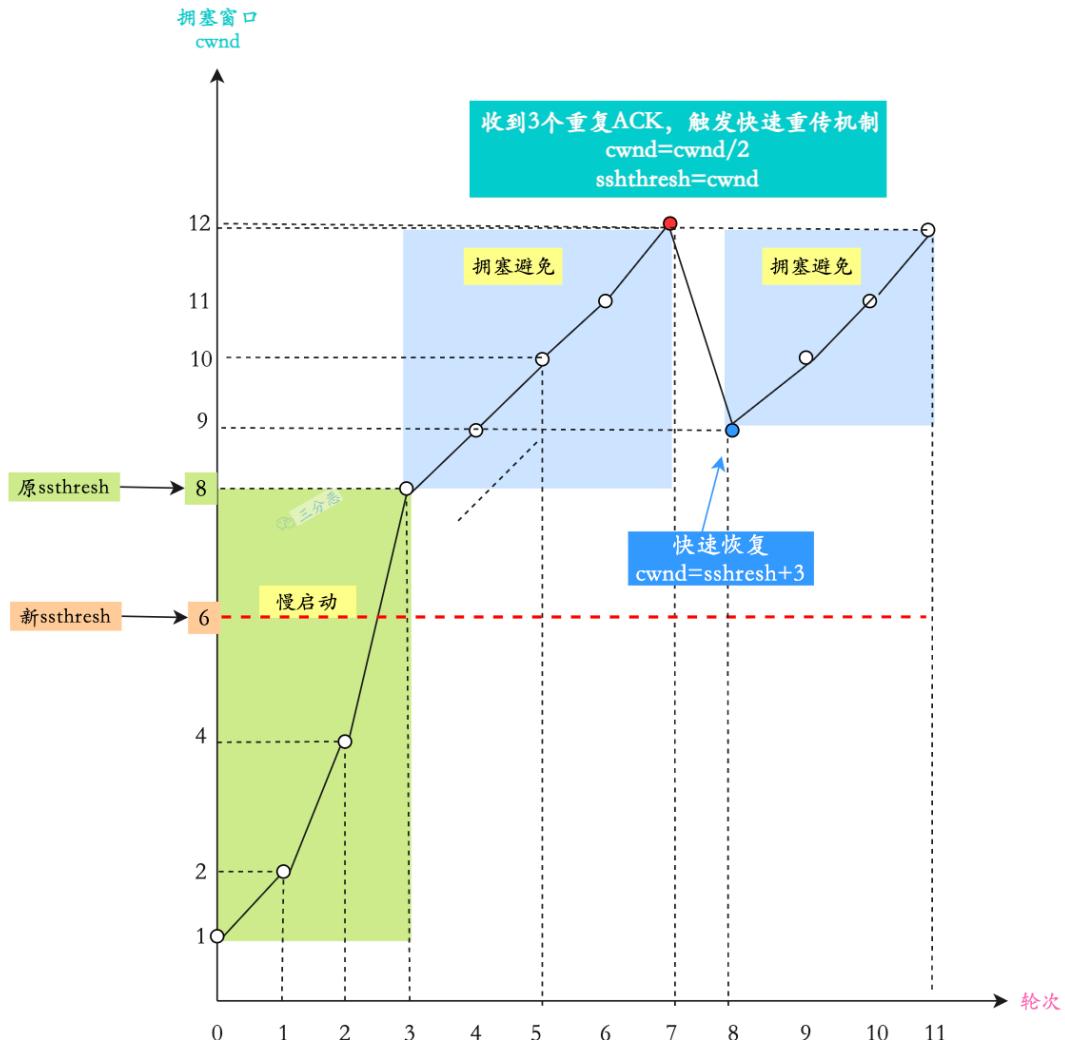
快速重传和快速恢复算法一般同时使用。快速恢复算法认为，还有3个重复ACK收到，说明网络也没那么糟糕，所以没有必要像RTO超时那么强烈。

正如前面所说，进入快速恢复之前， $cwnd$ 和 $ssthresh$ 已被更新：

- $cwnd = cwnd /2$
- $ssthresh = cwnd$

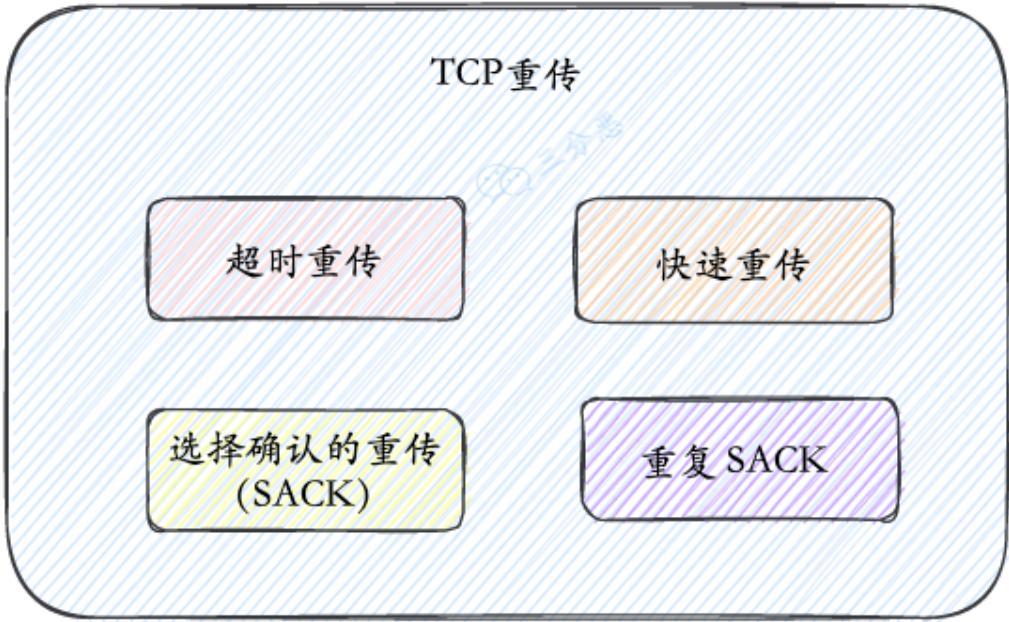
然后，进入快速恢复算法如下：

- $cwnd = sshthresh + 3$
- 重传重复的那几个 ACK（即丢失的那个数据包）
- 如果再收到重复的 ACK，那么 $cwnd = cwnd + 1$
- 如果收到新数据的 ACK 后， $cwnd = sshthresh$ 。因为收到新数据的 ACK，表明恢复过程已经结束，可以再次进入了拥塞避免的算法了。



42. 说说 TCP 的重传机制？

重传包括超时重传、快速重传、带选择确认的重传（SACK）、重复 SACK 四种。

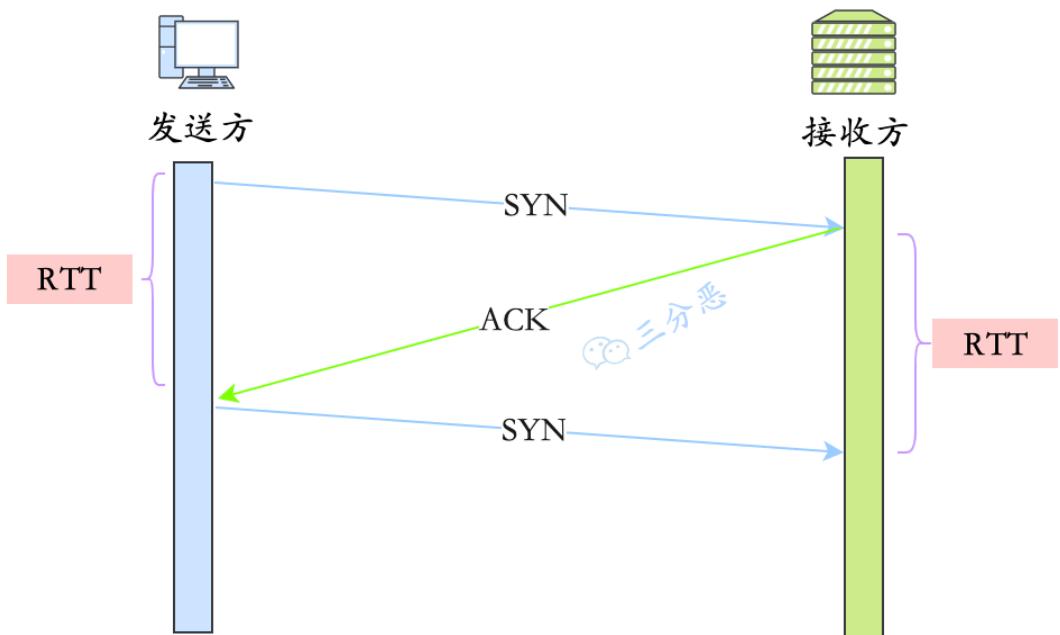


H5 超时重传

超时重传，是 TCP 协议保证数据可靠性的另一个重要机制，其原理是在发送某一个数据以后就开启一个计时器，在一定时间内如果没有得到发送的数据报的 ACK 报文，那么就重新发送数据，直到发送成功为止。

超时时间应该设置为多少呢？

先来看下什么叫 **RTT (Round-Trip Time, 往返时间)**。



RTT 就是数据完全发送完，到收到确认信号的时间，即数据包的一次往返时间。

超时重传时间，就是 RTO (Retransmission Timeout)。那么，**RTO 到底设置多大呢？**

- 如果 RTO 设置很大，等了很久都没重发，这样肯定就不行。
- 如果 RTO 设置很小，那很可能数据都没有丢失，就开始重发了，这会导致网络阻塞，从而恶性循环，导致更多的超时出现。

一般来说，RTO 略微大于 RTT，效果是最佳的。

其实，RTO 有个标准方法的计算公式，也叫 **Jacobson / Karels 算法**。

1. 首先计算 SRTT (即计算平滑的 RTT)

```
1 | SRTT = (1 - α) * SRTT + α * RTT //求 SRTT 的加权平均
```

2. 其次，计算 RTTVAR (round-trip time variation)

```
1 | RTTVAR = (1 - β) * RTTVAR + β * (|RTT - SRTT|) //计算 SRTT 与  
真实值的差距
```

3. 最后，得出最终的 RTO

```
1 | RTO = μ * SRTT + δ * RTTVAR = SRTT + 4 · RTTVAR
```

在 Linux 下， **$\alpha = 0.125$** ， **$\beta = 0.25$** ， **$\mu = 1$** ， **$\delta = 4$** 。别问这些参数是怎么来的，它们是大量实践，调出的最优参数。

超时重传不是十分完美的重传方案，它有这些缺点：

- 当一个报文丢失时，会等待一定的超时周期，才重传分组，增加了端到端的时延。
- 当一个报文丢失时，在其等待超时的过程中，可能会出现这种情况：其后的报文段已经被接收端接收但却迟迟得不到确认，发送端会认为也丢失了，从而引起不必要的重传，既浪费资源也浪费时间。

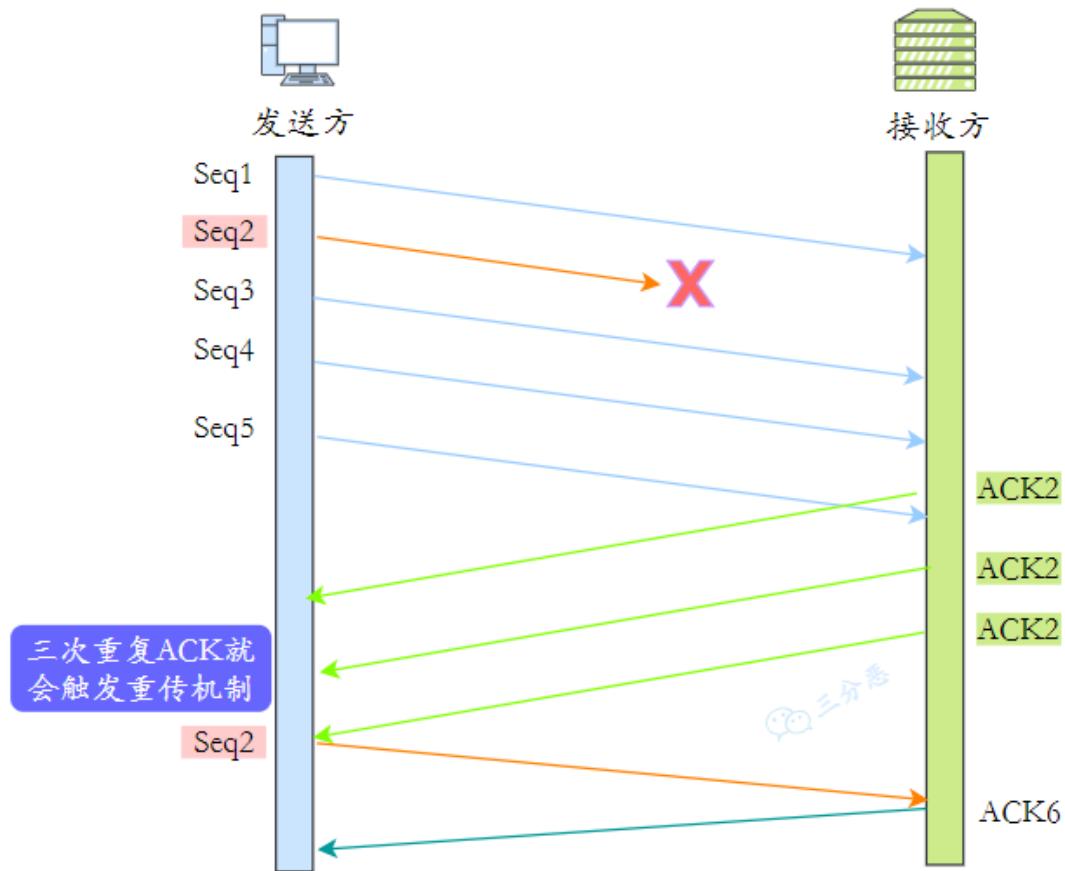
并且，对于 TCP，如果发生一次超时重传，时间间隔下次就会加倍。

H5 快速重传

TCP 还有另外一种快速重传 (**Fast Retransmit**) 机制，它不以时间为驱动，而是以数据驱动重传。

它不以时间驱动，而是以数据驱动。它是基于接收端的反馈信息来引发重传的。

可以用它来解决超时重发的时间等待问题，快速重传流程如下：



在上图，发送方发出了 1, 2, 3, 4, 5 份数据：

- 第一份 Seq1 先送到了，于是就 Ack 回 2；
- 结果 Seq2 因为某些原因没收到，Seq3 到达了，于是还是 Ack 回 2；
- 后面的 Seq4 和 Seq5 都到了，但还是 Ack 回 2，因为 Seq2 还是没有收到；
- 发送端收到了三个 Ack = 2 的确认，知道了 Seq2 还没有收到，就会在定时器过期之前，重传丢失的 Seq2。
- 最后，收到了 Seq2，此时因为 Seq3, Seq4, Seq5 都收到了，于是 Ack 回 6。

快速重传机制只解决了一个问题，就是超时时间的问题，但是它依然面临着另外一个问题。就是重传的时候，是重传之前的一个，还是重传所有的问题。

比如对于上面的例子，是重传 Seq2 呢？还是重传 Seq2、Seq3、Seq4、Seq5 呢？因为发送端并不清楚这连续的三个 Ack 2 是谁传回来的。

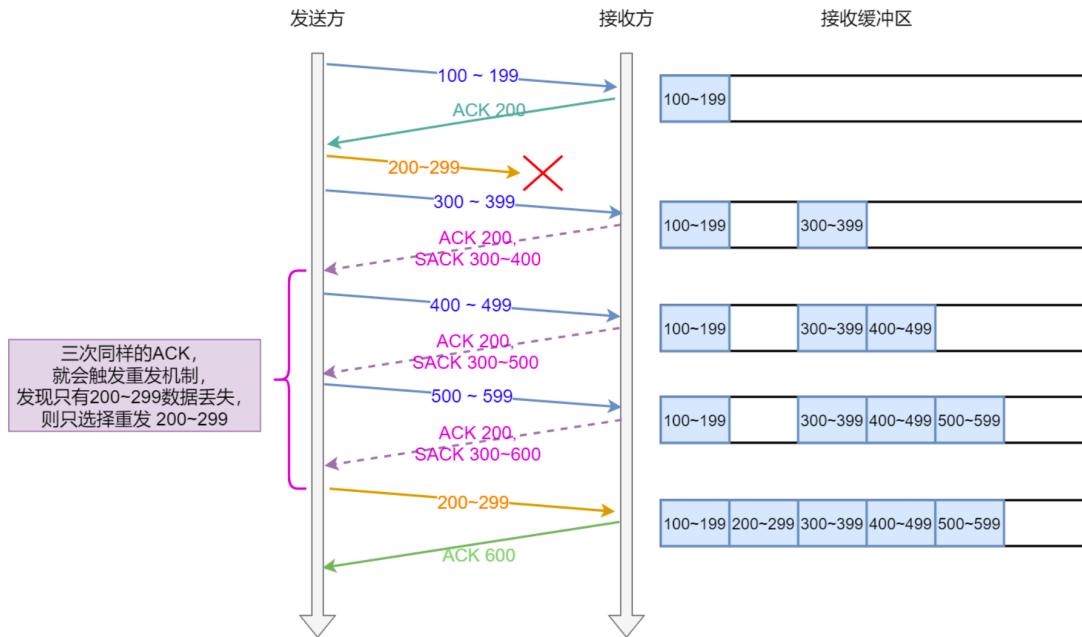
根据 TCP 不同的实现，以上两种情况都是有可能的。可见，这是一把双刃剑。

为了解决不知道该重传哪些 TCP 报文，于是就有 SACK 方法。

H5 带选择确认的重传（SACK）

为了解决应该重传多少个包的问题? TCP 提供了带选择确认的重传（即 SACK，Selective Acknowledgment）。

SACK 机制就是，在快速重传的基础上，接收方返回最近收到报文段的序列号范围，这样发送方就知道接收方哪些数据包是没收到的。这样就很清楚应该重传哪些数据包。



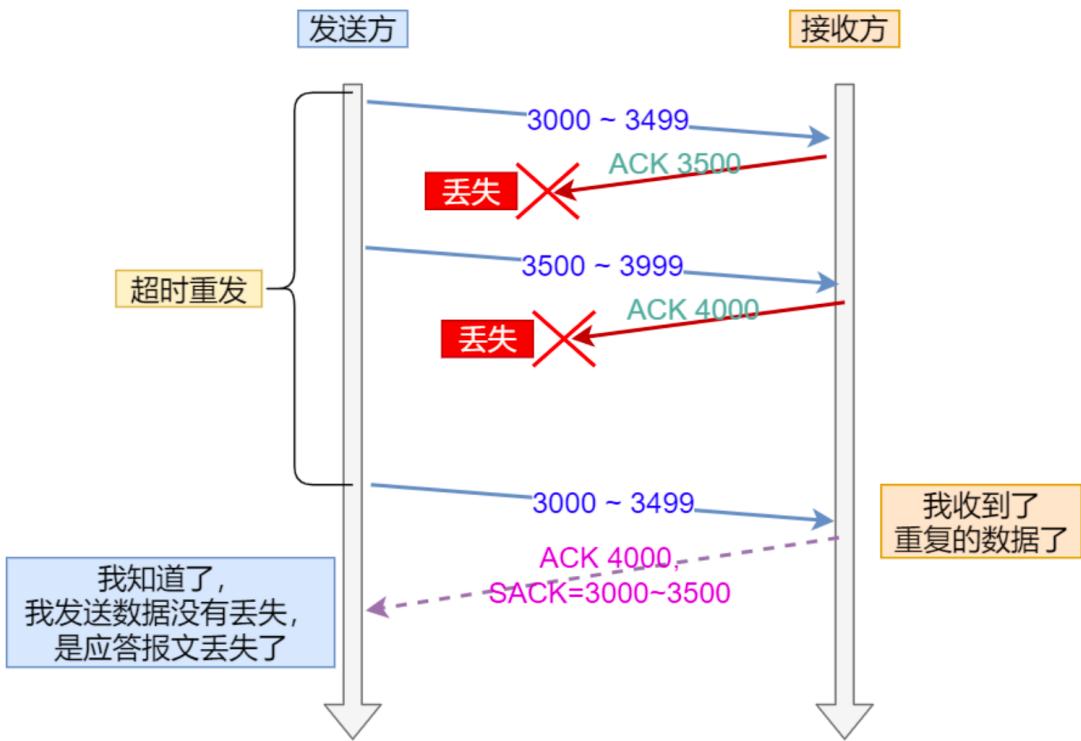
如上图中，发送方收到了三次同样的 ACK 确认报文，于是就会触发快速重发机制，通过 SACK 信息发现只有200~299 这段数据丢失，则重发时，就只选择了这个 TCP 段进行重发。

H5 重复 SACK (D-SACK)

D-SACK，英文是 Duplicate SACK，是在 SACK 的基础上做了一些扩展，主要用来告诉发送方，有哪些数据包，自己重复接受了。

DSACK 的目的是帮助发送方判断，是否发生了包失序、ACK 丢失、包重复或伪重传。让 TCP 可以更好的做网络流控。

例如ACK丢包导致的数据包重复：



- 接收方发给发送方的两个 ACK 确认应答都丢失了，所以发送方超时后，重传第一个数据包（3000 ~ 3499）
- 于是接收方发现数据是重复收到的，于是回了一个 $SACK = 3000 \sim 3500$ ，告诉「发送方」 3000~3500的数据早已被接收了，因为 ACK 都到了 4000 了，已经意味着 4000 之前的所有数据都已收到，所以这个SACK 就代表着 D-SACK。这样发送方就知道了，数据没有丢，是接收方的 ACK 确认报文丢了。

43. 说说TCP 的粘包和拆包？

TCP 的粘包和拆包更多的是业务上的概念！

什么是TCP粘包和拆包？

TCP 是面向流，没有界限的一串数据。TCP 底层并不了解上层业务数据的具体含义，它会根据 TCP 缓冲区的实际情况进行包的划分，所以在业务上认为，一个完整的包可能会被 TCP 拆分成多个包进行发送，也有可能把多个小的包封装成一个大的数据包发送，这就是所谓的 TCP 粘包和拆包问题。



正常情况:



粘包:



拆包:



拆包/粘包:



为什么会产生粘包和拆包呢?

- 要发送的数据小于 TCP 发送缓冲区的大小, TCP 将多次写入缓冲区的数据一次发送出去, 将会发生粘包;
- 接收数据端的应用层没有及时读取接收缓冲区中的数据, 将发生粘包;
- 要发送的数据大于 TCP 发送缓冲区剩余空间大小, 将会发生拆包;
- 待发送数据大于 MSS (最大报文长度), TCP 在传输前将进行拆包。即 TCP 报文长度 - TCP 头部长度 > MSS。

那怎么解决呢?

- 发送端将每个数据包封装为固定长度
- 在数据尾部增加特殊字符进行分割
- 将数据分为两部分, 一部分是头部, 一部分是内容体; 其中头部结构大小固定, 且有一个字段声明内容体的大小。

UDP

UDP用的不多, 基本上是被拿来和TCP比较。

44. 说说 TCP 和 UDP 的区别?

最根本区别: **TCP 是面向连接, 而 UDP 是无连接。**

类型	TCP	UDP
是否面向连接	是	否
传输可靠性	可靠	不可靠
传输形式	字节流	数据段报文
传输效率	慢	快
所需资源	多	少
应用场景	文件传输、邮件传输 	即时通讯，域名转换
首部字节	20-60	8个字节

可以这么形容：TCP是打电话， UDP是大喇叭。

TCP：电话

(图片来源广告8848钛合金智能手机，纵享奢华)



UDP：广播

(图片来源旺仔牛奶广告-三年六班李子明同学，你妈妈给你带了一罐旺仔牛奶)

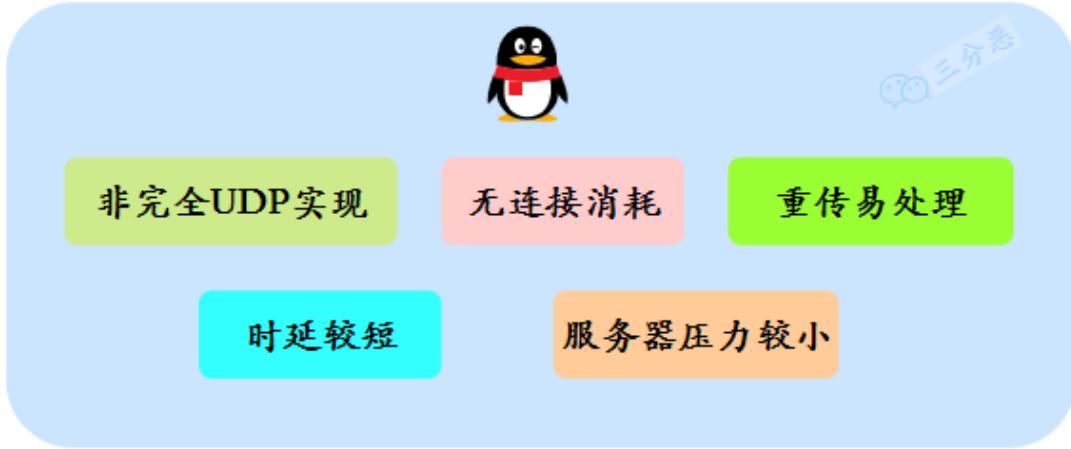


说说TCP和UDP的应用场景？

- TCP应用场景： 效率要求相对低，但对准确性要求相对高的场景。因为传输中需要对数据确认、重发、排序等操作，相比之下效率没有UDP高。例如：文件传输（准确高要求高、但是速度可以相对慢）、收发邮件、远程登录。
- UDP应用场景： 效率要求相对高，对准确性要求相对低的场景。例如：QQ聊天、在线视频、网络语音电话（即时通讯，速度要求高，但是出现偶尔断续不是太大问题，并且此处完全不可以使用重发机制）、广播通信（广播、多播）。

45.为什么QQ采用UDP协议？

PS：这是多年前的老题了，拉出来怀怀旧。



- 首先，QQ并不是完全基于UDP实现。比如在使用QQ进行文件传输等活动的时候，就会使用TCP作为可靠传输的保证。
- 使用UDP进行交互通信的好处在于，延迟较短，对数据丢失的处理比较简单。同时，TCP是一个全双工协议，需要建立连接，所以网络开销也会相对大。
- 如果使用QQ语音和QQ视频的话，UDP的优势就更为突出了，首先延迟较小。最重要的一点是不可靠传输，这意味着如果数据丢失的话，不会有重传。因为用户一般来说可以接受图像稍微模糊一点，声音稍微不清晰一点，但是如果在几秒钟以后再出现之前丢失的画面和声音，这恐怕是很难接受的。
- 由于QQ的服务器设计容量是海量级的应用，一台服务器要同时容纳十几万的并发连接，因此服务器端只有采用UDP协议与客户端进行通讯才能保证这种超大规模的服务

简单总结一下：UDP协议是无连接方式的协议，它的效率高，速度快，占资源少，对服务器的压力比较小。但是其传输机制为不可靠传送，必须依靠辅助的算法来完成传输控制。QQ采用的通信协议以UDP为主，辅以TCP协议。

46. UDP协议为什么不可靠？

UDP在传输数据之前不需要先建立连接，远地主机的运输层在接收到UDP报文后，不需要确认，提供不可靠交付。总结就以下四点：

- 不保证消息交付：不确认，不重传，无超时
- 不保证交付顺序：不设置包序号，不重排，不会发生队首阻塞
- 不跟踪连接状态：不必建立连接或重启状态机
- 不进行拥塞控制：不内置客户端或网络反馈机制

47.DNS为什么要用UDP?

更准确地说，DNS既使用TCP又使用UDP。

当进行区域传送（主域名服务器向辅助域名服务器传送变化的那部分数据）时会使用TCP，因为数据同步传送的数据量比一个请求和应答的数据量要多，而TCP允许的报文长度更长，因此为了保证数据的正确性，会使用基于可靠连接的TCP。

当客户端想DNS服务器查询域名（域名解析）的时候，一般返回的内容不会超过 UDP报文的最大长度，即512字节，用UDP传输时，不需要创建连接，从而大大提高了响应速度，但这要求域名解析服务器和域名服务器都必须自己处理超时和重传从而保证可靠性。

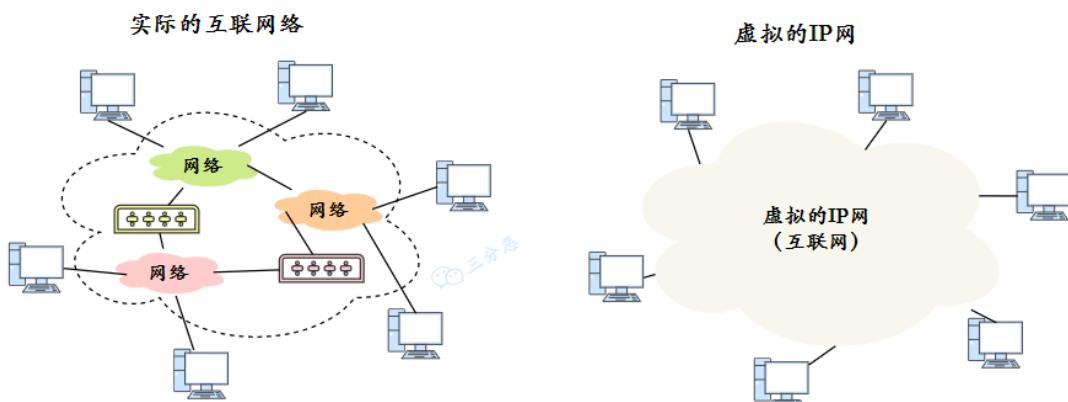
IP

48.IP 协议的定义和作用？

IP协议是什么？

IP协议（Internet Protocol）又被称为互联网协议，是支持网间互联的数据包协议，工作在**网际层**，主要目的就是为了提高网络的可扩展性。

通过**网际协议IP**，可以把参与互联的，性能各异的网络**看作一个统一的网络**。



和传输层TCP相比，IP协议是一种无连接/不可靠、尽力而为的数据包传输服务，和TCP协议一起构成了TCP/IP协议的核心。

IP协议有哪些作用？

IP协议主要有以下几个作用：

- 寻址和路由：在IP数据报中携带源IP地址和目的IP地址来表示该数据包的源主机和目标主机。IP数据报在传输过程中，每个中间节点（IP网关、路由器）只根据网络地址来进行转发，如果中间节点是路由器，则路由器会根据路由表选择合适的路径。IP协议根据路由选择协议提供的路由信息对IP数据报进行转发，直至目标主机。
- 分段和重组：IP数据报在传输过程中可能会经过不同的网络，在不同的网络中数据报的最大长度限制是不同的，IP协议通过给每个IP数据报分配一个标识符以及分段与组装的相关信息，使得数据报在不同的网络中能够被传输，被分段后的IP数据报可以独立地在网络中进行转发，在达到目标主机后由目标主机完成重组工作，恢复出原来的IP数据报。

传输层协议和网络层协议有什么区别？

网络层协议负责提供主机间的逻辑通信；传输层协议负责提供进程间的逻辑通信。

49. IP 地址有哪些分类？

一个IP地址在整个互联网范围内是唯一的，一般可以这么认为，IP 地址 = {<网络号>, <主机号>}。

1. 网络号：它标志主机所连接的网络地址表示属于互联网的哪一个网络。
2. 主机号：它标志主机地址表示其属于该网络中的哪一台主机。

IP 地址分为 A, B, C, D, E 五大类：

- A 类地址 (1~126): 以 0 开头，网络号占前 8 位，主机号占后面 24 位。
- B 类地址 (128~191): 以 10 开头，网络号占前 16 位，主机号占后面 16 位。
- C 类地址 (192~223): 以 110 开头，网络号占前 24 位，主机号占后面 8 位。
- D 类地址 (224~239): 以 1110 开头，保留为多播地址。
- E 类地址 (240~255): 以 1111 开头，保留位为将来使用



50. 域名和IP的关系？一个IP可以对应多个域名吗？

- IP地址在同一个网络中是惟一的，用来标识每一个网络上的设备，其相当于一个人的身份证号
- 域名在同一个网络中也是惟一的，就像是一个人的名字、绰号

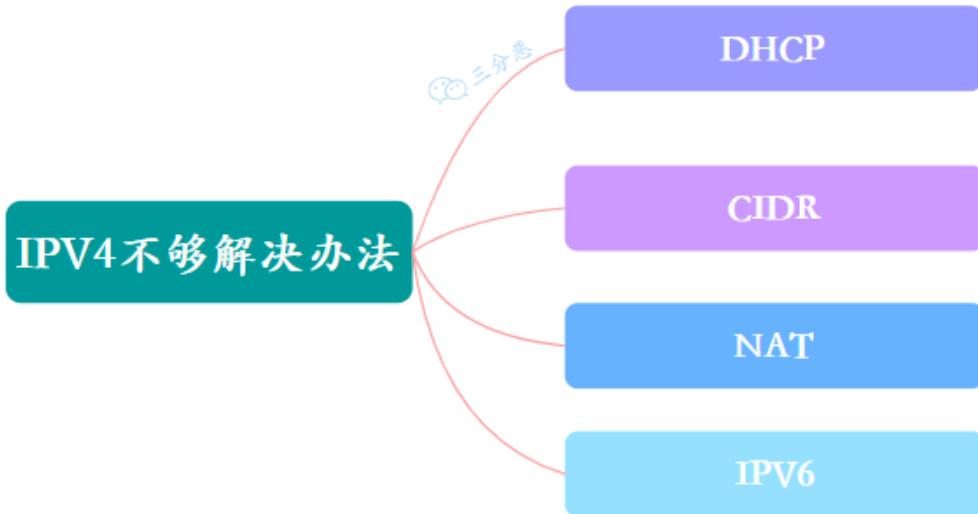
假如你有多个不用的绰号，你的朋友可以用其中任何一个绰号叫你，但你的身份证号码却是惟一的。但同时你的绰号也可能和别人重复，假如你不在，有人叫你的绰号，其它人可能就答应了。

一个域名可以对应多个IP，但这种情况DNS做负载均衡的，在用户访问过程中，一个域名只能对应一个IP。

而一个IP却可以对应多个域名，是一对多的关系。

51. IPV4 地址不够如何解决？

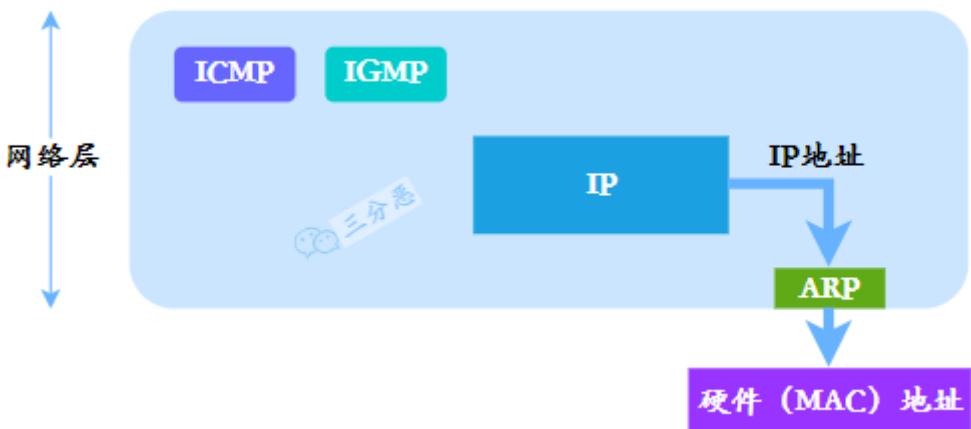
我们知道，IP地址有32位，可以标记 2^{32} 个地址，听起来很多，但是全球的网络设备数量已经远远超过这个数字，所以IPV4地址已经不够用了，那怎么解决呢？



- **DHCP**: 动态主机配置协议，动态分配IP地址，只给接入网络的设备分配IP地址，因此同一个MAC地址的设备，每次接入互联网时，得到的IP地址不一定是相同的，该协议使得空闲的IP地址可以得到充分利用。
- **CIDR**: 无类别域间路由。CIDR消除了传统的A类、B类、C类地址以及划分子网的概念，因而更加有效地分配IPv4的地址空间，但无法从根本上解决地址耗尽的问题。
- **NAT**: 网络地址转换协议，我们知道属于不同局域网的主机可以使用相同的IP地址，从而一定程度上缓解了IP资源枯竭的问题，然而主机在局域网中使用的IP地址是不能在公网中使用的，当局域网主机想要与公网主机进行通信时，NAT方法可以将该主机IP地址转换为全球IP地址。该协议能够有效解决IP地址不足的问题。
- **IPv6**: 作为接替IPv4的下一代互联网协议，其可以实现 2^{128} 个地址，而这个数量级，即使给地球上每一粒沙子都分配一个IP地址也够用，该协议能够从根本上解决IPv4地址不够用的问题。

52. 说下 ARP 协议的工作过程？

ARP 协议，**Address Resolution Protocol**，地址解析协议，它是用于实现 IP 地址到 MAC 地址的映射。



1. 首先，每台主机都会在自己的 ARP 缓冲区中建立一个 ARP 列表，以表示 IP 地址和 MAC 地址的对应关系。
2. 当源主机需要将一个数据包要发送到目的主机时，会首先检查自己的 ARP 列表，是否存在该 IP 地址对应的 MAC 地址；如果有，就直接将数据包发送到这个 MAC 地址；如果没有，就向本地网段发起一个 ARP 请求的广播包，查询此目的主机对应的 MAC 地址。此 ARP 请求的数据包里，包括源主机的 IP 地址、硬件地址、以及目的主机的 IP 地址。
3. 网络中所有的主机收到这个 ARP 请求后，会检查数据包中的目的 IP 是否和自己的 IP 地址一致。如果不相同，就会忽略此数据包；如果相同，该主机首先将发送端的 MAC 地址和 IP 地址添加到自己的 ARP 列表中，如果 ARP 表中已经存在该 IP 的信息，则将其覆盖，然后给源主机发送一个 ARP 响应数据包，告诉对方自己是它需要查找的 MAC 地址。
4. 源主机收到这个 ARP 响应数据包后，将得到的目的主机的 IP 地址和 MAC 地址添加到自己的 ARP 列表中，并利用此信息开始数据的传输。如果源主机一直没有收到 ARP 响应数据包，表示 ARP 查询失败。

53.为什么既有IP地址，又有MAC地址？

MAC地址和IP地址都有什么作用？

- MAC地址是数据链路层和物理层使用的地址，是写在网卡上的物理地址，用来定义网络设备的位置，不可变更。
- IP地址是网络层和以上各层使用的地址，是一种逻辑地址。IP地址用来区别网络上的计算机。

为什么有了MAC地址还需要IP地址？

如果我们只使用MAC地址进行寻址的话，我们需要路由器记住每个MAC地址属于哪个子网，不然一次路由器收到数据包都要满世界寻找目的MAC地址。而我们知道MAC地址的长度为48位，也就是最多共有 2^{48} 个MAC地址，这就意味着每个路由器需要256T的内存，显然是不现实的。

和MAC地址不同，IP地址是和地域相关的，在一个子网中的设备，我们给其分配的IP地址前缀都是一样的，这样路由器就能根据IP地址的前缀知道这个设备属于哪个子网，剩下的寻址就交给子网内部实现，从而大大减少了路由器所需要的内存。

为什么有了IP地址还需要MAC地址？

收件地址：IP地址

收件人：MAC地址



- 只有当设备连入网络时，才能根据他进入了哪个子网来为其分配IP地址，在设备还没有IP地址的时候，或者在分配IP的过程中。我们需要MAC地址来区分不同的设备。
- IP地址可以比作为地址，MAC地址为收件人，在一次通信过程中，两者是缺一不可的。

54.ICMP协议的功能？

ICMP（Internet Control Message Protocol），网际控制报文协议。

- ICMP协议是一种面向无连接的协议，用于传输出错报告控制信息。
- 它是一个非常重要的协议，它对于网络安全具有极其重要的意义。它属于网络层协议，主要用于在主机与路由器之间传递控制信息，包括报告错误、交换受限控制和状态信息等。
- 当遇到IP数据无法访问目标、IP路由器无法按当前的传输速率转发数据包等情况时，会自动发送ICMP消息。

比如我们日常使用得比较多的 **ping**，就是基于 ICMP 的。

55. 说下 ping 的原理？

ping，**Packet Internet Groper**，是一种因特网包探索器，用于测试网络连接量的程序。Ping 是工作在 TCP/IP 网络体系结构中应用层的一个服务命令，主要是向特定的目的主机发送 ICMP（Internet Control Message Protocol 因特网报文控制协议）请求报文，测试目的站是否可达及了解其有关状态。

```
λ ping www.baidu.com

正在 Ping www.a.shifen.com [110.242.68.4] 具有 32 字节的数据:
来自 110.242.68.4 的回复: 字节=32 时间=13ms TTL=53
来自 110.242.68.4 的回复: 字节=32 时间=14ms TTL=53
来自 110.242.68.4 的回复: 字节=32 时间=13ms TTL=53
来自 110.242.68.4 的回复: 字节=32 时间=13ms TTL=53

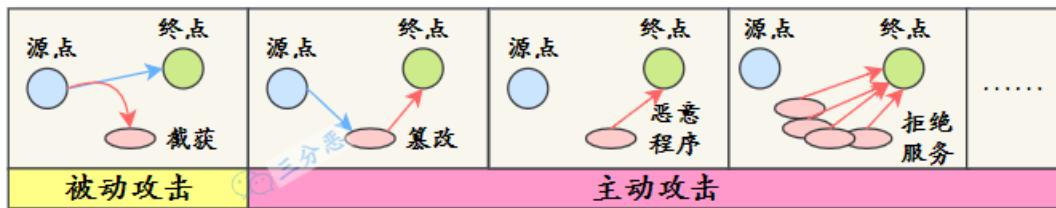
110.242.68.4 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 13ms, 最长 = 14ms, 平均 = 13ms
```

一般来说，ping 可以用来检测网络通不通。它是基于 **ICMP** 协议工作的。假设 **机器 A** ping **机器 B**，工作过程如下：

1. ping 通知系统，新建一个固定格式的 ICMP 请求数据包
2. ICMP 协议，将该数据包和目标机器 B 的 IP 地址打包，一起转交给 IP 协议层
3. IP 层协议将本机 IP 地址为源地址，机器 B 的 IP 地址为目标地址，加上一些其他的控制信息，构建一个 IP 数据包
4. 先获取目标机器 B 的 MAC 地址。
5. 数据链路层构建一个数据帧，目的地址是 IP 层传过来的 MAC 地址，源地址是本机的 MAC 地址
6. 机器 B 收到后，对比目标地址，和自己本机的 MAC 地址是否一致，符合就处理返回，不符合就丢弃。
7. 根据目的主机返回的 ICMP 回送回答报文中的时间戳，从而计算出往返时间
8. 最终显示结果有这几项：发送到目的主机的 IP 地址、发送 & 收到 & 丢失的分组数、往返时间的最小、最大 & 平均值

56.说说有哪些安全攻击？

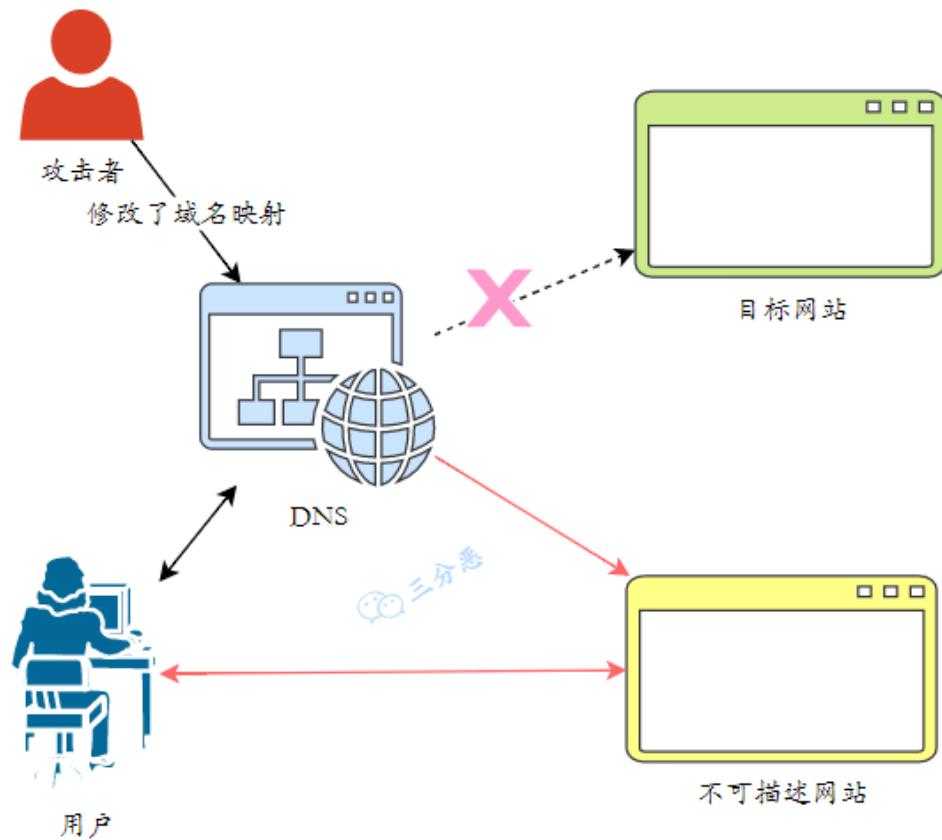
网络安全攻击主要分为两种类型，**被动攻击**和**主动攻击**：



- **被动攻击**：是指攻击者从网络上窃听他人的通信内容，通常把这类攻击称为截获，被动攻击主要有两种形式：消息内容泄露攻击和流量分析攻击。由于攻击者没有修改数据，使得这种攻击很难被检测到。
- **主动攻击**：直接对现有的数据和服务造成影响，常见的主动攻击类型有：
 - 篡改：攻击者故意篡改网络上送的报文，甚至把完全伪造的报文传送给接收方。
 - 恶意程序：恶意程序种类繁多，包括计算机病毒、计算机蠕虫、特洛伊木马、后门入侵、流氓软件等等。
 - 拒绝服务Dos：攻击者向服务器不停地发送分组，使服务器无法提供正常服务。

57.DNS劫持了解吗？

DNS劫持即域名劫持，是通过将原域名对应的IP地址进行替换，从而使用户访问到错误的网站，或者使用户无法正常访问网站的一种攻击方式。



域名劫持往往只能在特定的网络范围内进行，范围外的DNS服务器能够返回正常的IP地址。攻击者可以冒充原域名所属机构，通过电子邮件的方式修改组织机构的域名注册信息，或者将域名转让给其它主机，并将新的域名信息保存在所指定的DNS服务器中，从而使用户无法对原域名来进行解析以访问目标地址。

DNS劫持的步骤是什么样的？

1. 获取要劫持的域名信息：攻击者会首先访问域名查询要劫持的站点的域名信息。
2. 控制域名响应的E-Mail账号：在获取到域名信息后，攻击者通过暴力破解或者专门的方法破解公司注册域名时使用的E-mail账号所对应的密码，更高级的攻击者甚至能够直接对E-Mail进行信息窃取。
3. 修改注册信息：当攻击者破解了E-Mail后，会利用相关的更改功能修改该域名的注册信息，包括域名拥有者信息，DNS服务器信息等。
4. 使用E-Mail收发确认函：在修改完注册信息后，攻击者E-Mail在真正拥有者之前收到修改域名注册信息的相关确认信息，并回复确认修改文件，待网络公司恢复已成功修改信件后，攻击者便成功完成DNS劫持。

怎么应对DNS劫持？

- 直接通过IP地址访问网站，避开DNS劫持

- 由于域名劫持往往只能在特定的网络范围内进行，因此一些高级用户可以通过网络设置让DNS指向正常的域名服务器以实现对目标网址的正常访问，例如计算机首选DNS服务器的地址固定为8.8.8.8。

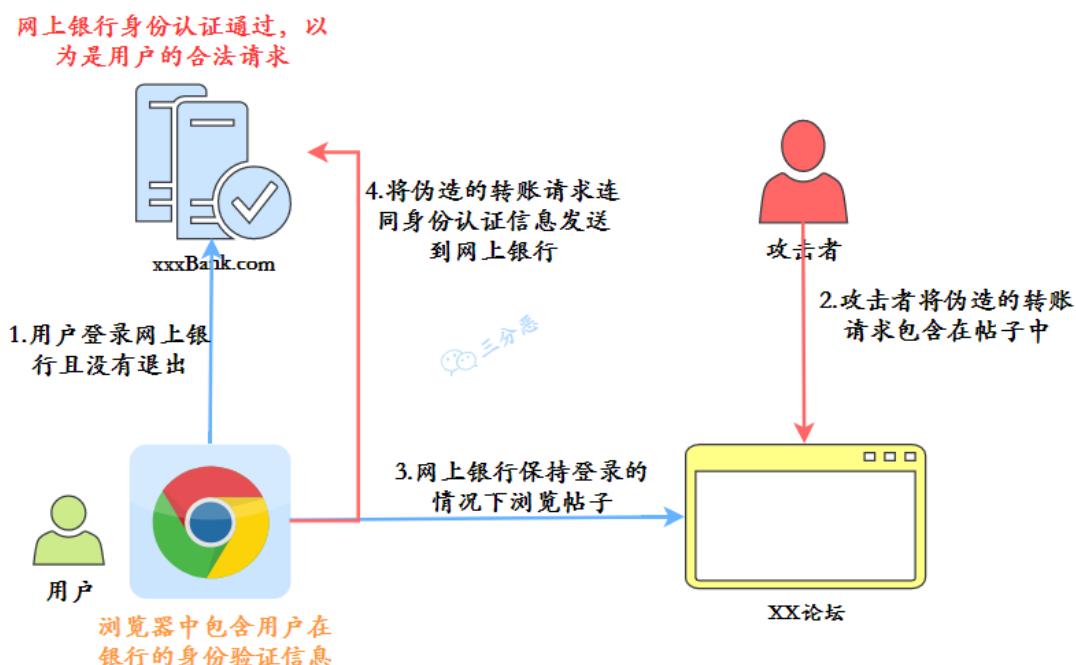
58.什么是 CSRF 攻击？如何避免？

什么是 CSRF 攻击？

CSRF，跨站请求伪造（英文全称是 Cross-site request forgery），是一种挟持用户在当前已登录的 Web 应用程序上执行非本意的操作的攻击方法。

CSRF 是如何攻击的呢？

来看一个例子：



1. 用户登陆银行，没有退出，浏览器包含了用户在银行的身份认证信息。
2. 攻击者将伪造的转账请求，包含在在帖子
3. 用户在银行网站保持登陆的情况下，浏览帖子
4. 将伪造的转账请求连同身份认证信息，发送到银行网站
5. 银行网站看到身份认证信息，以为就是用户的合法操作，最后造成用户资金损失。

怎么应对 CSRF 攻击呢？

- 检查 Referer 字段

HTTP头中的Referer字段记录了该 HTTP 请求的来源地址。在通常情况下，访问一个安全受限页面的请求来自于同一个网站，而如果黑客要对其实施 CSRF 攻击，他一般只能在他自己的网站构造请求。因此，可以通过验证Referer值来防御 CSRF 攻击。

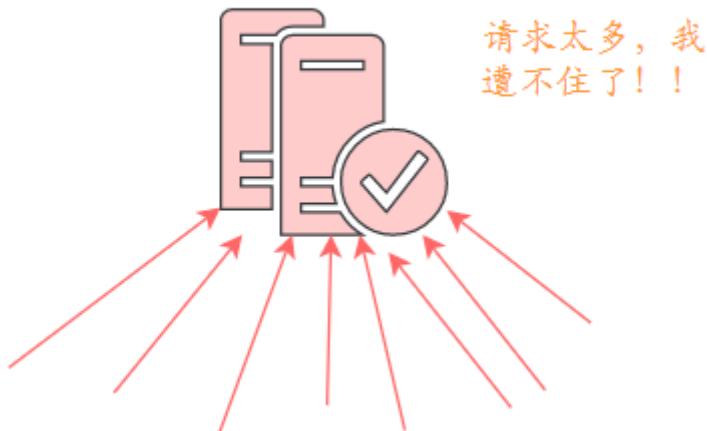
- 添加校验 token

以在 HTTP 请求中以参数的形式加入一个随机产生的 token，并在服务器端建立一个拦截器来验证这个 token，如果请求中没有token或者 token 内容不正确，则认为可能是 CSRF 攻击而拒绝该请求。

- 敏感操作多重校验

对一些敏感的操作，除了需要校验用户的认证信息，还可以通过邮箱确认、验证码确认这样方式多重校验。

59.什么是 DoS、DDoS、DRDoS 攻击？



- **DOS**: (Denial of Service), 翻译过来就是拒绝服务，一切能引起拒绝 行为的攻击都被称为 DOS 攻击。最常见的 DoS 攻击就有**计算机网络宽带攻击**、**连通性攻击**。
- **DDoS**: (Distributed Denial of Service)，翻译过来是分布式拒绝服务。是指处于不同位置的多个攻击者同时向一个或几个目标发动攻击，或者一个攻击者控制了位于不同位置的多台机器，并利用这些机器对受害者同时实施攻击。
主要形式有流量攻击和资源耗尽攻击，常见的 DDoS 攻击有：**SYN Flood**、**Ping of Death**、**ACK Flood**、**UDP Flood** 等。

- **DRDoS**: (Distributed Reflection Denial of Service), 中文是分布式反射拒绝服务，该方式靠的是发送大量带有被害者 IP 地址的数据包给攻击主机，然后攻击主机对 IP 地址源做出大量回应，从而形成拒绝服务攻击。

如何防范DDoS?

针对DDoS中的流量攻击，最直接的方法是增加带宽，理论上只要带宽大于攻击流量就可以了，但是这种方法成本非常高。在有充足带宽的前提下，我们应该尽量提升路由器、网卡、交换机等硬件设施的配置。

针对资源耗尽攻击，我们可以升级主机服务器硬件，在网络带宽得到保证的前提下，使得服务器能够有效对抗海量的SYN攻击包。我们也可以安装专业的抗DDoS防火墙，从而对抗SYN Flood等流量型攻击。瓷碗，负载均衡，CDN等技术都能有效对抗DDoS攻击。

60.什么是 XSS 攻击，如何避免？

XSS 攻击也是比较常见，XSS，叫**跨站脚本攻击（Cross-Site Scripting）**，因为会与层叠样式表 (Cascading Style Sheets, CSS) 的缩写混淆，因此有人将跨站脚本攻击缩写为 XSS。它指的是恶意攻击者往 Web 页面里插入恶意 html 代码，当用户浏览网页的时候，嵌入其中 Web 里面的 html 代码会被执行，从而达到恶意攻击用户的特殊目的。

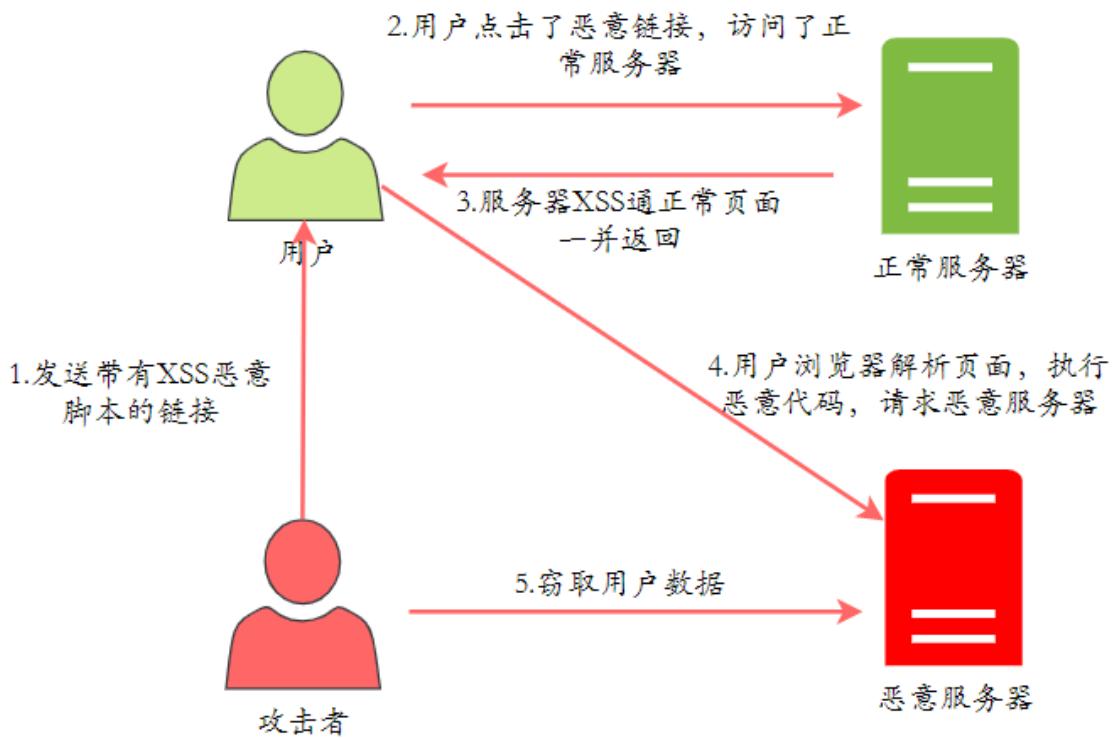
XSS 攻击一般分三种类型：**存储型、反射型、DOM 型 XSS**

XSS 是如何攻击的呢？

简单说，XSS的攻击方式就是想办法“教唆”用户的浏览器去执行一些这个网页中原本不存在的前端代码。

拿反射型举个例子吧，流程图如下：

1. 攻击者构造出特殊的 URL，其中包含恶意代码。
2. 用户打开带有恶意代码的 URL 时，访问正常网站服务器
3. 网站服务端将恶意代码从 URL 中取出，拼接在 HTML 中返回给浏览器。
4. 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行，请求恶意服务器，发送用户数据
5. 攻击者就可以窃取用户的数据，以此冒充用户的行为，调用目标网站接口执行攻击者指定的操作。



如何应对 XSS 攻击？

- 对输入进行过滤，过滤标签等，只允许合法值。
- HTML 转义
- 对于链接跳转，如 `` 等，要校验内容，禁止以 `script` 开头的非法链接。
- 限制输入长度

61. 对称加密与非对称加密有什么区别？

对称加密：指加密和解密使用同一密钥，优点是运算速度较快，缺点是如何安全将密钥传输给另一方。常见的对称加密算法有：DES、AES 等。



非对称加密: 指的是加密和解密使用不同的密钥（即公钥和私钥）。公钥与私钥是成对存在的，如果用公钥对数据进行加密，只有对应的私钥才能解密。常见的非对称加密算法有 RSA。



62.RSA和AES算法有什么区别？

- **RSA**

采用非对称加密的方式，采用公钥进行加密，私钥解密的形式。其私钥长度一般较长，由于需要大数的乘幂求模等运算，其运算速度较慢，不合适大量数据文件加密。

- **AES**

采用对称加密的方式，其秘钥长度最长只有256个比特，加密和解密速度较快，易于硬件实现。由于是对称加密，通信双方在进行数据传输前需要获知加密密钥。

参考：

[1]. [2W字！梳理50道经典计算机网络面试题（收藏版）](#)

[2]. 小林coding 《图解网络》

[3]. [“三次握手，四次挥手”这么讲，保证你忘不了](#)

[4]. 艾小仙 《我要进大厂》

[5]. 《图解HTTP》

[6]. [浅析DNS域名解析过程](#)

[7]. [「2021」高频前端面试题汇总之计算机网络篇](#)

[8]. [计算机网络](#)

- [9]. 谢希仁编著《计算机网络》
 - [10]. 《图解TCP/IP》
 - [11]. [TCP的可靠性传输是如何保证的](#)
 - [12]. [前端安全系列（一）：如何防止XSS攻击？](#)
-

关注公众号：三分恶

手册更新动态
即刻送达



添加个人微信：ThirdFighter

技术交流
加大佬云集微信群



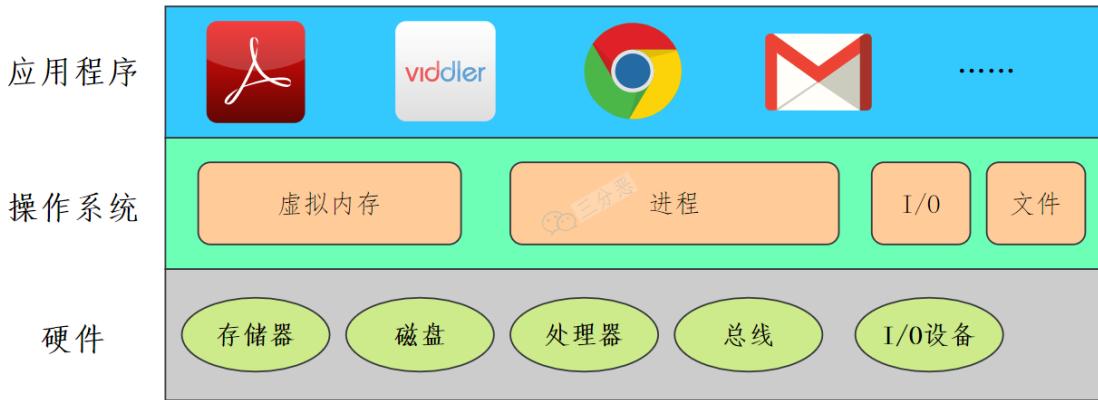
二、操作系统

引论

1.什么是操作系统？

可以这么说，操作系统是一种运行在内核态的软件。

它是应用程序和硬件之间的媒介，向应用程序提供硬件的抽象，以及管理硬件资源。

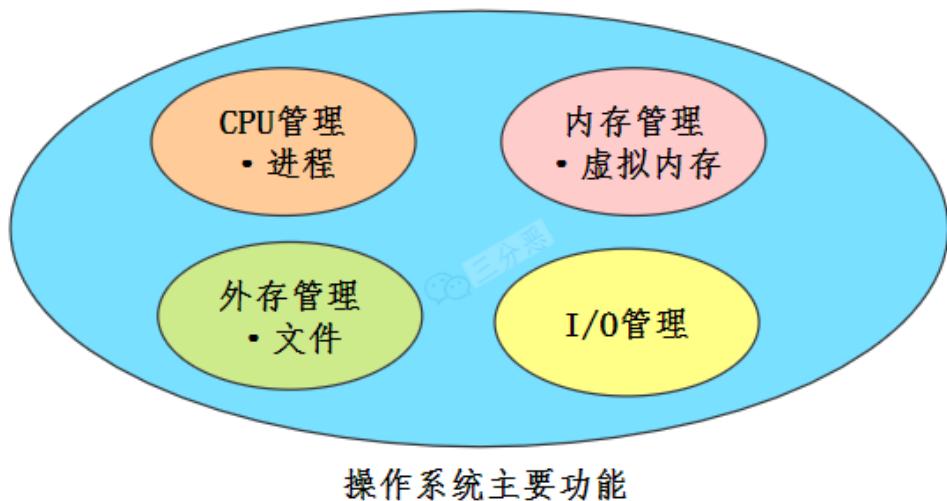


2. 系统主要有哪些功能？

操作系统最主要的功能：

- 处理器（CPU）管理：CPU的管理和分配，主要指的是进程管理。
- 内存管理：内存的分配和管理，主要利用了虚拟内存的方式。
- 外存管理：外存（磁盘等）的分配和管理，将外存以文件的形式提供出去。
- I/O管理：对输入/输出设备的统一管理。

除此之外，还有保证自身正常运行的健壮性管理，防止非法操作和入侵的安全性管理。



操作系统结构

4.什么是内核？

可以这么说，内核是一个计算机程序，它是操作系统的核，提供了操作系统最核心的能力，可以控制操作系统中所有的内容。

5.什么是用户态和内核态？

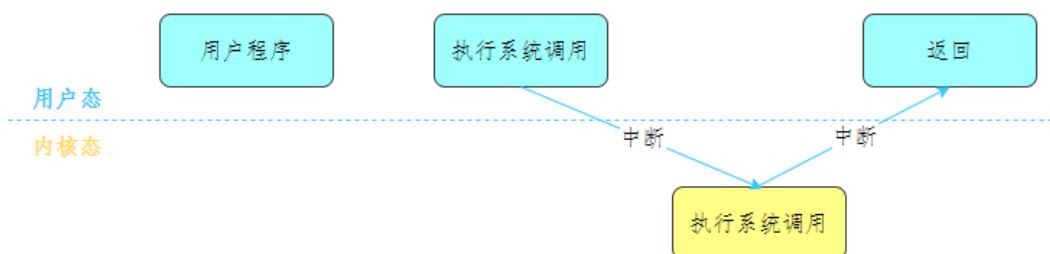
内核具有很高的权限，可以控制 CPU、内存、硬盘等硬件，出于权限控制的考虑，因此大多数操作系统，把内存分成了两个区域：

- 内核空间，这个内存空间只有内核程序可以访问；
- 用户空间，这个内存空间专门给应用程序使用，权限比较小；

用户空间的代码只能访问一个局部的内存空间，而内核空间的代码可以访问所有内存空间。因此，当程序使用用户空间时，我们常说该程序在**用户态**执行，而当程序使内核空间时，程序则在**内核态**执行。

6.用户态和内核态是如何切换的？

应用程序如果需要进入内核空间，就需要通过系统调用，来进入内核态：



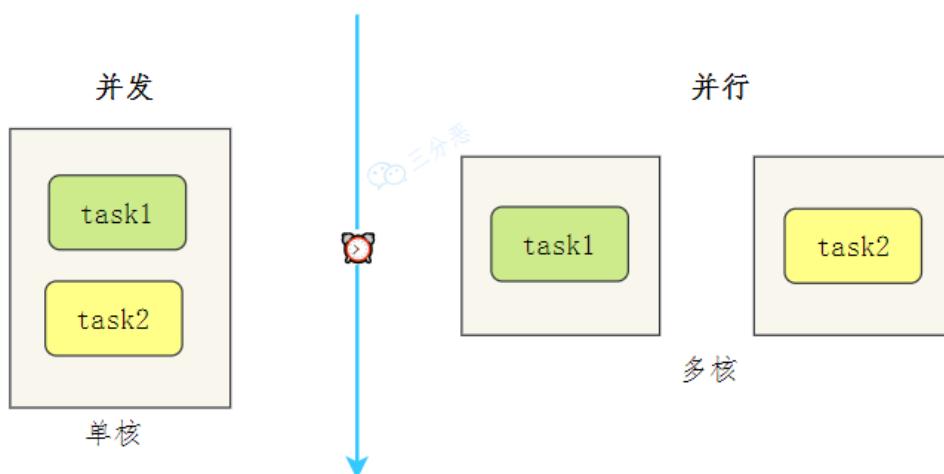
内核程序执行在内核态，用户程序执行在用户态。当应用程序使用系统调用时，会产生一个中断。发生中断后，CPU 会中断当前在执行的用户程序，转而跳转到中断处理程序，也就是开始执行内核程序。内核处理完后，主动触发中断，把 CPU 执行权限交回给用户程序，回到用户态继续工作。

进程和线程

7. 并行和并发有什么区别？

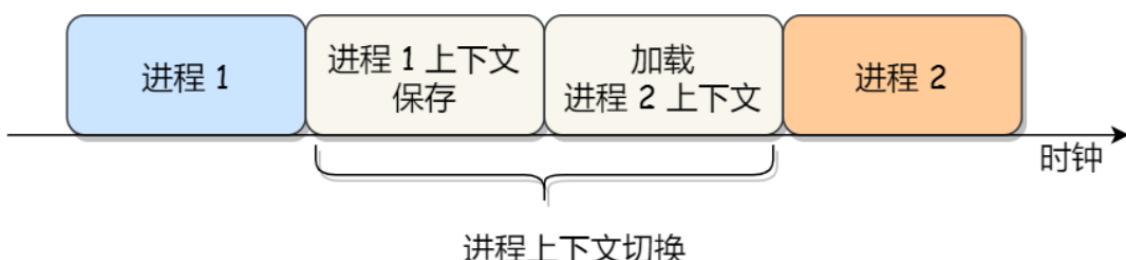
并发就是在一段时间内，多个任务都会被处理；但在某一时刻，只有一个任务在执行。单核处理器做到的并发，其实是利用时间片的轮转，例如有两个进程A和B，A运行一个时间片之后，切换到B，B运行一个时间片之后又切换到A。因为切换速度足够快，所以宏观上表现为在一段时间内能同时运行多个程序。

并行就是在同一时刻，有多个任务在执行。这个需要多核处理器才能完成，在微观上就能同时执行多条指令，不同的程序被放到不同的处理器上运行，这个是物理上的多个进程同时进行。



8. 什么是进程上下文切换？

对于单核单线程 CPU 而言，在某一时刻只能执行一条 CPU 指令。上下文切换 (Context Switch) 是一种将 CPU 资源从一个进程分配给另一个进程的机制。从用户角度看，计算机能够并行运行多个进程，这恰恰是操作系统通过快速上下文切换造成的结果。在切换的过程中，操作系统需要先存储当前进程的状态 (包括内存空间的指针，当前执行完的指令等等)，再读入下一个进程的状态，然后执行此进程。

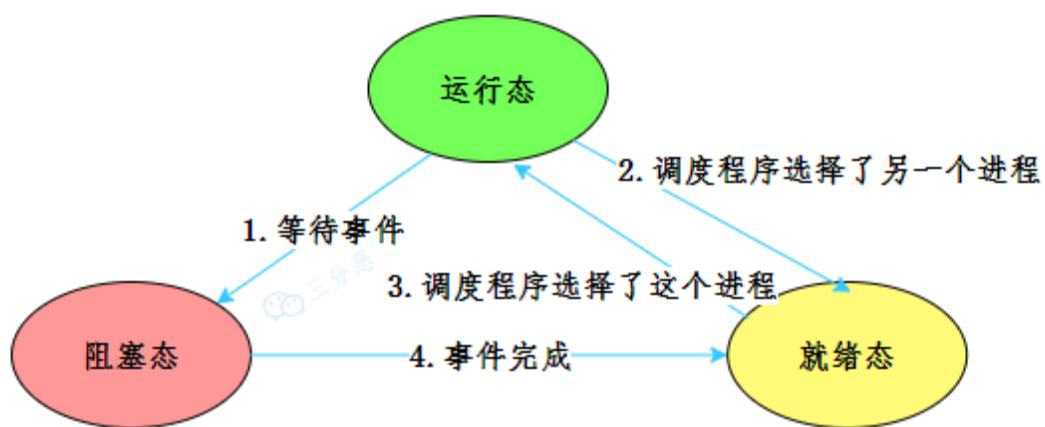


9. 进程有哪些状态？

当一个进程开始运行时，它可能会经历下面这几种状态：

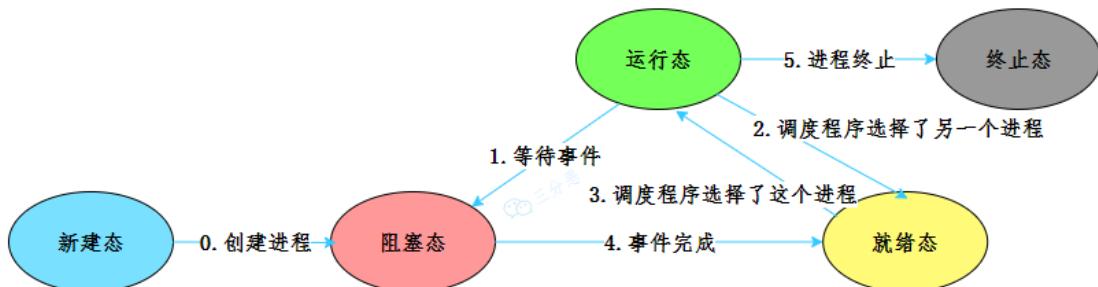
上图中各个状态的意义：

- 运行状态（Runing）：该时刻进程占用 CPU；
- 就绪状态（Ready）：可运行，由于其他进程处于运行状态而暂时停止运行；
- 阻塞状态（Blocked）：该进程正在等待某一事件发生（如等待输入/输出操作的完成）而暂时停止运行，这时，即使给它 CPU 控制权，它也无法运行；



当然，进程还有另外两个基本状态：

- 创建状态（new）：进程正在被创建时的状态；
- 结束状态（Exit）：进程正在从系统中消失时的状态；



10. 什么是僵尸进程？

僵尸进程是已完成且处于终止状态，但在进程表中却仍然存在的进程。

僵尸进程一般发生在有父子关系的进程中，一个子进程的进程描述符在子进程退出时不会释放，只有当父进程通过 `wait()` 或 `waitpid()` 获取了子进程信息后才会释放。如果子进程退出，而父进程并没有调用 `wait()` 或 `waitpid()`，那么子进程的进程描述符仍然保存在系统中。

11. 什么1是孤儿进程？

一个父进程退出，而它的一个或多个子进程还在运行，那么这些子进程将成为孤儿进程。孤儿进程将被 `init` 进程（进程 ID 为 1 的进程）所收养，并由 `init` 进程对它们完成状态收集工作。因为孤儿进程会被 `init` 进程收养，所以孤儿进程不会对系统造成危害。

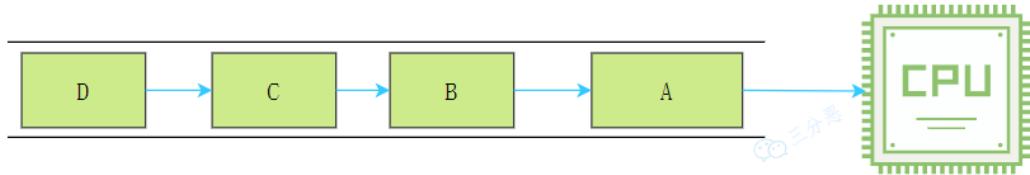
12. 进程有哪些调度算法？

进程调度就是确定某一个时刻 CPU 运行哪个进程，常见的进程调度算法有：



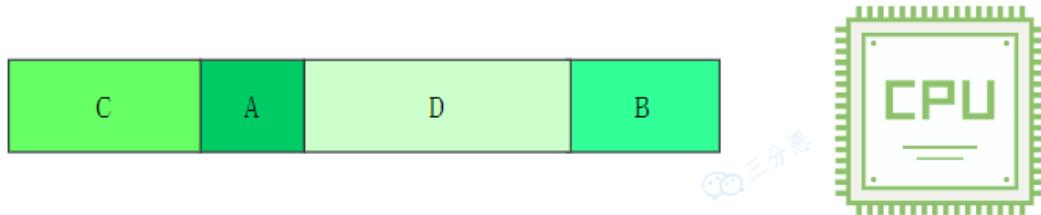
- 先来先服务

非抢占式的调度算法，按照请求的顺序进行调度。有利于长作业，但不利于短作业，因为短作业必须一直等待前面的长作业执行完毕才能执行，而长作业又需要执行很长时间，造成了短作业等待时间过长。另外，对I/O密集型进程也不利，因为这种进程每次进行I/O操作之后又得重新排队。



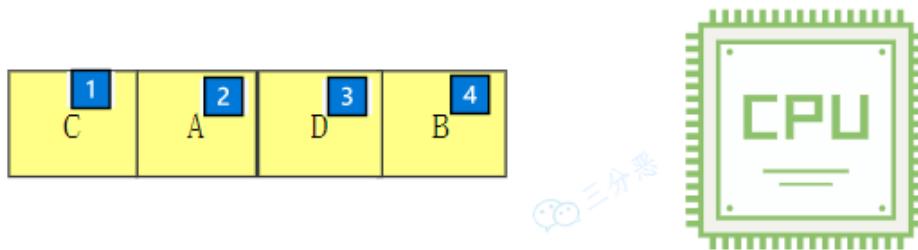
- 短作业优先

非抢占式的调度算法，按估计运行时间最短的顺序进行调度。长作业有可能会饿死，处于一直等待短作业执行完毕的状态。因为如果一直有短作业到来，那么长作业永远得不到调度。



- 优先级调度

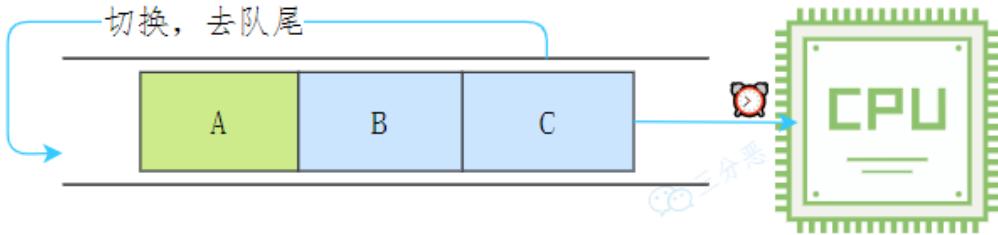
为每个进程分配一个优先级，按优先级进行调度。为了防止低优先级的进程永远等不到调度，可以随着时间的推移增加等待进程的优先级。



- 时间片轮转

将所有就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU时间分配给队首进程，该进程可以执行一个时间片。当时间片用完时，由计时器发出时钟中断，调度程序便停止该进程的执行，并将它送往就绪队列的末尾，同时继续把CPU时间分配给队首的进程。

时间片轮转算法的效率和时间片的大小有很大关系：因为进程切换都要保存进程的信息并且载入新进程的信息，如果时间片太小，会导致进程切换得太频繁，在进程切换上就会花过多时间。而如果时间片过长，那么实时性就不能得到保证。



- 最短剩余时间优先

最短作业优先的抢占式版本，按剩余运行时间的顺序进行调度。当一个新的作业到达时，其整个运行时间与当前进程的剩余时间作比较。如果新的进程需要的时间更少，则挂起当前进程，运行新的进程。否则新的进程等待。

13. 进程间通信有哪些方



- 管道：管道可以理解成不同进程之间的对白，一方发声，一方接收，声音的介质可是是空气或者电缆，进程之间就可以通过管道，**所谓的管道就是内核中的一串缓存**，从管道的一端写入数据，就是缓存在了内核里，另一端读取，也是从内核中读取这段数据。

管道可以分为两类：匿名管道和命名管道。匿名管道是单向的，只能在有亲缘关系的进程间通信；命名管道是双向的，可以实现本机任意两个进程通信。

"奉先我儿" 就可以理解成一种命名管道，董卓叫亲儿子，直接叫“我儿”就可以了



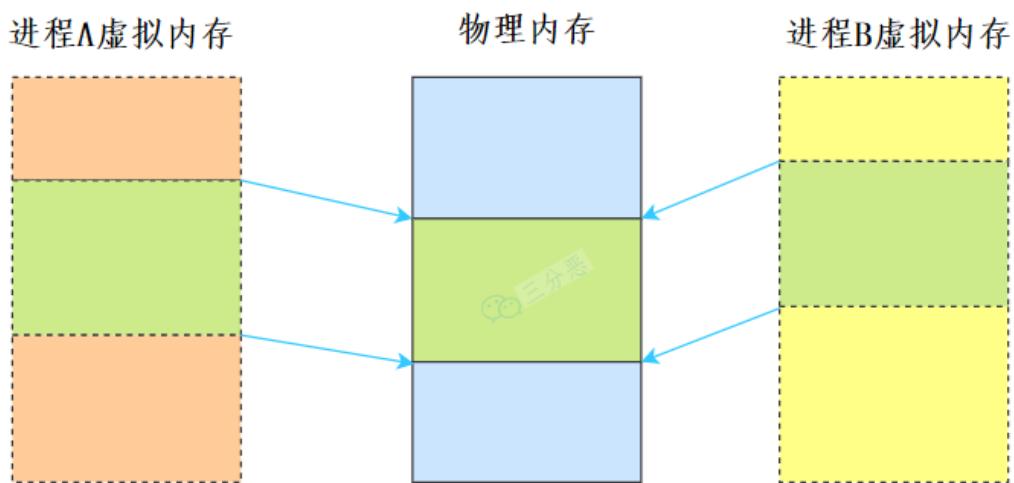
- 信号：信号可以理解成一种电报，发送方发送内容，指定接收进程，然后发出特定的软件中断，操作系统接到中断请求后，找到接收进程，通知接收进程处理信号。

比如 `kill -9 1050` 就表示给PID为1050的进程发送 `SIGKIL` 信号。Linux系统中常用信号：

- (1) `SIGHUP`: 用户从终端注销，所有已启动进程都将收到该进程。系统缺省状态下对该信号的处理是终止进程。
 - (2) `SIGINT`: 程序终止信号。程序运行过程中，按Ctrl+C键将产生该信号。
 - (3) `SIGQUIT`: 程序退出信号。程序运行过程中，按Ctrl+\键将产生该信号。
 - (4) `SIGBUS`和`SIGSEGV`: 进程访问非法地址。
 - (5) `SIGFPE`: 运算中出现致命错误，如除零操作、数据溢出等。
 - (6) `SIGKILL`: 用户终止进程执行信号。shell下执行`kill -9`发送该信号。
 - (7) `SIGTERM`: 结束进程信号。shell下执行`kill`进程pid发送该信号。
 - (8) `SIGALRM`: 定时器信号。
 - (9) `SIGCLD`: 子进程退出信号。如果其父进程没有忽略该信号也没有处理该信号，则子进程退出后将形成僵尸进程。
- 消息队列：消息队列就是保存在内核中的消息链表，包括Posix消息队列和System V消息队列。有足够的权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。



- 共享内存：共享内存的机制，就是拿出一块虚拟地址空间来，映射到相同的物理内存中。这样这个进程写入的东西，另外的进程马上就能看到。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。

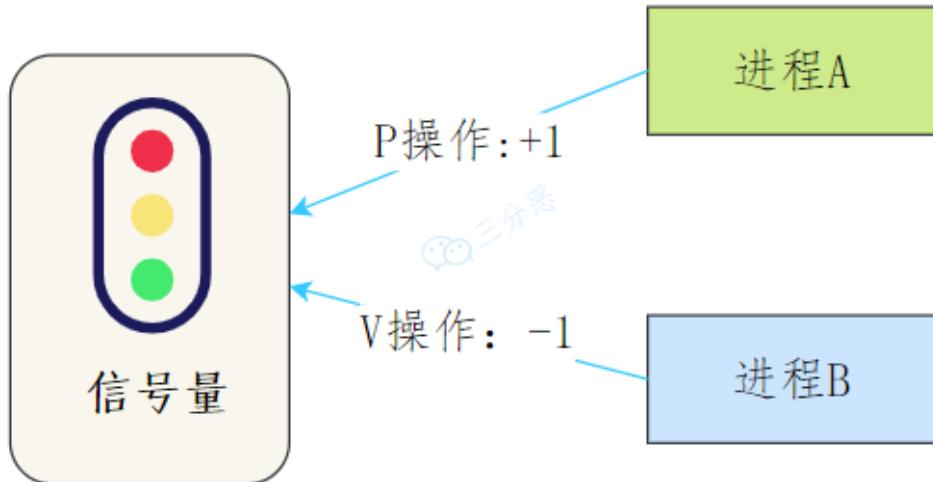


- 信号量：信号量我们可以理解成红绿灯，红灯行，绿灯停。**它本质上是一个整数计数器**，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

信号量表示资源的数量，控制信号量的方式有两种原子操作：

- 一个是 P 操作，这个操作会把信号量减去 1，相减后如果信号量 < 0 ，则表明资源已被占用，进程需阻塞等待；相减后如果信号量 ≥ 0 ，则表明还有资源可使用，进程可正常继续执行。
- 另一个是 V 操作，这个操作会把信号量加上 1，相加后如果信号量 ≤ 0 ，则表明当前有阻塞中的进程，于是会将该进程唤醒运行；相加后如果信号量 > 0 ，则表明当前没有阻塞中的进程；

P 操作是用在进入共享资源之前，V 操作是用在离开共享资源之后，这两个操作是必须成对出现的。



- **Socket:** 与其他通信机制不同的是，它可用于不同机器间的进程通信。

优缺点：

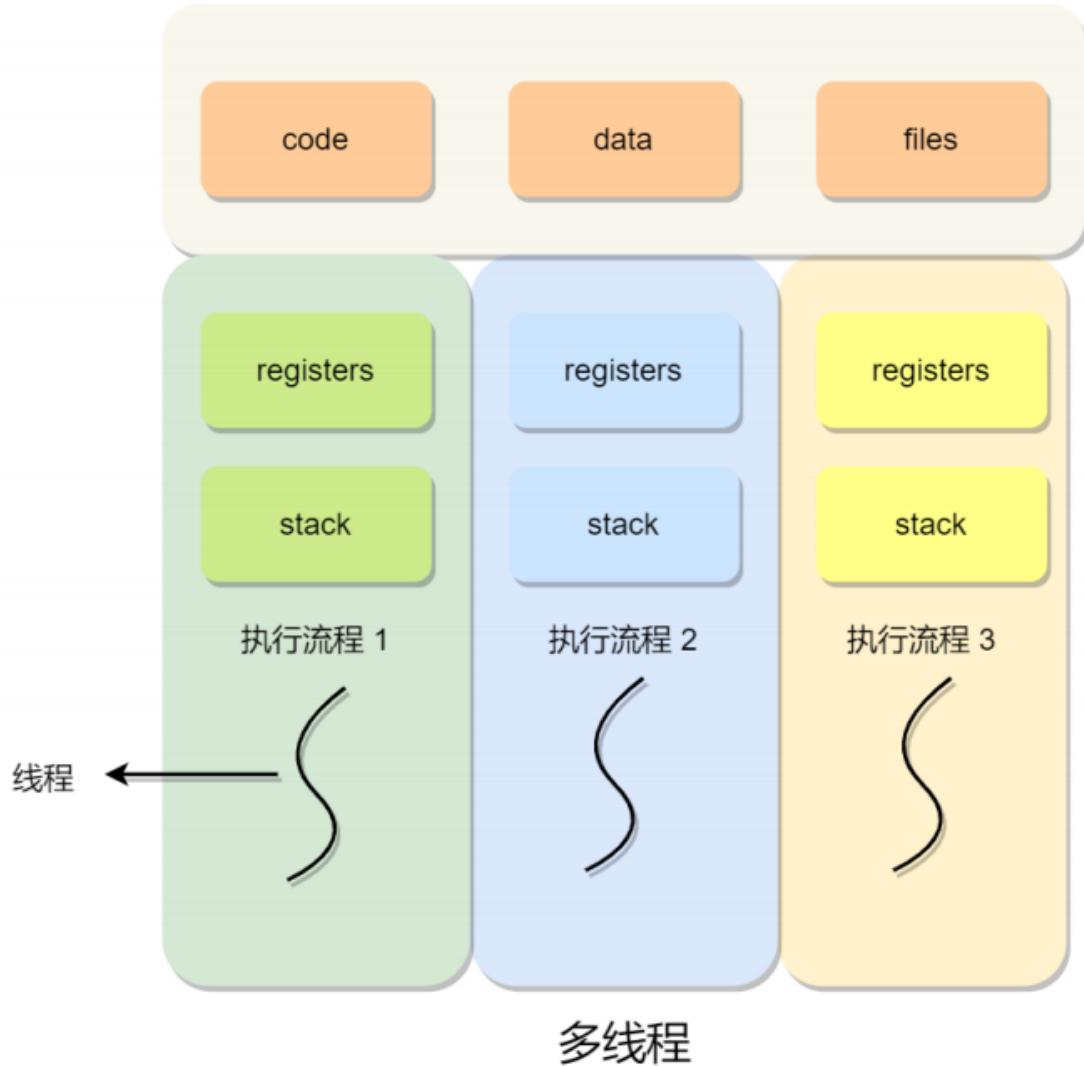
- 管道：简单；效率低，容量有限；
- 消息队列：不及时，写入和读取需要用户态、内核态拷贝。
- 共享内存区：能够很容易控制容量，速度快，但需要注意不同进程的同步问题。
- 信号量：不能传递复杂消息，一般用来实现进程间的同步；
- 信号：它是进程间通信的唯一异步机制。
- **Socket:** 用于不同主机进程间的通信。

14. 进程和线程的联系和区别？

线程和进程的联系：

线程是进程当中的一条执行流程。

同一个进程内多个线程之间可以共享代码段、数据段、打开的文件等资源，但每个线程各自都有一套独立的寄存器和栈，这样可以确保线程的控制流是相对独立的。



多线程

线程与进程的比较如下：

- 调度：进程是资源（包括内存、打开的文件等）分配的单位，线程是CPU调度的单位；
- 资源：进程拥有一个完整的资源平台，而线程只独享必不可少的资源，如寄存器和栈；
- 拥有资源：线程同样具有就绪、阻塞、执行三种基本状态，同样具有状态之间的转换关系；
- 系统开销：线程能减少并发执行的时间和空间开销——创建或撤销进程时，系统都要为之分配或回收系统资源，如内存空间，I/O设备等，OS所付出的开销显著大于在创建或撤销线程时的开销，进程切换的开销也远大于线程切换的开销。

15.线程上下文切换了解吗？

这还得看线程是不是属于同一个进程：

- 当两个线程不是属于同一个进程，则切换的过程就跟进程上下文切换一样；

- 当两个线程是属于同一个进程，因为虚拟内存是共享的，所以在切换时，虚拟内存这些资源就保持不动，只需要切换线程的私有数据、寄存器等不共享的数据；

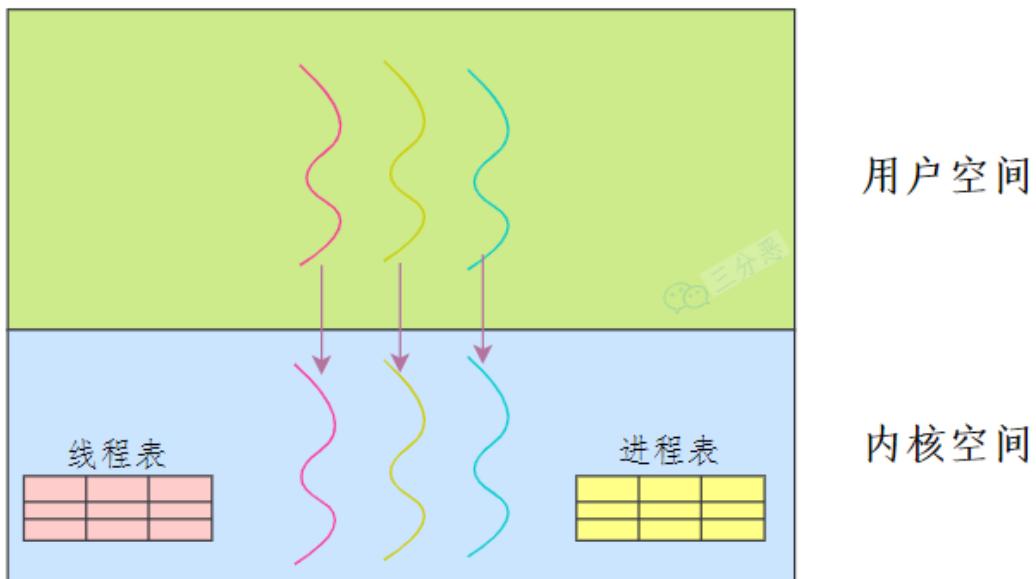
所以，线程的上下文切换相比进程，开销要小很多。

16.线程有哪些实现方式？

主要有三种线程的实现方式：

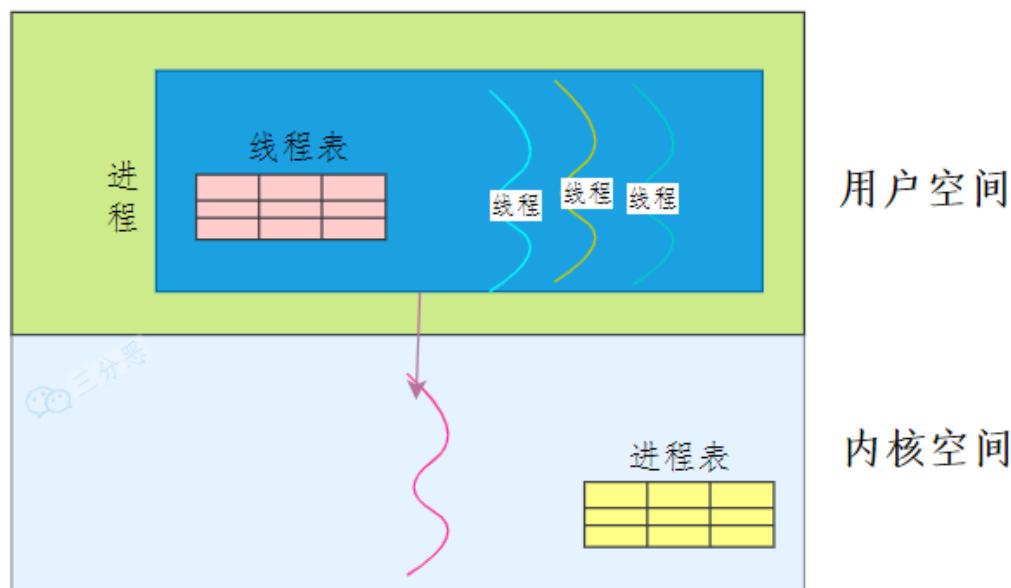
- 内核态线程实现：在内核空间实现的线程，由内核直接管理直接管理线程。

每个用户线程
就是一个内核线程

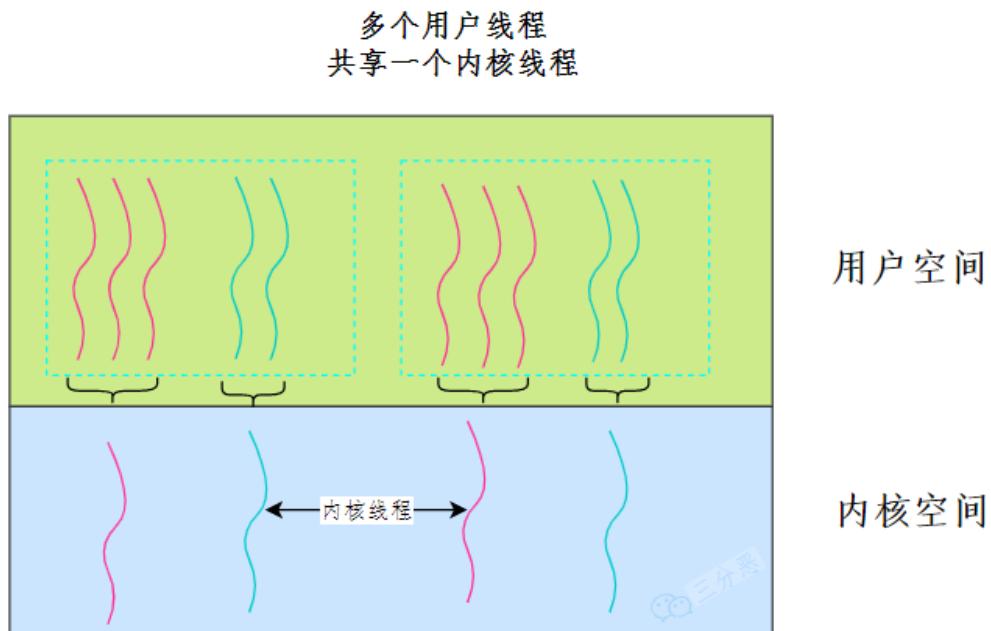


- 用户态线程实现：在用户空间实现线程，不需要内核的参与，内核对线程无感知。

操作系统只看到线程



- 混合线程实现：现代操作系统基本都是将两种方式结合起来使用。用户态的执行系统负责进程内部线程在非阻塞时的切换；内核态的操作系统负责阻塞线程的切换。即我们同时实现内核态和用户态线程管理。其中内核态线程数量较少，而用户态线程数量较多。每个内核态线程可以服务一个或多个用户态线程。

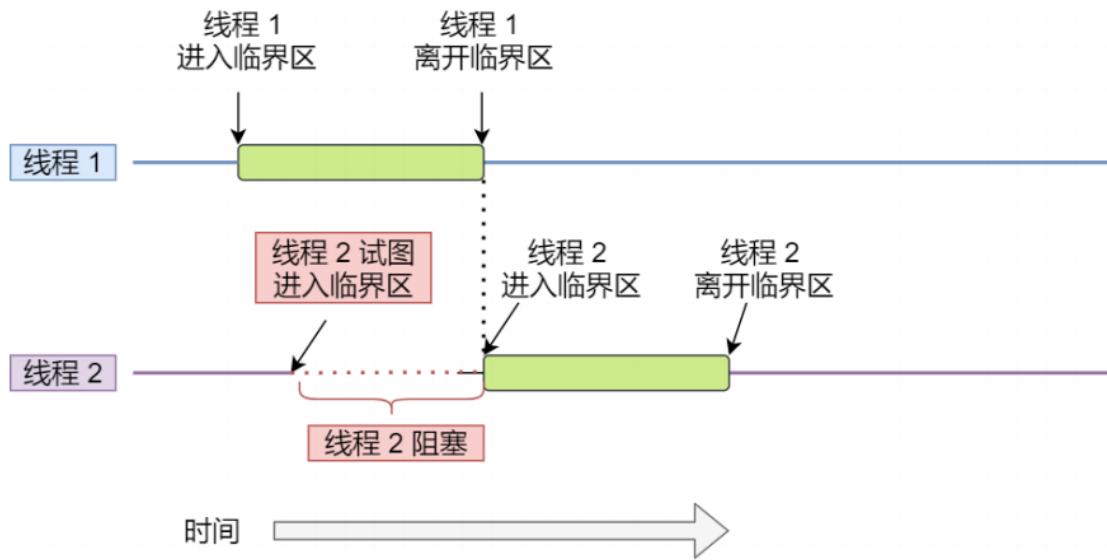


17.线程间如何同步？

同步解决的多线程操作共享资源的问题，目的是不管线程之间的执行如何穿插，最后的结果都是正确的。

我们前面知道线程和进程的关系：线程是进程当中的一条执行流程。所以说下面的一些同步机制不止针对线程，同样也可以针对进程。

临界区：我们把对共享资源访问的程序片段称为 **临界区**，我们希望这段代码是 **互斥** 的，保证在某时刻只能被一个线程执行，也就是说一个线程在临界区执行时，其它线程应该被阻止进入临界区。



临界区不仅针对线程，同样针对进程。

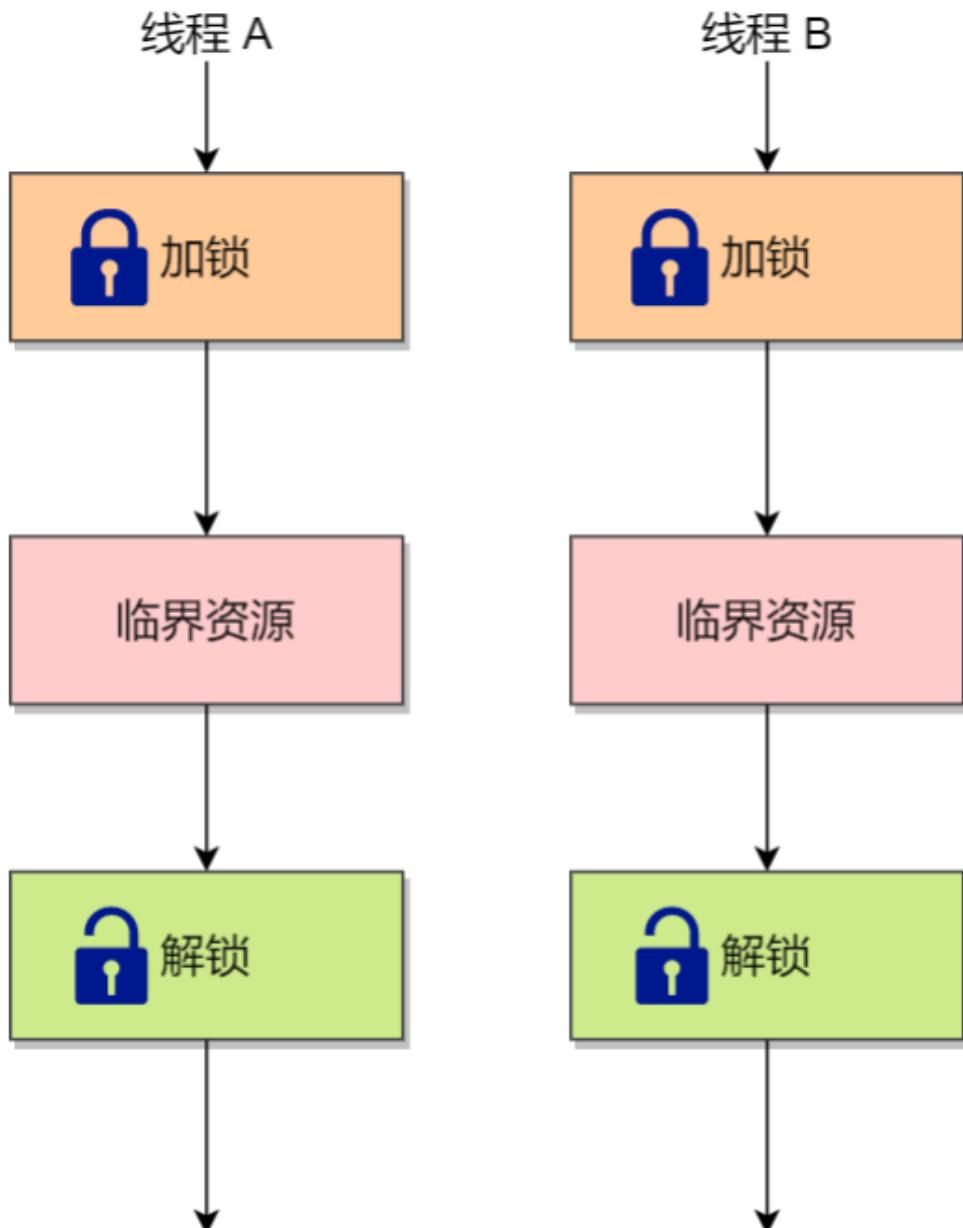
临界区同步的一些实现方式：

1、锁

使用加锁操作和解锁操作可以解决并发线程/进程的互斥问题。

任何想进入临界区的线程，必须先执行加锁操作。若加锁操作顺利通过，则线程可进入临界区；在完成对临界资源的访问后再执行解锁操作，以释放该临界资源。

加锁和解锁锁住的是什么呢？可以是 **临界区对象**，也可以只是一个简单的 **互斥量**，例如互斥量是 **0** 无锁，**1** 表示加锁。



根据锁的实现不同，可以分为 **忙等待锁** 和 **无忙等待锁**。

忙等待锁 就是加锁失败的线程，会不断尝试获取锁，也被称为自旋锁，它会一直占用CPU。

无忙等待锁 就是加锁失败的线程，会进入阻塞状态，放弃CPU，等待被调度。

2、信号量

信号量是操作系统提供的一种协调共享资源访问的方法。

通常 **信号量表示资源的数量**，对应的变量是一个整型（sem）变量。

另外，还有**两个原子操作的系统调用函数来控制信号量的**，分别是：

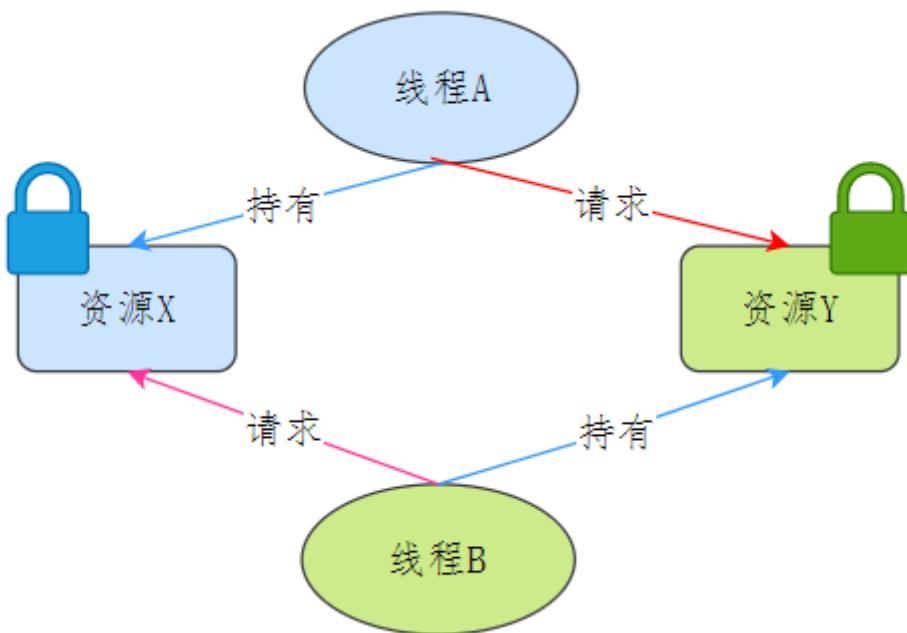
- P 操作：将 sem 减 1，相减后，如果 $sem < 0$ ，则进程/线程进入阻塞等待，否则继续，表明 P 操作可能会阻塞；

- V 操作：将 sem 加 1，相加后，如果 $sem \leq 0$ ，唤醒一个等待中的进程/线程，表明 V 操作不会阻塞；

P 操作是用在进入临界区之前，V 操作是用在离开临界区之后，这两个操作是必须成对出现的。

18.什么是死锁？

在两个或者多个并发线程中，如果每个线程持有某种资源，而又等待其它线程释放它或它们现在保持着的资源，在未改变这种状态之前都不能向前推进，称这一组线程产生了死锁。通俗的讲就是两个或多个线程无限期的阻塞、相互等待的一种状态。



19.死锁产生有哪些条件？

死锁产生需要 **同时** 满足四个条件：

- 互斥条件：指线程对已经获取到的资源进行独占使用，即该资源同时只由一个线程占用。如果此时还有其它线程请求获取该资源，则请求者只能等待，直至占有资源的线程释放该资源。
- 请求并持有条件：指一个线程已经持有了至少一个资源，但又提出了新的资源请求，而新资源已被其它线程占有，所以当前线程会被阻塞，但阻塞的同时并不释放自己已经获取的资源。
- 不可剥夺条件：指线程获取到的资源在自己使用完之前不能被其它线程抢占，只有在自己使用完毕后才由自己释放该资源。

- 环路等待条件：指在发生死锁时，必然存在一个线程——资源的环形链，即线程集合 {T0, T1, T2, ……, Tn} 中 T0 正在等待一 T1 占用的资源，T1 正在等待 T2 用的资源，…… Tn 在等待已被 T0 占用的资源。

20. 如何避免死锁呢？

产生死锁的有四个必要条件：互斥条件、持有并等待条件、不可剥夺条件、环路等待条件。

避免死锁，破坏其中的一个就可以。

消除互斥条件

这个是没法实现，因为很多资源就是只能被一个线程占用，例如锁。

消除请求并持有条件

消除这个条件的办法很简单，就是一个线程一次请求其所需的所有资源。

消除不可剥夺条件

占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源，这样不可剥夺这个条件就破坏掉了。

消除环路等待条件

可以靠按序申请资源来预防。所谓按序申请，是指资源是有线性顺序的，申请的时候可以先申请资源序号小的，再申请资源序号大的，这样线性化后就不存在环路了。

21. 活锁和饥饿锁了解吗？

饥饿锁：

饥饿锁，这个饥饿指的是资源饥饿，某个线程一直等不到它所需要的资源，从而无法向前推进，就像一个人因为饥饿无法成长。

活锁：

在活锁状态下，处于活锁线程组里的线程状态可以改变，但是整个活锁组的线程无法推进。

活锁可以用两个人过一条很窄的小桥来比喻：为了让对方先过，两个人都往旁边让，但两个人总是让到同一边。这样，虽然两个人的状态一直在变化，但却都无法往前推进。

内存管理

22.什么是虚拟内存？

我们实际的物理内存主要是主存，但是物理主存空间有限，所以一般现代操作系统都会想办法把一部分内存块放到磁盘中，用到的时候再装入主存，但是对用户程序而言，是不需要注意实际的物理内存的，为什么呢？因为有**虚拟内存**的机制。

简单说，**虚拟内存**是操作系统提供的一种机制，将不同进程的虚拟地址和不同内存的物理地址映射起来。

每个进程都有自己独立的地址空间，再由操作系统映射到到实际的物理内存。

于是，这里就引出了两种地址的概念：

程序所使用的内存地址叫做**虚拟内存地址**（Virtual Memory Address）

实际存在硬件里面的空间地址叫**物理内存地址**（Physical Memory Address）。



23.什么是内存分段？

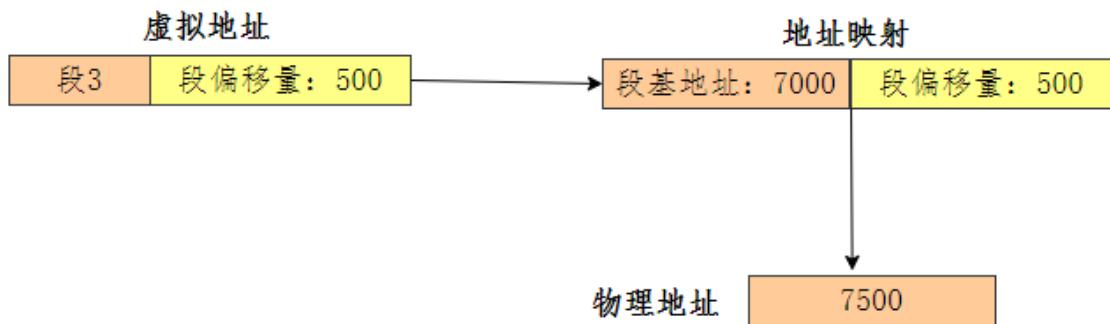
程序是由若干个逻辑分段组成的，如可由代码分段、数据分段、栈段、堆段组成。不同的段是有不同的属性的，所以就用分段（Segmentation）的形式把这些段分离出来。

分段机制下的虚拟地址由两部分组成，**段号**和**段内偏移量**。

虚拟地址和物理地址通过段表映射，段表主要包括**段号**、**段的界限**。



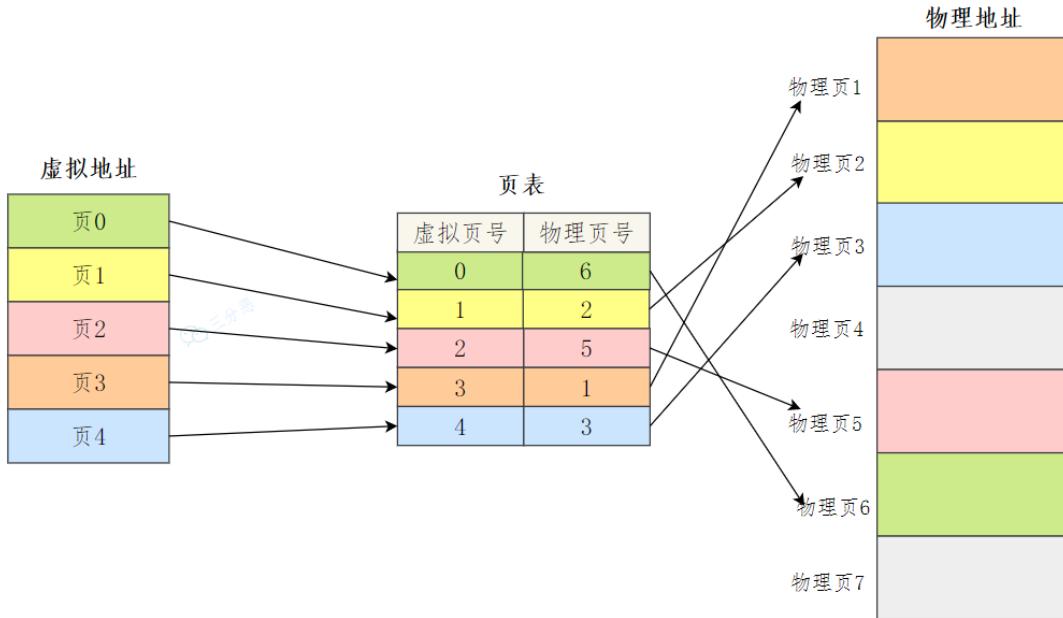
我们来看一个映射，虚拟地址：段3、段偏移量500 ----> 段基址7000+段偏移量500 ----> 物理地址：7500。



24.什么是内存分页？

分页是把整个虚拟和物理内存空间切成一段段固定尺寸的大小。这样一个连续并且尺寸固定的内存空间，我们叫页（Page）。在 Linux 下，每一页的大小为 4KB。

访问分页系统中内存数据需要两次的内存访问：一次是从内存中访问页表，从中找到指定的物理页号，加上页内偏移得到实际物理地址，第二次就是根据第一次得到的物理地址访问内存取出数据。

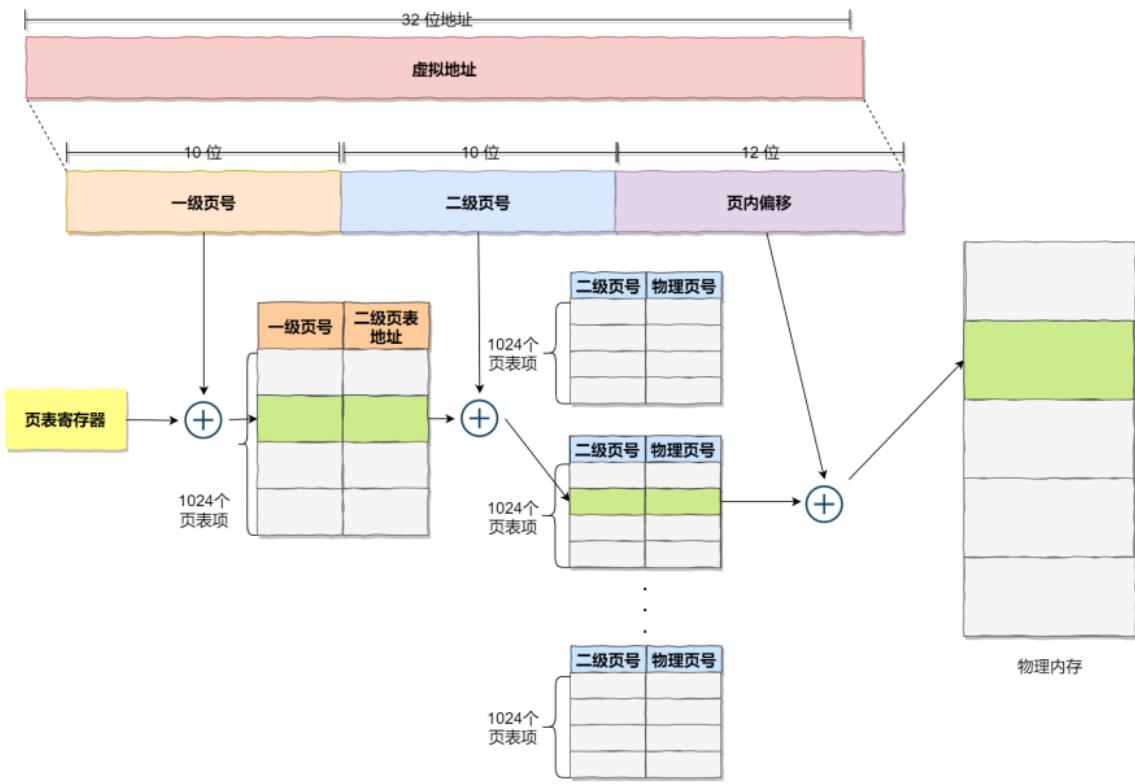


25.多级页表知道吗？

操作系统可能会有非常多进程，如果只是使用简单分页，可能导致的后果就是页表变得非常庞大。

所以，引入了多级页表的解决方案。

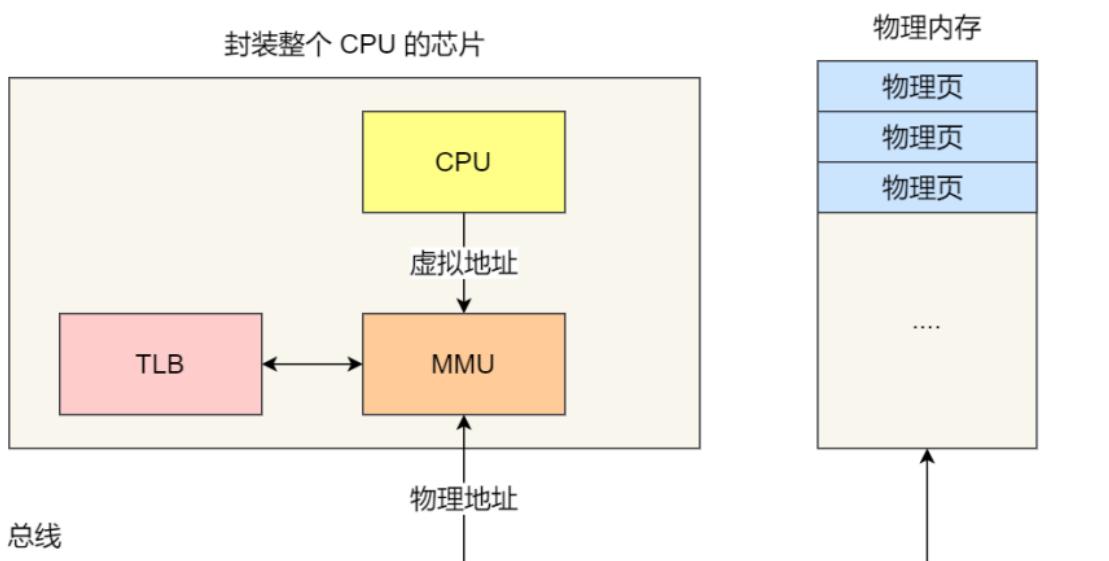
所谓的多级页表，就是把我们原来的单级页表再次分页，这里利用了**局部性原理**，除了顶级页表，其它级别的页表一来可以在需要的时候才被创建，二来内存紧张的时候还可以被置换到磁盘中。



26. 什么是块表？

同样利用了 **局部性原理**，即在一段时间内，整个程序的执行仅限于程序中的某一部分。相应地，执行所访问的存储空间也局限于某个内存区域。

利用这一特性，把最常访问的几个页表项存储到访问速度更快的硬件，于是计算机科学家们，就在 CPU 芯片中，加入了一个专门存放程序最常访问的页表项的 Cache，这个 Cache 就是 TLB（Translation Lookaside Buffer），通常称为页表缓存、转址旁路缓存、快表等。



27.分页和分段有什么区别？

- 段是信息的逻辑单位，它是根据用户的需要划分的，因此段对用户是可见的；
页是信息的物理单位，是为了管理主存的方便而划分的，对用户是透明的。
- 段的大小不固定，有它所完成的功能决定；页的大小固定，由系统决定
- 段向用户提供二维地址空间；页向用户提供的是一维地址空间
- 段是信息的逻辑单位，便于存储保护和信息的共享，页的保护和共享受到限制。

28.什么是交换空间？

操作系统把物理内存(Physical RAM)分成一块一块的小内存，每一块内存被称为页(page)。当内存资源不足时，Linux把某些页的内容转移至磁盘上的一块空间上，以释放内存空间。磁盘上的那块空间叫做交换空间(swap space)，而这一过程被称为交换(swapping)。物理内存和交换空间的总容量就是虚拟内存的可用容量。

用途：

- 物理内存不足时一些不常用的页可以被交换出去，腾给系统。
- 程序启动时很多内存页被用来初始化，之后便不再需要，可以交换出去。

29.页面置换算法有哪些？

在分页系统里，一个虚拟的页面可能在主存里，也可能在磁盘中，如果CPU发现虚拟地址对应的物理页不在主存里，就会产生一个缺页中断，然后从磁盘中把该页调入主存中。

如果内存里没有空间，就需要从主存里选择一个页面来置换。

常见的页面置换算法：



- 最佳页面置换算法 (OPT)

最佳页面置换算法是一个理想的算法，基本思路是，**置换在未来最长时间不访问的页面**。

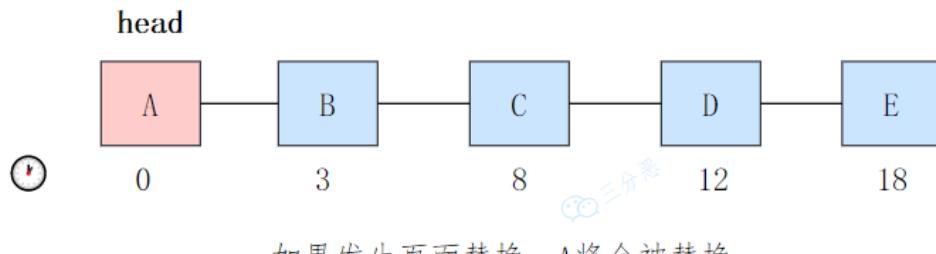
所以，该算法实现需要计算内存中每个逻辑页面的下一次访问时间，然后比较，选择未来最长时间不访问的页面。

但这个算法是无法实现的，因为当缺页中断发生时，操作系统无法知道各个页面下一次将在什么时候被访问。

- 先进先出置换算法 (FIFO)

既然我们无法预知页面在下一次访问前所需的等待时间，那可以**选择在内存驻留时间很长的页面进行置换**，这个就是「先进先出置换」算法的思想。

FIFO的实现机制是使用链表将所有在内存的页面按照进入时间的早晚链接起来，然后每次置换链表头上的页面就行了，新加进来的页面则挂在链表的末端。

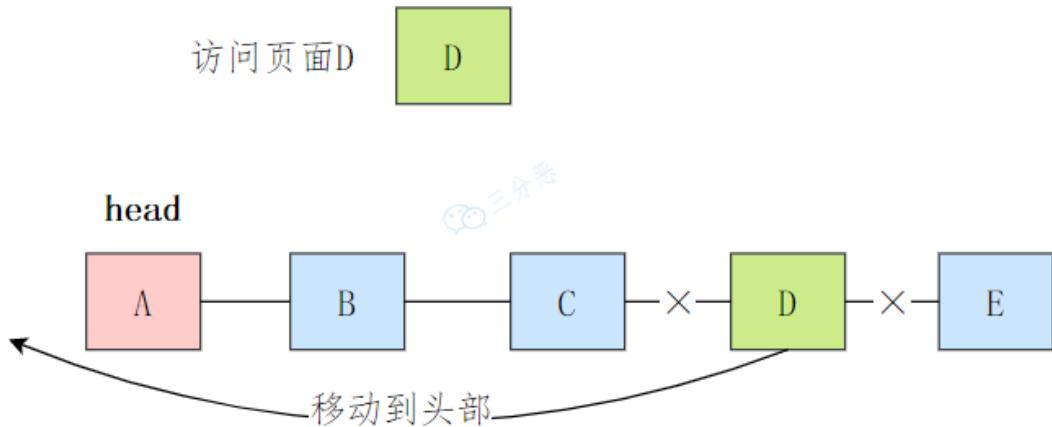


- 最近最久未使用的置换算法 (LRU)

最近最久未使用（**LRU**）的置换算法的基本思路是，发生缺页时，**选择最长时间没有被访问的页面进行置换**，也就是说，该算法假设已经很久没有使用的页面很有可能在未来较长的一段时间内仍然不会被使用。

这种算法近似最优置换算法，最优置换算法是通过「未来」的使用情况来推测要淘汰的页面，而 LRU 则是通过**历史**的使用情况来推测要淘汰的页面。

LRU 在理论上是可以实现的，但代价很高。为了完全实现 LRU，需要在内存中维护一个所有页面的链表，最近最多使用的页面在表头，最近最少使用的页面在表尾。



困难的是，在每次访问内存时都必须要更新整个链表。在链表中找到一个页面，删除它，然后把它移动到表头是一个非常费时的操作。

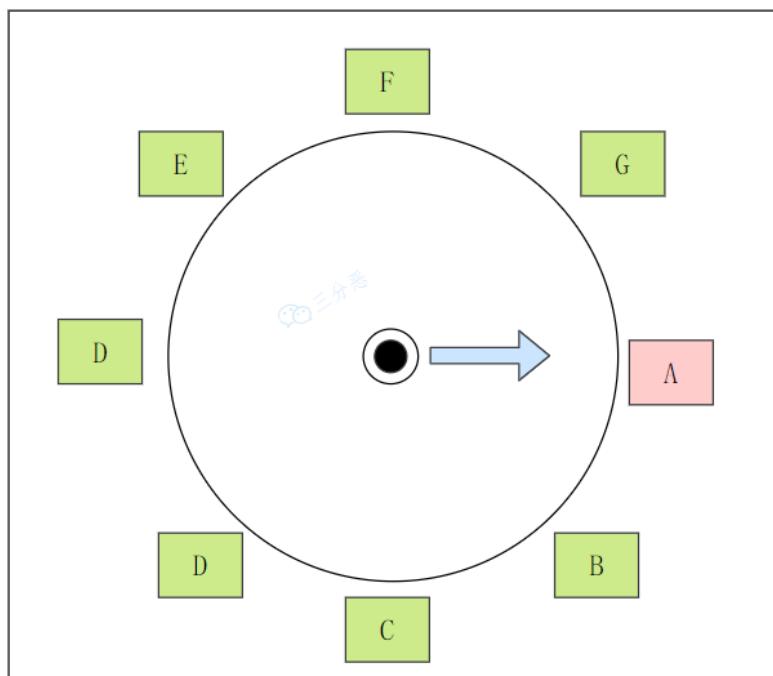
所以，LRU 虽然看上去不错，但是由于开销比较大，实际应用中比较少使用。

- 时钟页面置换算法

这个算法的思路是，把所有的页面都保存在一个类似钟面的环形链表中，一个表针指向最老的页面。

链表节点

页号	访问位	物理页号
----	-----	------



当发生缺页中断时，算法首先检查表针指向的页面：

如果它的访问位位是 0 就淘汰该页面，并把新的页面插入这个位置，然后把表针前移一个位置；

如果访问位是 1 就清除访问位，并把表针前移一个位置，重复这个过程直到找到了一个访问位为 0 的页面为止；

- 最不常用置换算法

最不常用算法（LFU），当发生缺页中断时，选择访问次数最少的那个页面，将其置换。

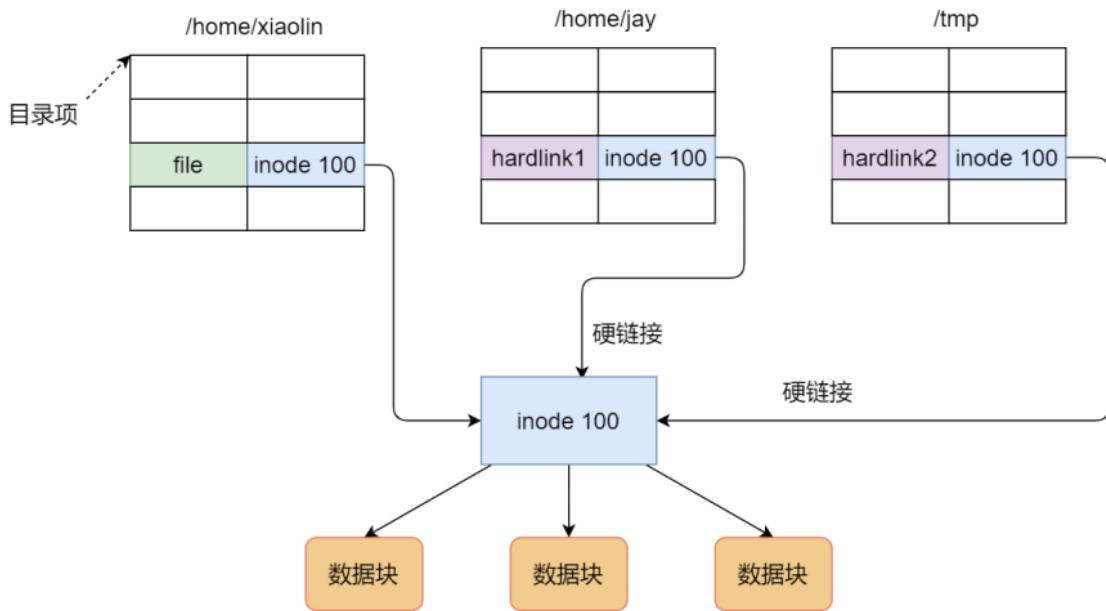
它的实现方式是，对每个页面设置一个「访问计数器」，每当一个页面被访问时，该页面的访问计数器就累加 1。在发生缺页中断时，淘汰计数器值最小的那个页面。

文件

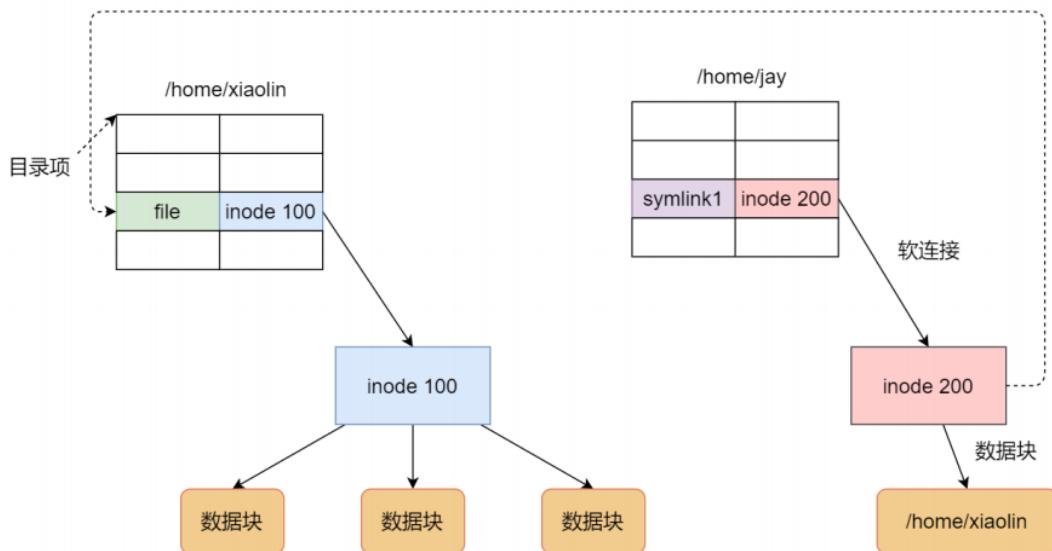
30.硬链接和软链接有什么区别？

- 硬链接就是在目录下创建一个条目，记录着文件名与 inode 编号，这个 inode 就是源文件的 inode。删除任意一个条目，文件还是存在，只要引用数量不为 0。但

是硬链接有限制，它不能跨越文件系统，也不能对目录进行链接。



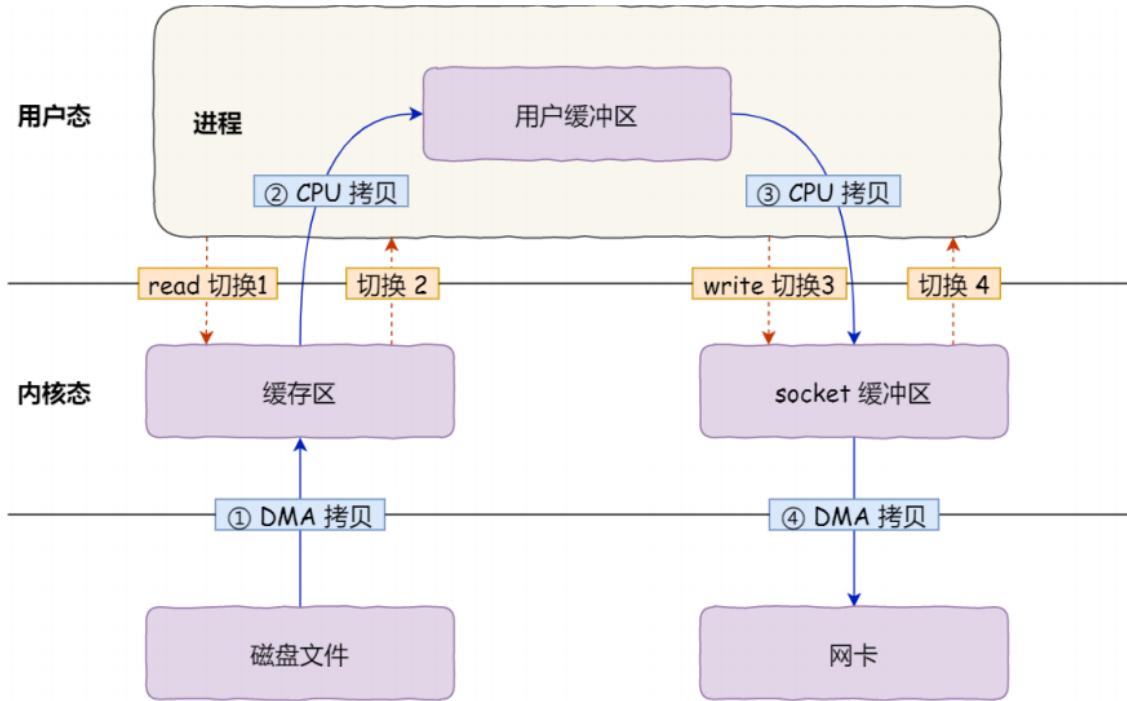
- 软链接相当于重新创建一个文件，这个文件有**独立的 inode**，但是这个**文件的内容是另外一个文件的路径**，所以访问软链接的时候，实际上相当于访问到了另外一个文件，所以软链接是可以跨文件系统的，甚至**目标文件被删除了，链接文件还是在的，只不过打不开指向的文件了而已。**



IO

31. 零拷贝了解吗？

假如需要文件传输，使用传统I/O，数据读取和写入是用户空间到内核空间来回赋值，而内核空间的数据是通过操作系统的I/O接口从磁盘读取或者写入，这期间发生了多次用户态和内核态的上下文切换，以及多次数据拷贝。

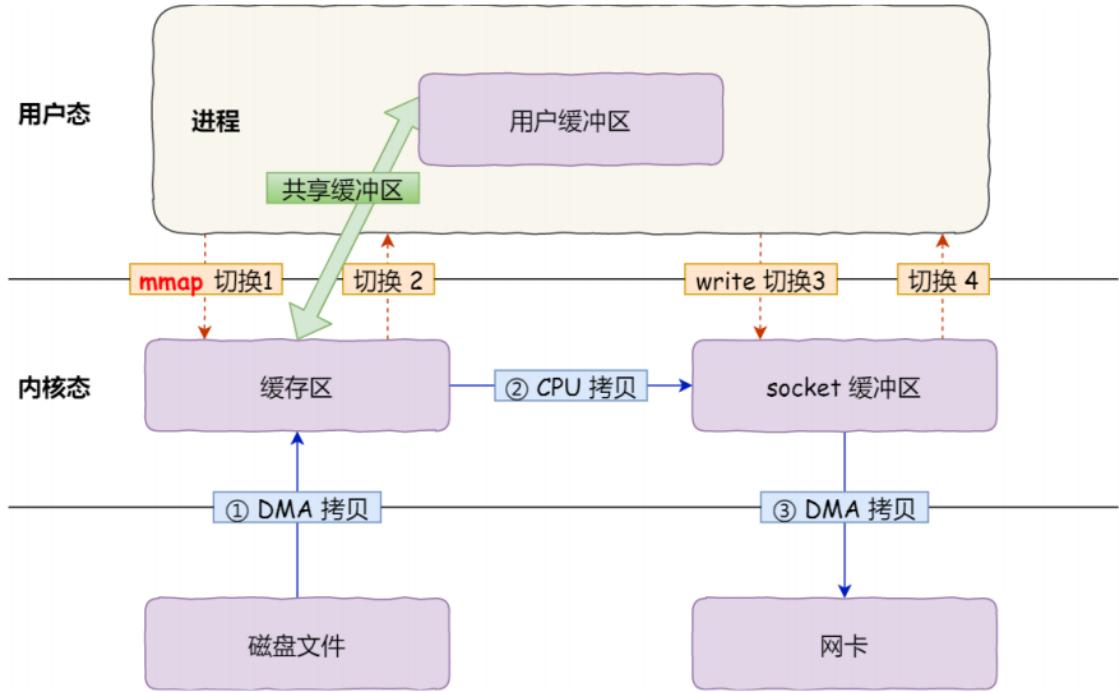


为了提升I/O性能，就需要减少用户态与内核态的上下文切换和内存拷贝的次数。

这就用到了我们零拷贝的技术，零拷贝技术实现主要有两种：

- mmap + write

mmap() 系统调用函数会直接把内核缓冲区里的数据「映射」到用户空间，这样，操作系统内核与用户空间就不需要再进行任何的数据拷贝操作。

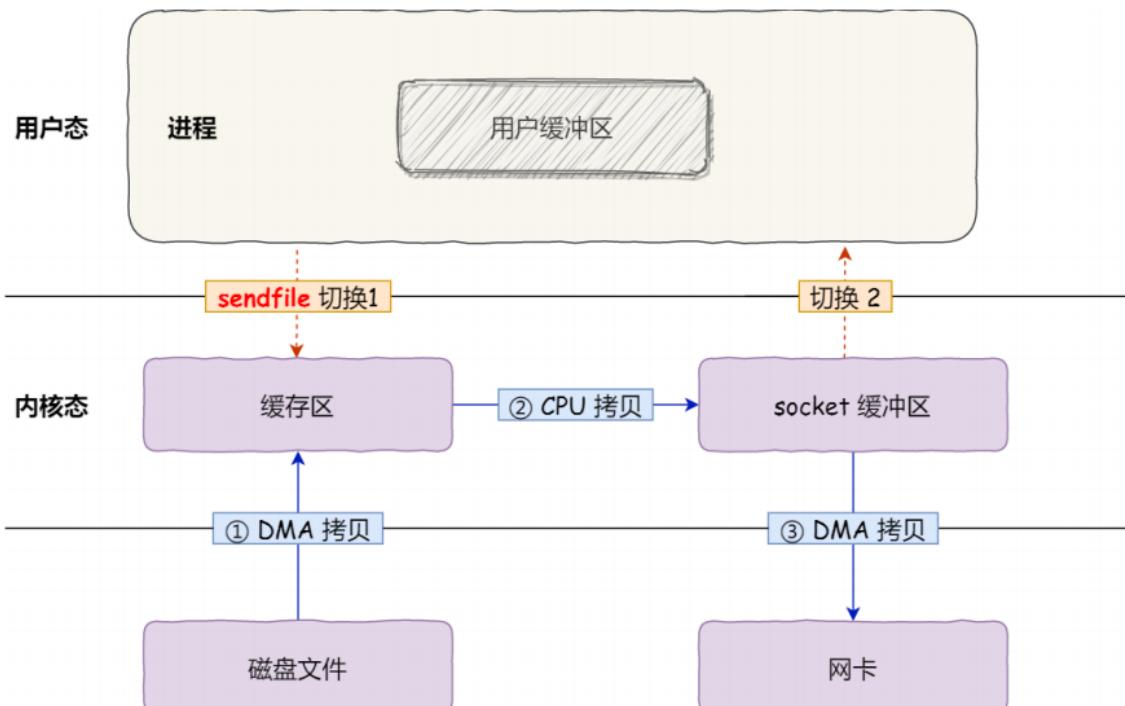


- `sendfile`

在 Linux 内核版本 2.1 中，提供了一个专门发送文件的系统调用函数 `sendfile()`。

首先，它可以替代前面的 `read()` 和 `write()` 这两个系统调用，这样就可以减少一次系统调用，也就减少了 2 次上下文切换的开销。

其次，该系统调用，可以直接把内核缓冲区里的数据拷贝到 socket 缓冲区里，不再拷贝到用户态，这样就只有 2 次上下文切换，和 3 次数据拷贝。



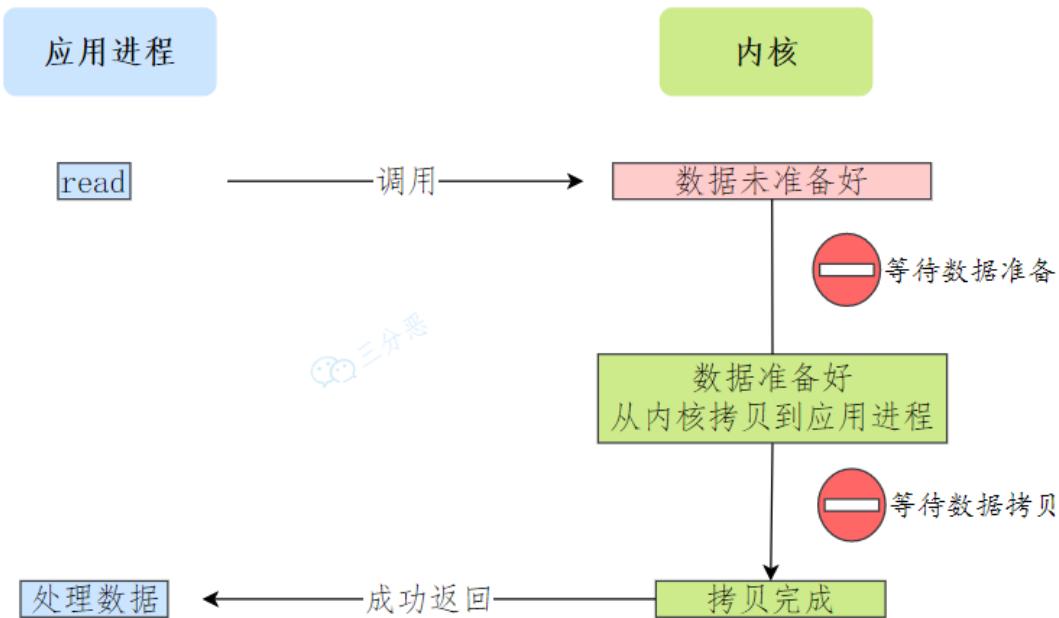
很多开源项目如Kafka、RocketMQ都采用了零拷贝技术来提升IO效率。

32. 聊聊阻塞与非阻塞 I/O、同步与异步 I/O？

- 阻塞I/O

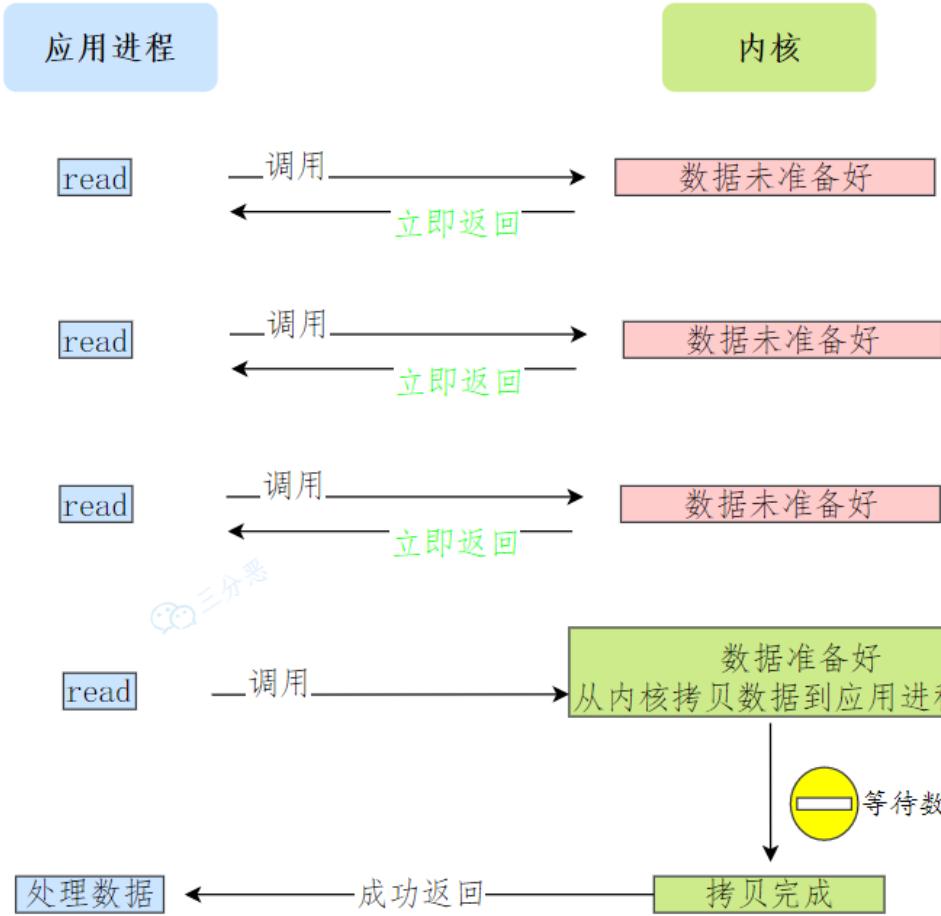
先来看看**阻塞 I/O**，当用户程序执行 `read`，线程会被阻塞，一直等到内核数据准备好，并把数据从内核缓冲区拷贝到应用程序的缓冲区中，当拷贝过程完成，`read` 才会返回。

注意，**阻塞等待的是 内核数据准备好 和 数据从内核态拷贝到用户态 这两个过程**。



- 非阻塞I/O

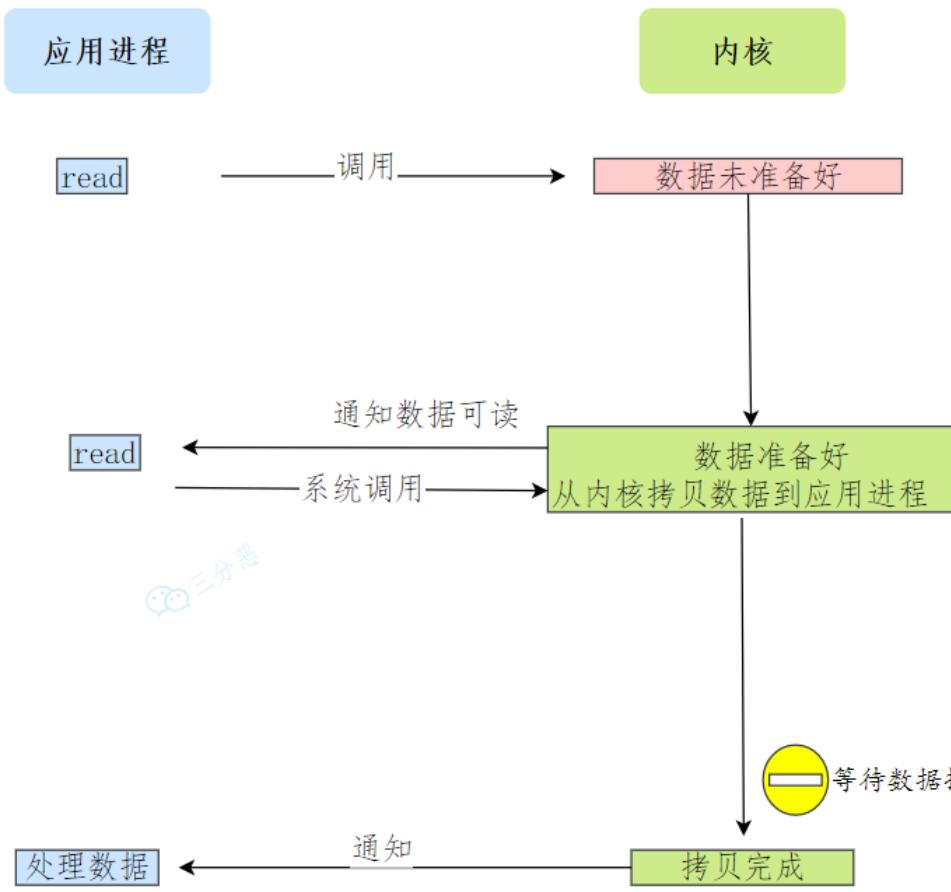
非阻塞的 `read` 请求在数据未准备好的情况下立即返回，可以继续往下执行，此时应用程序不断轮询内核，直到数据准备好，内核将数据拷贝到应用程序缓冲区，`read` 调用才可以获取到结果。



- 基于非阻塞的I/O多路复用

我们上面的非阻塞I/O有一个问题，什么问题呢？应用程序要一直轮询，这个过程没法干其它事情，所以引入了**I/O 多路复用**技术。

当内核数据准备好时，以事件通知应用程序进行操作。

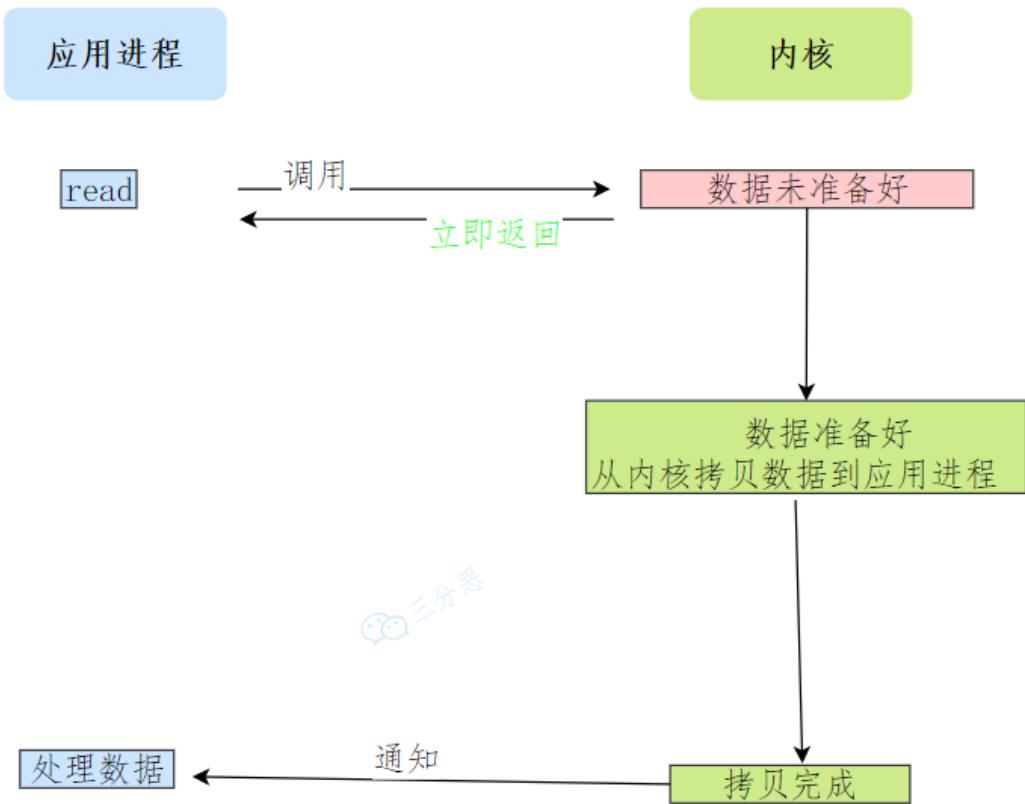


注意：无论是阻塞 I/O、还是非阻塞 I/O、非阻塞I/O多路复用，都是同步调用。因为它们在read调用时，内核将数据从内核空间拷贝到应用程序空间，过程都是需要等待的，也就是说这个过程是**同步**的，如果内核实现的拷贝效率不高，read调用就会在这个同步过程中等待比较长的时间。

- 异步I/O

真正的**异步 I/O** 是 **内核数据准备好** 和 **数据从内核态拷贝到用户态** 这两个过程都不用等待。

发起 `aio_read` 之后，就立即返回，内核自动将数据从内核空间拷贝到应用程序空间，这个拷贝过程同样是异步的，内核自动完成的，和前面的同步操作不一样，应用程序并不需要主动发起拷贝动作。



拿例子理解几种I/O模型

老三关注了很多UP主，有些UP主是老鸽子，到了更新的时间：

阻塞I/O就是，老三不干别的，就干等着，盯着UP的更新。

非阻塞I/O就是，老三发现UP没更，就去喝个茶什么的，过一会儿来盯一次，一直等到UP更新。

基于非阻塞的I/O多路复用好比，老三发现UP没更，就去干别的，过了一会儿B站推送消息了，老三一看，有很多条，就去翻动态，看看等的UP是不是更新了。

异步I/O就是，老三说UP你该更了，UP赶紧爆肝把视频做出来，然后把视频亲自呈到老三面前，这个过程不用等待。



首页 动态 投稿 57 视频列表 搜索视频、动态

置顶 你滴鸽宗，总在挖坑



小约翰可汗

09-28

感谢大家为我p头像，不过私信确实是真的看不过来了，大家如果想p图的话可以发在这个话题下哦#小约翰头像二创计划#



33. 详细讲一讲I/O多路复用？

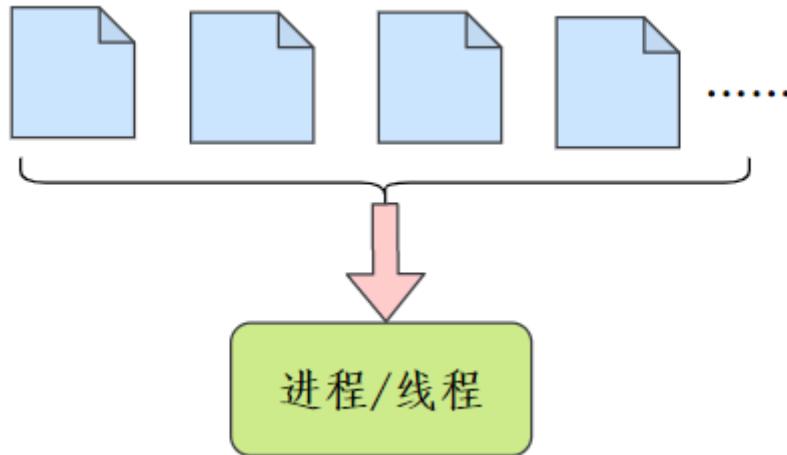
我们先了解什么是I/O多路复用？

我们在传统的I/O模型中，如果服务端需要支持多个客户端，我们可能要为每个客户端分配一个进程/线程。

不管是基于重一点的进程模型，还是轻一点的线程模型，假如连接多了，操作系统是扛不住的。

所以就引入了**I/O多路复用** 技术。

简单说，就是一个进程/线程维护多个Socket，这个多路复用就是多个连接复用一个进程/线程。



我们来看看I/O多路复用三种实现机制：

- select

select 实现多路复用的方式是：

将已连接的 Socket 都放到一个**文件描述符集合** fd_set，然后调用 select 函数将 fd_set 集合拷贝到内核里，让内核来检查是否有网络事件产生，检查的方式很粗暴，就是通过遍历 fd_set 的方式，当检查到有事件产生后，将此 Socket 标记为可读或可写，接着再把整个 fd_set 拷贝回用户态里，然后用户态还需要再通过遍历的方法找到可读或可写的 Socket，再对其处理。

select 使用固定长度的 BitsMap，表示文件描述符集合，而且所支持的文件描述符的个数是有限制的，在Linux 系统中，由内核中的 FD_SETSIZE 限制，默认最大值为 1024，只能监听 0~1023 的文件描述符。

select机制的缺点：

- (1) 每次调用 select，都需要把 fd_set 集合从用户态拷贝到内核态，如果 fd_set 集合很大时，那这个开销也很大，比如百万连接却只有少数活跃连接时这样做就太没有效率。
- (2) 每次调用 select 都需要在内核遍历传递进来的所有 fd_set，如果 fd_set 集合很大时，那这个开销也很大。
- (3) 为了减少数据拷贝带来的性能损坏，内核对被监控的 fd_set 集合大小做了限制，一般为 1024，如果想要修改会比较麻烦，可能还需要编译内核。
- (4) 每次调用 select 之前都需要遍历设置监听集合，重复工作。

- poll

poll 不再用 BitsMap 来存储所关注的文件描述符，取而代之用动态数组，以链表形式来组织，突破了select 的文件描述符个数限制，当然还会受到系统文件描述符限制。

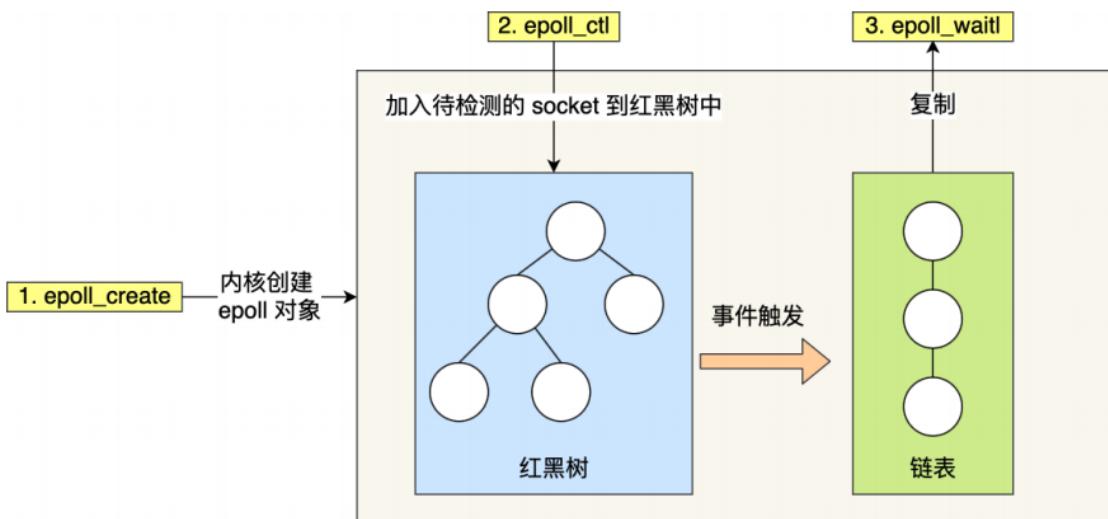
但是 poll 和 select 并没有太大的本质区别，都是使用线性结构存储进程关注的Socket 集合，因此都需要遍历文件描述符集合来找到可读或可写的Socket，时间复杂度为 $O(n)$ ，而且也需要在用户态与内核态之间拷贝文件描述符集合，这种方式随着并发数上来，性能的损耗会呈指数级增长。

- epoll

epoll 通过两个方面，很好解决了 select/poll 的问题。

第一点，epoll 在内核里使用 **红黑树来跟踪进程所有待检测的文件描述字**，把需要监控的 socket 通过 `epoll_ctl()` 函数加入内核中的红黑树里，红黑树是个高效的数据结构，增删查一般时间复杂度是 $O(\log n)$ ，通过对这棵黑红树进行操作，这样就不需要像 select/poll 每次操作时都传入整个 socket 集合，只需要传入一个待检测的 socket，**减少了内核和用户空间大量的数据拷贝和内存分配**。

第二点，epoll 使用事件驱动的机制，内核里 **维护了一个链表来记录就绪事件**，当某个 socket 有事件发生时，通过回调函数，内核会将其加入到这个就绪事件列表中，当用户调用 `epoll_wait()` 函数时，只会返回有事件发生的文件描述符的个数，不需要像 select/poll 那样轮询扫描整个 socket 集合，大大提高了检测的效率。



epoll 的方式即使监听的 Socket 数量越多的时候，效率不会大幅度降低，能够同时监听的 Socket 的数目也非常的多了，上限就为系统定义的进程打开的最大文件描述符个数。因而，**epoll 被称为解决 C10K 问题的利器**。

博主水平有限，参阅的资料在某些问题上也有一些出入，如有错漏，欢迎指出！

参考：

- [1]. [这可能最全的操作系统面试题](#)
- [2]. [操作系统常见面试题（2021最新版）](#)
- [3]. 小林coding《图解系统》
- [4]. 《现代操作系统》
- [5]. 《深入理解计算机系统》
- [6]. 《操作系统之哲学原理》
- [7]. [IO多路复用与epoll原理探究](#)

关注公众号：三分恶

手册更新动态
即刻送达



添加个人微信：ThirdFighter

技术交流
加大佬云集微信群



第四部分：数据库

一、MySQL

| 基础



作为SQL Boy，基础部分不会有人不会吧？面试也不怎么问，基础掌握不错的小伙伴可以跳过这一部分。当然，可能会现场写一些SQL语句，SQL语句可以通过牛客、LeetCode、LintCode之类的网站来练习。

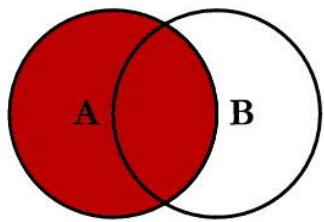
1. 什么是内连接、外连接、交叉连接、笛卡尔积呢？

- 内连接（inner join）：取得两张表中满足存在连接匹配关系的记录。
- 外连接（outer join）：不只取得两张表中满足存在连接匹配关系的记录，还包括某张表（或两张表）中不满足匹配关系的记录。
- 交叉连接（cross join）：显示两张表所有记录一一对应，没有匹配关系进行筛选，它是笛卡尔积在SQL中的实现，如果A表有m行，B表有n行，那么A和B交叉连接的结果就有 $m \times n$ 行。
- 笛卡尔积：是数学中的一个概念，例如集合A={a,b}，集合B={1,2,3}，那么 $A \times B = \{(a,1), (a,2), (a,3), (b,1), (b,2), (b,3)\}$ 。

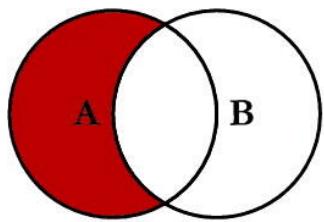
2. 那MySQL 的内连接、左连接、右连接有什么区别？

MySQL的连接主要分为内连接和外连接，外连接常用的有左连接、右连接。

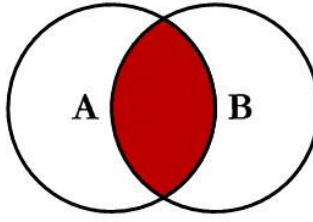
SQL JOINS



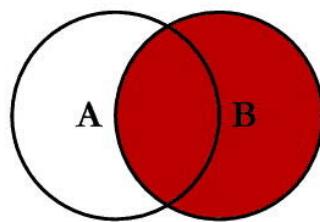
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



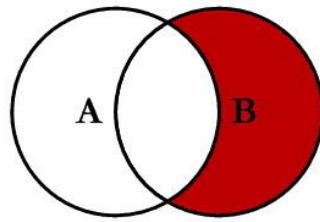
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



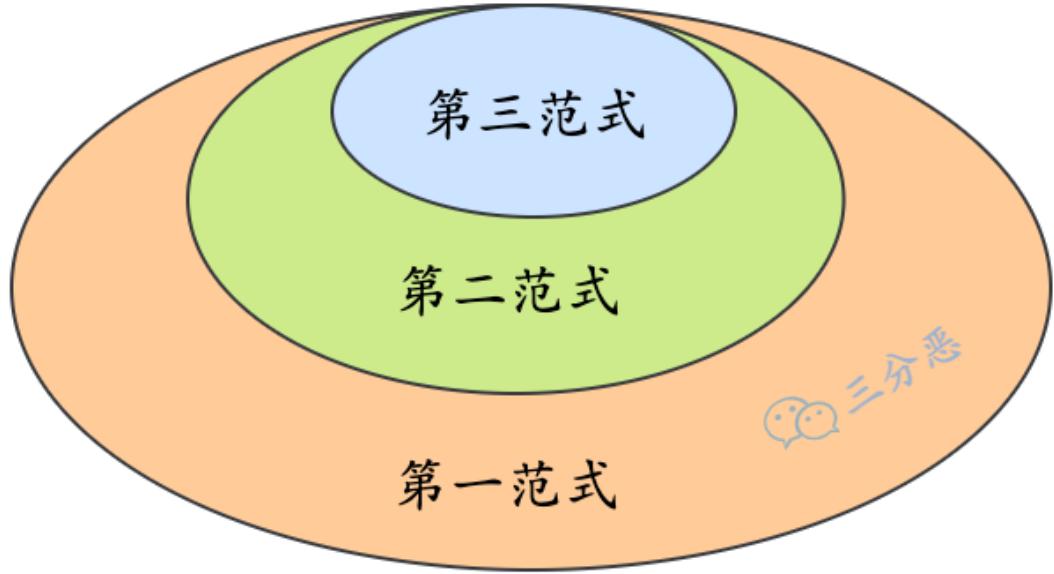
```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

© C.L. Moffatt, 2008

- inner join 内连接，在两张表进行连接查询时，只保留两张表中完全匹配的结果集
- left join 在两张表进行连接查询时，会返回左表所有的行，即使在右表中没有匹配的记录。
- right join 在两张表进行连接查询时，会返回右表所有的行，即使在左表中没有匹配的记录。

3.说一下数据库的三大范式？

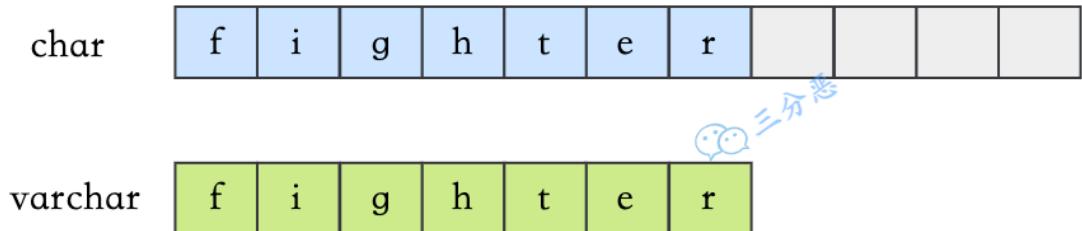


- 第一范式：数据表中的每一列（每个字段）都不可以再拆分。
例如用户表，用户地址还可以拆分成国家、省份、市，这样才是符合第一范式的。
- 第二范式：在第一范式的基础上，非主键列完全依赖于主键，而不能是依赖于主键的一部分。
例如订单表里，存储了商品信息（商品价格、商品类型），那就需要把商品ID和订单ID作为联合主键，才满足第二范式。
- 第三范式：在满足第二范式的基础上，表中的非主键只依赖于主键，而不依赖于其他非主键。
例如订单表，就不能存储用户信息（姓名、地址）。



三大范式的作用是为了控制数据库的冗余，是对空间的节省，实际上，一般互联网公司的设计都是反范式的，通过冗余一些数据，避免跨表跨库，利用空间换时间，提高性能。

4.varchar与char的区别？



char：

- char表示定长字符串，长度是固定的；
- 如果插入数据的长度小于char的固定长度时，则用空格填充；
- 因为长度固定，所以存取速度要比varchar快很多，甚至能快50%，但正因为其长度固定，所以会占据多余的空间，是空间换时间的做法；
- 对于char来说，最多能存放的字符个数为255，和编码无关

varchar：

- varchar表示可变长字符串，长度是可变的；
- 插入的数据是多长，就按照多长来存储；
- varchar在存取方面与char相反，它存取慢，因为长度不固定，但正因如此，不占据多余的空间，是时间换空间的做法；
- 对于varchar来说，最多能存放的字符个数为65532

日常的设计，对于长度相对固定的字符串，可以使用char，对于长度不确定的，使用varchar更合适一些。

5.blob和text有什么区别？

- blob用于存储二进制数据，而text用于存储大字符串。
- blob没有字符集，text有一个字符集，并且根据字符集的校对规则对值进行排序和比较

6.DATETIME和TIMESTAMP的异同？

相同点：

1. 两个数据类型存储时间的表现格式一致。均为 YYYY-MM-DD HH:MM:SS
2. 两个数据类型都包含「日期」和「时间」部分。
3. 两个数据类型都可以存储微秒的小数秒（秒后6位小数秒）

区别：



1. 日期范围：DATETIME 的日期范围是 1000-01-01 00:00:00.000000 到 9999-12-31 23:59:59.999999；TIMESTAMP 的时间范围是 1970-01-01 00:00:01.000000 UTC 到 `2038-01-09 03:14:07.999999 UTC
2. 存储空间：DATETIME 的存储空间为 8 字节；TIMESTAMP 的存储空间为 4 字节
3. 时区相关：DATETIME 存储时间与时区无关；TIMESTAMP 存储时间与时区有关，显示的值也依赖于时区
4. 默认值：DATETIME 的默认值为 null；TIMESTAMP 的字段默认不为空(not null)，默认值为当前时间(CURRENT_TIMESTAMP)

7.MySQL中 in 和 exists 的区别？

MySQL中的in语句是把外表和内表作hash连接，而exists语句是对外表作loop循环，每次loop循环再对内表进行查询。我们可能认为exists比in语句的效率要高，这种说法其实是不准确的，要区分情景：

1. 如果查询的两个表大小相当，那么用in和exists差别不大。
2. 如果两个表中一个较小，一个是大表，则子查询表大的用exists，子查询表小的用in。
3. not in 和not exists：如果查询语句使用了not in，那么内外表都进行全表扫描，没有用到索引；而not extsts的子查询依然能用到表上的索引。所以无论那个表大，用not exists都比not in要快。

8.MySQL里记录货币用什么字段类型比较好？

货币在数据库中MySQL常用Decimal和Numric类型表示，这两种类型被MySQL实现为同样的类型。他们被用于保存与货币有关的数据。

例如salary DECIMAL(9,2)，9(precision)代表将被用于存储值的总的小数位数，而2(scale)代表将被用于存储小数点后的位数。存储在salary列中的值的范围是从-9999999.99到9999999.99。

DECIMAL和NUMERIC值作为字符串存储，而不是作为二进制浮点数，以便保存那些值的小数精度。

之所以不使用float或者double的原因：因为float和double是以二进制存储的，所以有一定的误差。

9.MySQL怎么存储emoji？

MySQL可以直接使用字符串存储emoji。

但是需要注意的，utf8 编码是不行的，MySQL中的utf8是阉割版的 utf8，它最多只用3个字节存储字符，所以存储不了表情。那该怎么办？

需要使用utf8mb4编码。

```
1 | alter table blogs modify content text CHARACTER SET utf8mb4  
    COLLATE utf8mb4_unicode_ci not null;
```

10.drop、delete与truncate的区别？

三者都表示删除，但是三者有一些差别：

	delete	truncate	drop
类型	属于DML	属于DDL	属于DDL
回滚	可回滚	不可回滚	不可回滚
删除内容	表结构还在，删除表的全部或者一部分数据行	表结构还在，删除表中的所有数据	从数据库中删除表，所有数据行，索引和权限也会被删除
删除速度	删除速度慢，需要逐行删除	删除速度快	删除速度最快

因此，在不再需要一张表的时候，用drop；在想删除部分数据行时候，用delete；在保留表而删除所有数据的时候用truncate。

11.UNION与UNION ALL的区别？

- 如果使用UNION ALL，不会合并重复的记录行
- 效率 UNION 高于 UNION ALL

12.count(1)、count(*) 与 count(列名) 的区别？

count(1)

count(*)

count(列名)

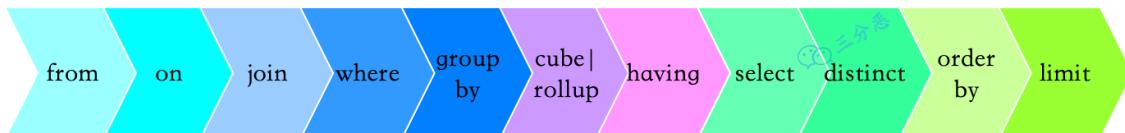
执行效果：

- count(*)包括了所有的列，相当于行数，在统计结果的时候，不会忽略列值为NULL
- count(1)包括了忽略所有列，用1代表代码行，在统计结果的时候，不会忽略列值为NULL
- count(列名)只包括列名那一列，在统计结果的时候，会忽略列值为空（这里的空不是只空字符串或者0，而是表示null）的计数，即某个字段值为NULL时，不统计。

执行速度：

- 列名为主键，count(列名)会比count(1)快
- 列名不为主键，count(1)会比count(列名)快
- 如果表多个列并且没有主键，则 count (1) 的执行效率优于 count (*)
- 如果有主键，则 select count (主键) 的执行效率是最优的
- 如果表只有一个字段，则 select count (*) 最优。

13.一条SQL查询语句的执行顺序？

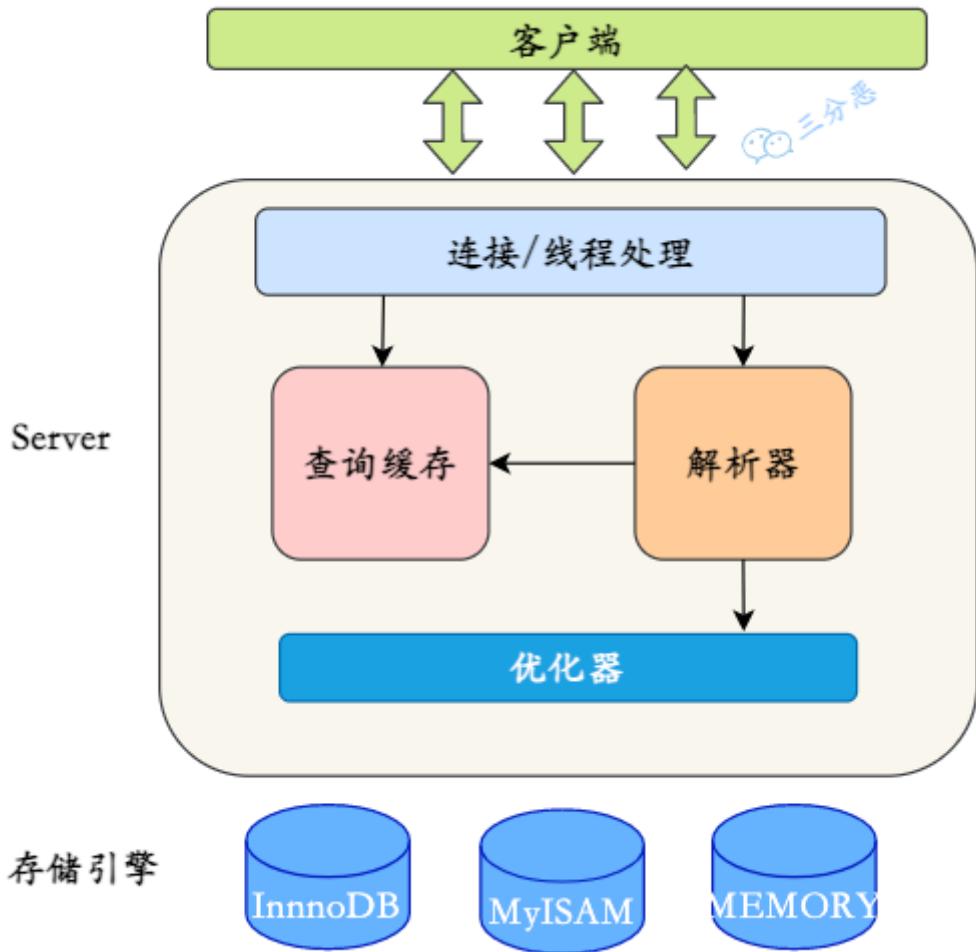


1. FROM：对FROM子句中的左表<left_table>和右表<right_table>执行笛卡儿积(Cartesianproduct)，产生虚拟表VT1
2. ON：对虚拟表VT1应用ON筛选，只有那些符合<join_condition>的行才被插入虚拟表VT2中
3. JOIN：如果指定了OUTER JOIN（如LEFT OUTER JOIN、RIGHT OUTER JOIN），那么保留表中未匹配的行作为外部行添加到虚拟表VT2中，产生虚拟表VT3。如果FROM子句包含两个以上表，则对上一个连接生成的结果表VT3和下一个表重复执行步骤1) ~步骤3)，直到处理完所有的表为止

- 4. WHERE** : 对虚拟表VT3应用WHERE过滤条件, 只有符合<where_condition>的记录才被插入虚拟表VT4中
- 5. GROUP BY** : 根据GROUP BY子句中的列, 对VT4中的记录进行分组操作, 产生VT5
- 6. CUBE|ROLLUP** : 对表VT5进行CUBE或ROLLUP操作, 产生表VT6
- 7. HAVING** : 对虚拟表VT6应用HAVING过滤器, 只有符合<having_condition>的记录才被插入虚拟表VT7中。
- 8. SELECT** : 第二次执行SELECT操作, 选择指定的列, 插入到虚拟表VT8中
- 9. DISTINCT** : 去除重复数据, 产生虚拟表VT9
- 10. ORDER BY** : 将虚拟表VT9中的记录按照<order_by_list>进行排序操作, 产生虚拟表VT10。11)
- 11. LIMIT** : 取出指定行的记录, 产生虚拟表VT11, 并返回给查询用户

数据库架构

14. 说说 MySQL 的基础架构?



MySQL逻辑架构图主要分三层：

- 客户端：最上层的服务并不是MySQL所独有的，大多数基于网络的客户端/服务器的工具或者服务都有类似的架构。比如连接处理、授权认证、安全等等。
- Server层：大多数MySQL的核心服务功能都在这一层，包括查询解析、分析、优化、缓存以及所有的内置函数（例如，日期、时间、数学和加密函数），所有跨存储引擎的功能都在这一层实现：存储过程、触发器、视图等。
- 存储引擎层：第三层包含了存储引擎。存储引擎负责MySQL中数据的存储和提取。Server层通过API与存储引擎进行通信。这些接口屏蔽了不同存储引擎之间的差异，使得这些差异对上层的查询过程透明。

15.一条 SQL 查询语句在 MySQL 中如何执行的？

- 先检查该语句 **是否有权限**，如果没有权限，直接返回错误信息，如果有权限会先查询缓存 (MySQL8.0 版本以前)。

- 如果没有缓存，分析器进行 **语法分析**，提取 sql 语句中 select 等关键元素，然后判断 sql 语句是否有语法错误，比如关键词是否正确等等。
- 语法解析之后，MySQL的服务器会对查询的语句进行优化，确定执行的方案。
- 完成查询优化后，按照生成的执行计划 **调用数据库引擎接口**，返回执行结果。

存储引擎

16.MySQL有哪些常见存储引擎？

主要存储引擎



主要存储引擎以及功能如下：

功能	MyISAM	MEMORY	InnoDB
存储限制	256TB	RAM	64TB
支持事务	No	No	Yes
支持全文索引	Yes	No	Yes
支持树索引	Yes	Yes	Yes
支持哈希索引	No	Yes	Yes
支持数据缓存	No	N/A	Yes
支持外键	No	No	Yes

MySQL5.5之前，默认存储引擎是MyISAM，5.5之后变成了InnoDB。

InnoDB支持的哈希索引是自适应的，InnoDB会根据表的使用情况自动为表生成哈希索引，不能人为干预是否在一张表中生成哈希索引。

MySQL 5.6开始InnoDB支持全文索引。

17.那存储引擎应该怎么选择？

大致上可以这么选择：

- 大多数情况下，使用默认的InnoDB就够了。如果要提供提交、回滚和恢复的事务安全（ACID 兼容）能力，并要求实现并发控制，InnoDB 就是比较靠前的选择了。
- 如果数据表主要用来插入和查询记录，则 MyISAM 引擎提供较高的处理效率。
- 如果只是临时存放数据，数据量不大，并且不需要较高的数据安全性，可以选择将数据保存在内存的 MEMORY 引擎中，MySQL 中使用该引擎作为临时表，存放查询的中间结果。

使用哪一种引擎可以根据需要灵活选择，因为存储引擎是基于表的，所以一个数据库中多个表可以使用不同的引擎以满足各种性能和实际需求。使用合适的存储引擎将会提高整个数据库的性能。

18.InnoDB和MyISAM主要有什么区别？

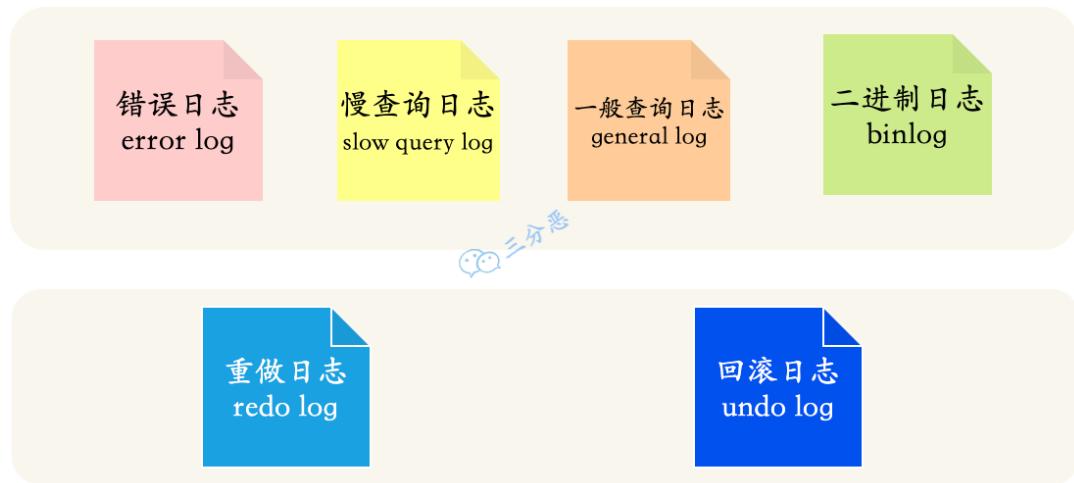
PS:MySQL8.0都开始慢慢流行了，如果不是面试，MyISAM其实可以不用怎么了解。

MyISAM和InnoDB区别



- 1. 存储结构**: 每个MyISAM在磁盘上存储成三个文件；InnoDB所有的表都保存在同一个数据文件中（也可能是多个文件，或者是独立的表空间文件），InnoDB表的大小只受限于操作系统文件的大小，一般为2GB。
- 2. 事务支持**: MyISAM不提供事务支持；InnoDB提供事务支持事务，具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全特性。
- 3 最小锁粒度**: MyISAM只支持表级锁，更新时会锁住整张表，导致其它查询和更新都会被阻塞InnoDB支持行级锁。
- 4. 索引类型**: MyISAM的索引为非聚簇索引，数据结构是B树；InnoDB的索引聚簇索引，数据结构是B+树。
- 5. 主键必需**: MyISAM允许没有任何索引和主键的表存在；InnoDB如果没有设定主键或者非空唯一索引，就会自动生成一个6字节的主键(用户不可见)，数据是主索引的一部分，附加索引保存的是主索引的值。
- 6. 表的具体行数**: MyISAM保存了表的总行数，如果select count(*) from table;会直接取出该值；InnoDB没有保存表的总行数，如果使用select count(*) from table; 就会遍历整个表；但是在加了where条件后，MyISAM和InnoDB处理的方式都一样。
- 7. 外键支持**: MyISAM不支持外键；InnoDB支持外键。

19.MySQL日志文件有哪些？分别介绍下作用？



MySQL日志文件有很多，包括：

- 错误日志（error log）：错误日志文件对MySQL的启动、运行、关闭过程进行了记录，能帮助定位MySQL问题。
- 慢查询日志（slow query log）：慢查询日志是用来记录执行时间超过 `long_query_time` 这个变量定义的时长的查询语句。通过慢查询日志，可以查找出哪些查询语句的执行效率很低，以便进行优化。
- 一般查询日志（general log）：一般查询日志记录了所有对MySQL数据库请求的信息，无论请求是否正确执行。
- 二进制日志（bin log）：关于二进制日志，它记录了数据库所有执行的DDL和DML语句（除了数据查询语句`select`、`show`等），以事件形式记录并保存在二进制文件中。

还有两个InnoDB存储引擎特有的日志文件：

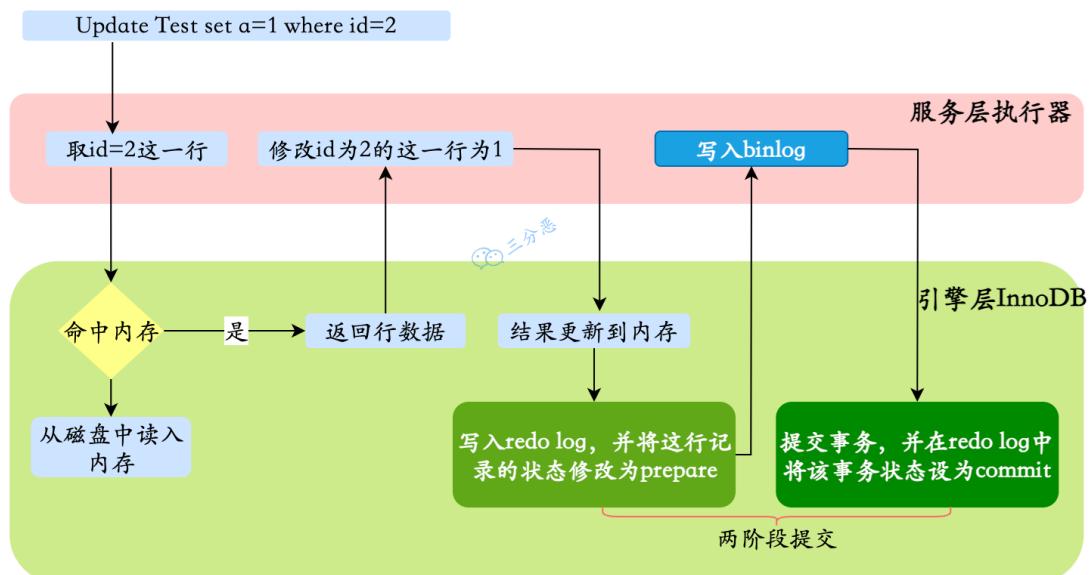
- 重做日志（redo log）：重做日志至关重要，因为它们记录了对于InnoDB存储引擎的事务日志。
- 回滚日志（undo log）：回滚日志同样也是InnoDB引擎提供的日志，顾名思义，回滚日志的作用就是对数据进行回滚。当事务对数据库进行修改，InnoDB引擎不仅会记录redo log，还会生成对应的undo log日志；如果事务执行失败或调用了`rollback`，导致事务需要回滚，就可以利用undo log中的信息将数据回滚到修改之前的样子。

20.binlog和redo log有什么区别？

- bin log会记录所有与数据库有关的日志记录，包括InnoDB、MyISAM等存储引擎的日志，而redo log只记InnoDB存储引擎的日志。
- 记录的内容不同，bin log记录的是关于一个事务的具体操作内容，即该日志是逻辑日志。而redo log记录的是关于每个页（Page）的更改的物理情况。
- 写入的时间不同，bin log仅在事务提交前进行提交，也就是只写磁盘一次。而在事务进行的过程中，却不断有redo entry被写入redo log中。
- 写入的方式也不相同，redo log是循环写入和擦除，bin log是追加写入，不会覆盖已经写的文件。

21.一条更新语句怎么执行的了解吗？

更新语句的执行是Server层和引擎层配合完成，数据除了要写入表中，还要记录相应的日志。



1. 执行器先找引擎获取ID=2这一行。ID是主键，存储引擎检索数据，找到这一行。如果ID=2这一行所在的数据页本来就在内存中，就直接返回给执行器；否则，需要先从磁盘读入内存，然后再返回。
2. 执行器拿到引擎给的行数据，把这个值加上1，比如原来是N，现在就是N+1，得到新的一行数据，再调用引擎接口写入这行新数据。
3. 引擎将这行新数据更新到内存中，同时将这个更新操作记录到redo log里面，此时redo log处于prepare状态。然后告知执行器执行完成了，随时可以提交事务。
4. 执行器生成这个操作的binlog，并把binlog写入磁盘。
5. 执行器调用引擎的提交事务接口，引擎把刚刚写入的redo log改成提交(commit)状态，更新完成。

从上图可以看出，MySQL在执行更新语句的时候，在服务层进行语句的解析和执行，在引擎层进行数据的提取和存储；同时在服务层对binlog进行写入，在InnoDB内进行redo log的写入。

不仅如此，在对redo log写入时有两个阶段的提交，一是binlog写入之前 **prepare** 状态的写入，二是binlog写入之后 **commit** 状态的写入。

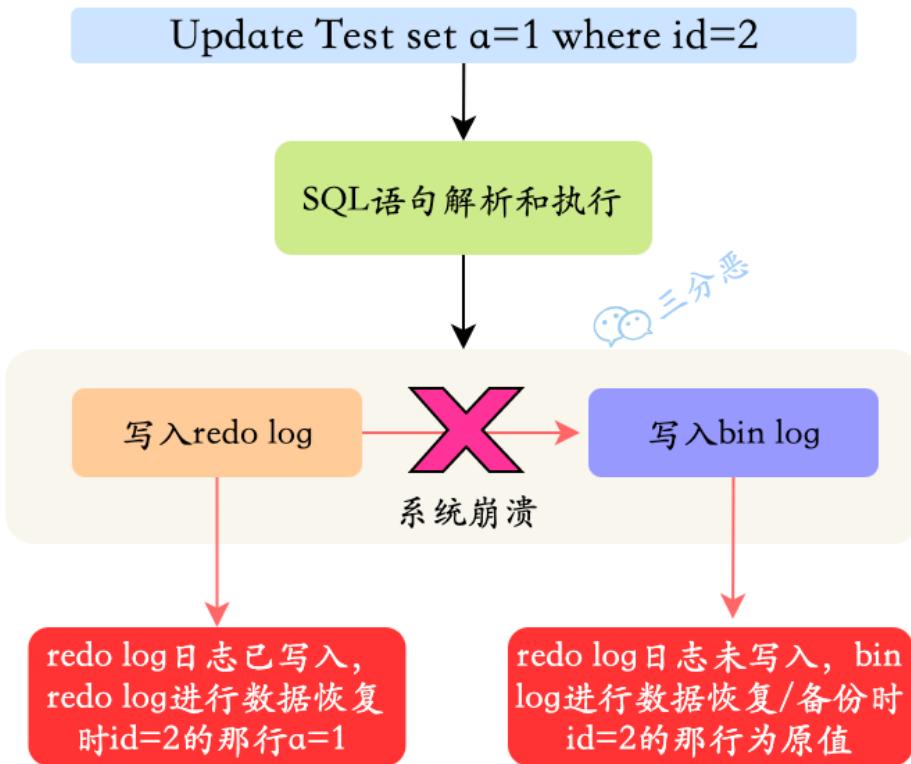
22.那为什么要两阶段提交呢？

为什么要两阶段提交呢？直接提交不行吗？

我们可以假设不采用两阶段提交的方式，而是采用“单阶段”进行提交，即要么先写入redo log，后写入binlog；要么先写入binlog，后写入redo log。这两种方式的提交都会导致原先数据库的状态和被恢复后的数据库的状态不一致。

先写入redo log，后写入binlog：

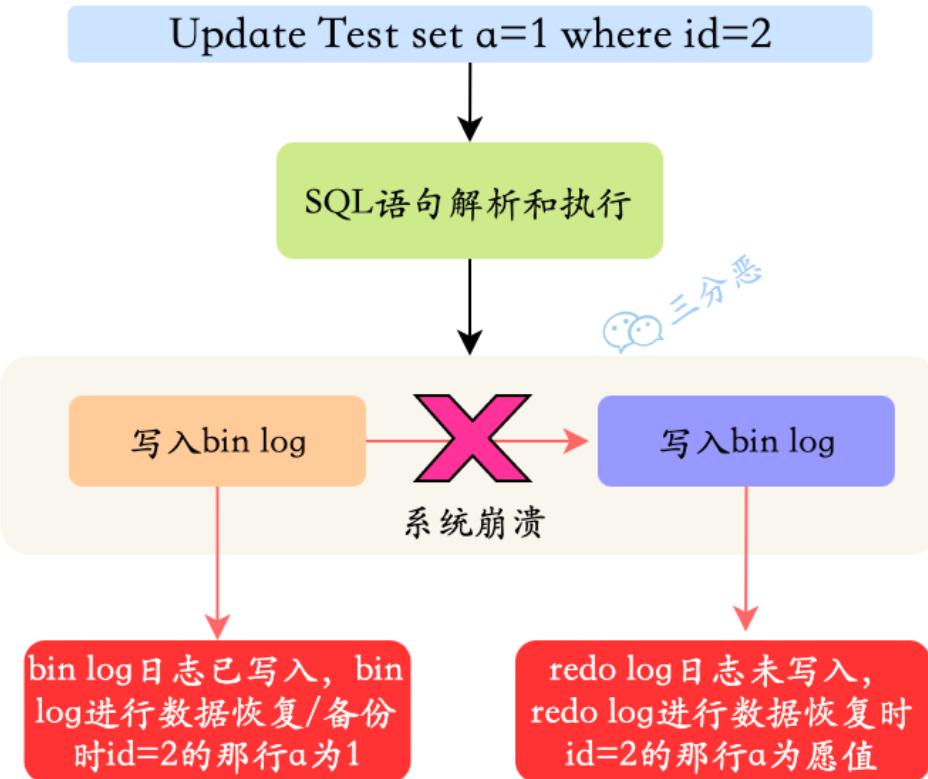
在写完redo log之后，数据此时具有 **crash-safe** 能力，因此系统崩溃，数据会恢复成事务开始之前的状态。但是，若在redo log写完时候，binlog写入之前，系统发生了宕机。此时binlog没有对上面的更新语句进行保存，导致当使用binlog进行数据库的备份或者恢复时，就少了上述的更新语句。从而使得 **id=2** 这一行的数据没有被更新。



先写redo log, 后写bin log的问题

先写binlog, 后写redo log:

写完binlog之后，所有的语句都被保存，所以通过binlog复制或恢复出来的数据库中 id=2这一行的数据会被更新为a=1。但是如果在redo log写入之前，系统崩溃，那么 redo log中记录的这个事务会无效，导致实际数据库中 **id=2** 这一行的数据并没有更新。

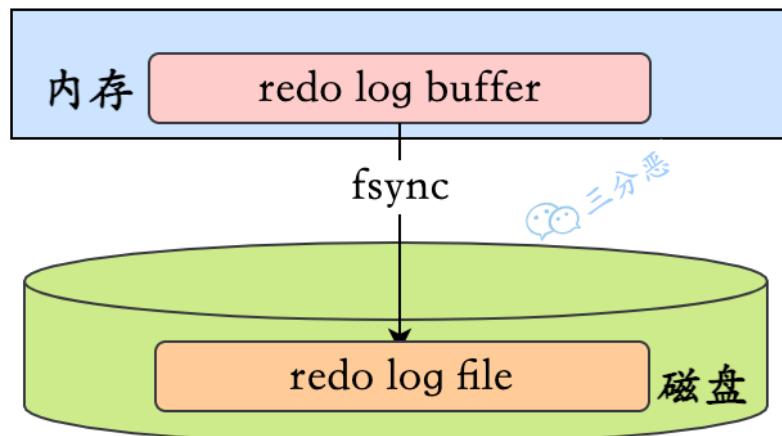


先写bin log，后写redo log的问题

简单说，redo log和binlog都可以用于表示事务的提交状态，而两阶段提交就是让这两个状态保持逻辑上的一致。

23.redo log怎么刷入磁盘的知道吗？

redo log的写入不是直接落到磁盘，而是在内存中设置了一片称之为 **redo log buffer** 的连续内存空间，也就是 **redo 日志缓冲区**。



什么时候会刷入磁盘？

在如下的一些情况下，log buffer的数据会刷入磁盘：

- log buffer 空间不足时

log buffer 的大小是有限的，如果不停的往这个有限大小的 log buffer 里塞入日志，很快它就会被填满。如果当前写入 log buffer 的redo 日志量已经占满了 log buffer 总容量的大约一半左右，就需要把这些日志刷新到磁盘上。

- 事务提交时

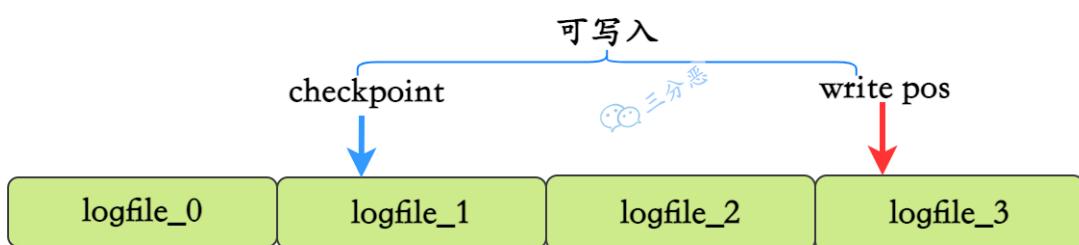
在事务提交时，为了保证持久性，会把log buffer中的日志全部刷到磁盘。注意，这时候，除了本事务的，可能还会刷入其它事务的日志。

- 后台线程输入

有一个后台线程，大约每秒都会刷新一次 log buffer 中的 redo log 到磁盘。

- 正常关闭服务器时
- 触发checkpoint规则

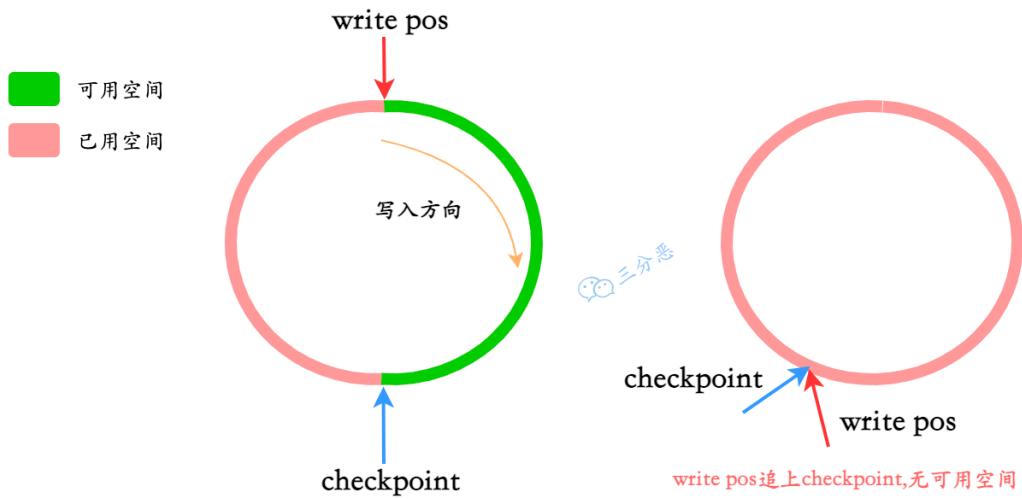
重做日志缓存、重做日志文件都是以块（block）的方式进行保存的，称之为重做日志块（redo log block），块的大小是固定的512字节。我们的redo log它是固定大小的，可以看作是一个逻辑上的 log group，由一定数量的 log block 组成。



它的写入方式是从头到尾开始写，写到末尾又回到开头循环写。

其中有两个标记位置：

write pos 是当前记录的位置，一边写一边后移，写到第3号文件末尾后就回到0号文件开头。**checkpoint** 是当前要擦除的位置，也是往后推移并且循环的，擦除记录前要把记录更新到磁盘。



当 `write_pos` 追上 `checkpoint` 时，表示redo log日志已经写满。这时候就不能接着往里写数据了，需要执行 `checkpoint` 规则腾出可写空间。

所谓的**checkpoint规则**，就是checkpoint触发后，将buffer中日志页都刷到磁盘。

SQL 优化

24. 慢SQL如何定位呢？

慢SQL的监控主要通过两个途径：

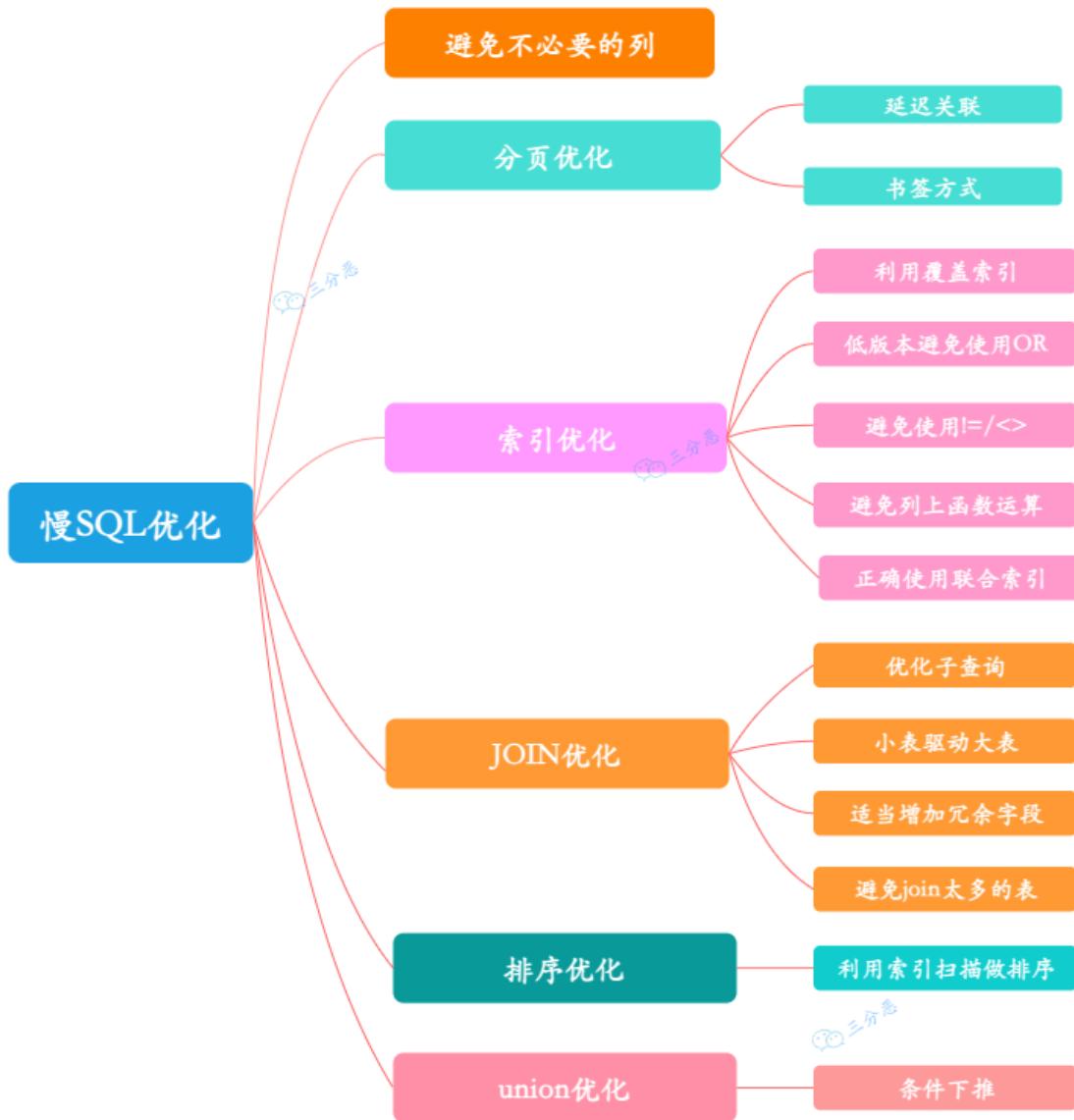


- 慢查询日志：开启MySQL的慢查询日志，再通过一些工具比如`mysqldumpslow`去分析对应的慢查询日志，当然现在一般的云厂商都提供了可视化的平台。

- 服务监控：可以在业务的基建中加入对慢SQL的监控，常见的方案有字节码插桩、连接池扩展、ORM框架过程，对服务运行中的慢SQL进行监控和告警。

25.有哪些方式优化慢SQL？

慢SQL的优化，主要从两个方面考虑，SQL语句本身的优化，以及数据库设计的优化。



H6 避免不必要的列

这个是老生常谈，但还是经常会出现的情况，SQL查询的时候，应该只查询需要的列，而不要包含额外的列，像 `select *` 这种写法应该尽量避免。

H6 分页优化

在数据量比较大，分页比较深的情况下，需要考虑分页的优化。

例如：

```
1 | select * from table where type = 2 and level = 9 order by id  
asc limit 190289,10;
```

优化方案：

- 延迟关联

先通过where条件提取出主键，在将该表与原数据表关联，通过主键id提取数据行，而不是通过原来的二级索引提取数据行

例如：

```
1 | select a.* from table a,  
2 |   (select id from table where type = 2 and level = 9 order  
by id asc limit 190289,10 ) b  
3 |   where a.id = b.id
```

- 书签方式

书签方式就是找到limit第一个参数对应的主键值，根据这个主键值再去过滤并limit

例如：

```
1 | select * from table where id >  
2 |   (select * from table where type = 2 and level = 9 order by  
id asc limit 190
```

H6 索引优化

合理地设计和使用索引，是优化慢SQL的利器。

利用覆盖索引

InnoDB使用非主键索引查询数据时会回表，但是如果索引的叶节点中已经包含要查询的字段，那它没有必要再回表查询了，这就叫覆盖索引

例如对于如下查询：

```
1 | select name from test where city='上海'
```

我们将被查询的字段建立到联合索引中，这样查询结果就可以直接从索引中获取

```
1 | alter table test add index idx_city_name (city, name);
```

低版本避免使用or查询

在 MySQL 5.0 之前的版本要尽量避免使用 or 查询，可以使用 union 或者子查询来替代，因为早期的 MySQL 版本使用 or 查询可能会导致索引失效，高版本引入了索引合并，解决了这个问题。

避免使用 != 或者 <> 操作符

SQL中，不等于操作符会导致查询引擎放弃查询索引，引起全表扫描，即使比较的字段上有索引

解决方法：通过把不等于操作符改成or，可以使用索引，避免全表扫描

例如，把 `column<>'aaa'`，改成`column>'aaa' or column<'aaa'`，就可以使用索引了

适当使用前缀索引

适当地使用前缀所云，可以降低索引的空间占用，提高索引的查询效率。

比如，邮箱的后缀都是固定的“`@xxx.com`”，那么类似这种后面几位为固定值的字段就非常适合定义为前缀索引

```
1 | alter table test add index index2(email(6));
```

PS:需要注意的是，前缀索引也存在缺点，MySQL无法利用前缀索引做order by和group by 操作，也无法作为覆盖索引

避免列上函数运算

要避免在列字段上进行算术运算或其他表达式运算，否则可能会导致存储引擎无法正确使用索引，从而影响了查询的效率

```
1 | select * from test where id + 1 = 50;
2 | select * from test where month(updateTime) = 7;
```

正确使用联合索引

使用联合索引的时候，注意最左匹配原则。

H6 JOIN优化

优化子查询

尽量使用 Join 语句来替代子查询，因为子查询是嵌套查询，而嵌套查询会新创建一张临时表，而临时表的创建与销毁会占用一定的系统资源以及花费一定的时间，同时对于返回结果集比较大的子查询，其对查询性能的影响更大

小表驱动大表

关联查询的时候要拿小表去驱动大表，因为关联的时候，MySQL内部会遍历驱动表，再去连接被驱动表。

比如left join，左表就是驱动表，A表小于B表，建立连接的次数就少，查询速度就被加快了。

```
1 | select name from A left join B ;
```

适当增加冗余字段

增加冗余字段可以减少大量的连表查询，因为多张表的连表查询性能很低，所有可以适当的增加冗余字段，以减少多张表的关联查询，这是以空间换时间的优化策略

避免使用JOIN关联太多的表

《阿里巴巴Java开发手册》规定不要join超过三张表，第一join太多降低查询的速度，第二join的buffer会占用更多的内存。

如果不可避免要join多张表，可以考虑使用数据异构的方式异构到ES中查询。

H6 排序优化

利用索引扫描做排序

MySQL有两种方式生成有序结果：其一是对结果集进行排序的操作，其二是按照索引顺序扫描得出的结果自然是有序的

但是如果索引不能覆盖查询所需列，就不得不每扫描一条记录回表查询一次，这个读操作是随机IO，通常会比顺序全表扫描还慢

因此，在设计索引时，尽可能使用同一个索引既满足排序又用于查找行

例如：

```
1 | --建立索引 (date,staff_id,customer_id)
2 | select staff_id, customer_id from test where date = '2010-01-
01' order by staff_id, customer_id;
```

只有当索引的列顺序和ORDER BY子句的顺序完全一致，并且所有列的排序方向都一样时，才能够使用索引来对结果做排序

H6 UNION优化

条件下推

MySQL处理union的策略是先创建临时表，然后将各个查询结果填充到临时表中最后再来做查询，很多优化策略在union查询中都会失效，因为它无法利用索引

最好手工将where、limit等子句下推到union的各个子查询中，以便优化器可以充分利用这些条件进行优化

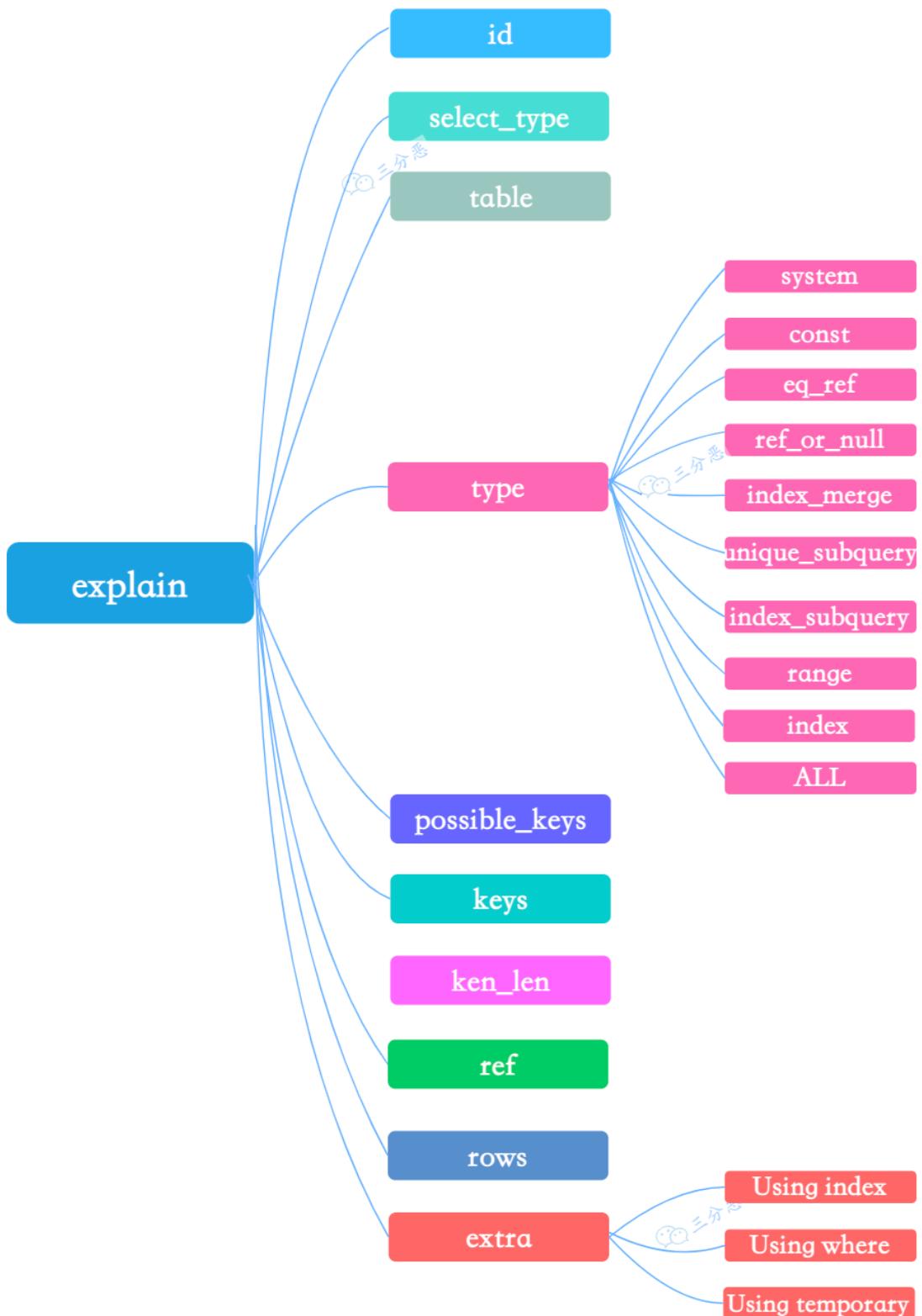
此外，除非确实需要服务器去重，一定要使用union all，如果不加all关键字，MySQL会给临时表加上distinct选项，这会导致对整个临时表做唯一性检查，代价很高。

26.怎么看执行计划（explain），如何理解其中各个字段的含义？

explain是sql优化的利器，除了优化慢sql，平时的sql编写，也应该先explain，查看一下执行计划，看看是否还有优化的空间。

直接在 select 语句之前增加 **explain** 关键字，就会返回执行计划的信息。

```
mysql> explain select name from student;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id  | select_type | table | type | possible_keys | key | key_len | ref  | rows |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|   1 | SIMPLE      | student | ALL  | NULL          | NULL | NULL    | NULL | 2    |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```



- 1. id** 列: MySQL会为每个select语句分配一个唯一的id值
- 2. select_type** 列, 查询的类型, 根据关联、union、子查询等等分类, 常见的查询类型有SIMPLE、PRIMARY。
- 3. table** 列: 表示 explain 的一行正在访问哪个表。
- 4. type** 列: 最重要的列之一。表示关联类型或访问类型, 即 MySQL 决定如何查找表中的行。

性能从最优到最差分别为： system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > range > index > ALL

- system

system：当表仅有一行记录时(系统表)，数据量很少，往往不需要进行磁盘IO，速度非常快

- const

const：表示查询时命中 primary key 主键或者 unique 唯一索引，或者被连接的部分是一个常量(const)值。这类扫描效率极高，返回数据量少，速度非常快。

- eq_ref

eq_ref：查询时命中主键 primary key 或者 unique key 索引， type 就是 eq_ref。

- ref_or_null

ref_or_null：这种连接类型类似于 ref，区别在于 MySQL 会额外搜索包含 NULL 值的行。

- index_merge

index_merge：使用了索引合并优化方法，查询使用了两个以上的索引。

- unique_subquery

unique_subquery：替换下面的 IN 子查询，子查询返回不重复的集合。

- index_subquery

index_subquery：区别于 unique_subquery，用于非唯一索引，可以返回重复值。

- range

range：使用索引选择行，仅检索给定范围内的行。简单点说就是针对一个有索引的字段，给定范围检索数据。在 where 语句中使用

between...and、**<**、**>**、**<=**、**in** 等条件查询 type 都是 range。

- index

index：Index 与 ALL 其实都是读全表，区别在于 index 是遍历索引树读取，而 ALL 是从硬盘中读取。

- ALL

就不用多说了，全表扫描。

5. possible_keys 列：显示查询可能使用哪些索引来查找，使用索引优化sql的时候比较重要。

6. key 列：这一列显示 mysql 实际采用哪个索引来优化对该表的访问，判断索引是否失效的时候常用。

7. key_len 列：显示了 MySQL 使用

8. ref 列：ref 列展示的就是与索引列作等值匹配的值，常见的有：const（常量），func，NULL，字段名。

9. rows 列：这也是一个重要的字段，MySQL 查询优化器根据统计信息，估算SQL要查到结果集需要扫描读取的数据行数，这个值非常直观显示SQL的效率好坏，原则上rows越少越好。

10. Extra 列：显示不适合在其它列的额外信息，虽然叫额外，但是也有一些重要的信息：

- Using index: 表示MySQL将使用覆盖索引，以避免回表
- Using where: 表示会在存储引擎检索之后再进行过滤
- Using temporary : 表示对查询结果排序时会使用一个临时表。

索引

索引可以说是MySQL面试中的重中之重，一定要彻底拿下。

27.能简单说一下索引的分类吗？

从三个不同维度对索引分类：



例如从基本使用使用的角度来讲：

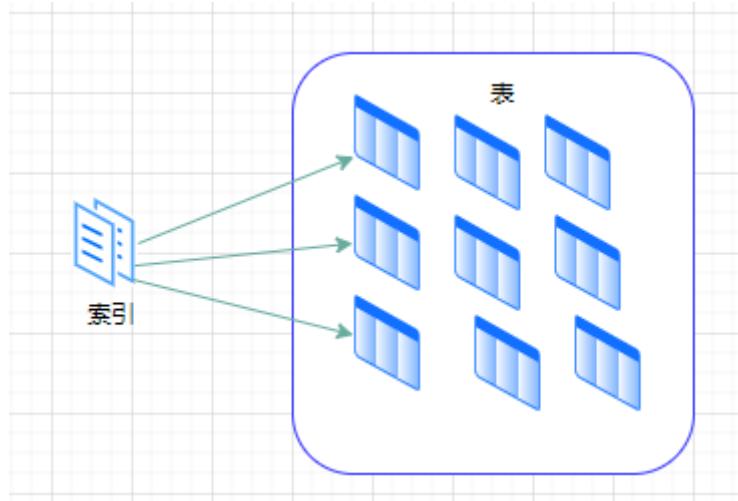
- 主键索引: InnoDB主键是默认的索引，数据列不允许重复，不允许为NULL，一个表只能有一个主键。
- 唯一索引: 数据列不允许重复，允许为NULL值，一个表允许多个列创建唯一索引。
- 普通索引: 基本的索引类型，没有唯一性的限制，允许为NULL值。
- 组合索引: 多列值组成一个索引，用于组合搜索，效率大于索引合并

28.为什么使用索引会加快查询？

传统的查询方法，是按照表的顺序遍历的，不论查询几条数据，MySQL需要将表的数据从头到尾遍历一遍。

在我们添加完索引之后，MySQL一般通过BTREE算法生成一个索引文件，在查询数据库时，找到索引文件进行遍历，在比较小的索引数据里查找，然后映射到对应的数据，能大幅提升查找的效率。

和我们通过书的目录，去查找对应的内容，一样的道理。



29. 创建索引有哪些注意点？

索引虽然是sql性能优化的利器，但是索引的维护也是需要成本的，所以创建索引，也要注意：

1. 索引应该建在查询应用频繁的字段

在用于 where 判断、 order 排序和 join 的(on)字段上创建索引。

2. 索引的个数应该适量

索引需要占用空间；更新时候也需要维护。

3. 区分度低的字段，例如性别，不要建索引。

离散度太低的字段，扫描的行数降低的有限。

4. 频繁更新的值，不要作为主键或者索引

维护索引文件需要成本；还会导致页分裂，IO次数增多。

5. 组合索引把散列性高(区分度高)的值放在前面

为了满足最左前缀匹配原则

6. 创建组合索引，而不是修改单列索引。

组合索引代替多个单列索引（对于单列索引，MySQL基本只能使用一个索引，所以经常使用多个条件查询时更适合使用组合索引）

7. 过长的字段，使用前缀索引。

当字段值比较长的时候，建立索引会消耗很多的空间，搜索起来也会很慢。我们可以通过截取字段的前面一部分内容建立索引，这个就叫前缀索引。

8. 不建议用无序的值(例如身份证、UUID)作为索引

当主键具有不确定性，会造成叶子节点频繁分裂，出现磁盘存储的碎片化

30. 索引哪些情况下会失效呢？

- 查询条件包含or，可能导致索引失效
- 如果字段类型是字符串，where时一定用引号括起来，否则会因为隐式类型转换，索引失效
- like通配符可能导致索引失效。
- 联合索引，查询时的条件列不是联合索引中的第一个列，索引失效。
- 在索引列上使用mysql的内置函数，索引失效。
- 对索引列运算（如，+、-、*、/），索引失效。
- 索引字段上使用（!= 或者 <>， not in）时，可能会导致索引失效。
- 索引字段上使用is null， is not null，可能导致索引失效。
- 左连接查询或者右连接查询查询关联的字段编码格式不一样，可能导致索引失效。
- MySQL优化器估计使用全表扫描要比使用索引快，则不使用索引。

31. 索引不适合哪些场景呢？

- 数据量比较少的表不适合加索引
- 更新比较频繁的字段也不适合加索引
- 离散低的字段不适合加索引（如性别）

32. 索引是不是建的越多越好呢？

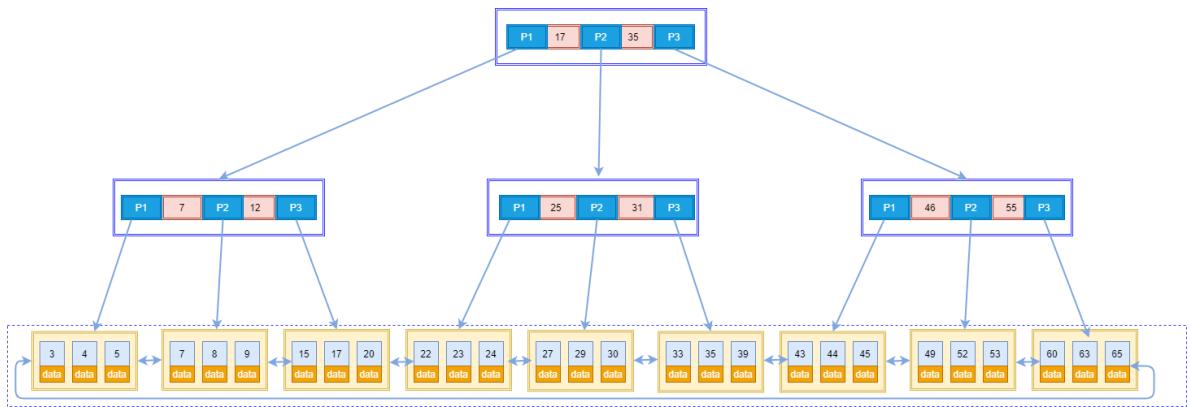
当然不是。

- 索引会占据磁盘空间
- 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。

33. MySQL索引用的什么数据结构了解吗？

MySQL的默认存储引擎是InnoDB，它采用的是B+树结构的索引。

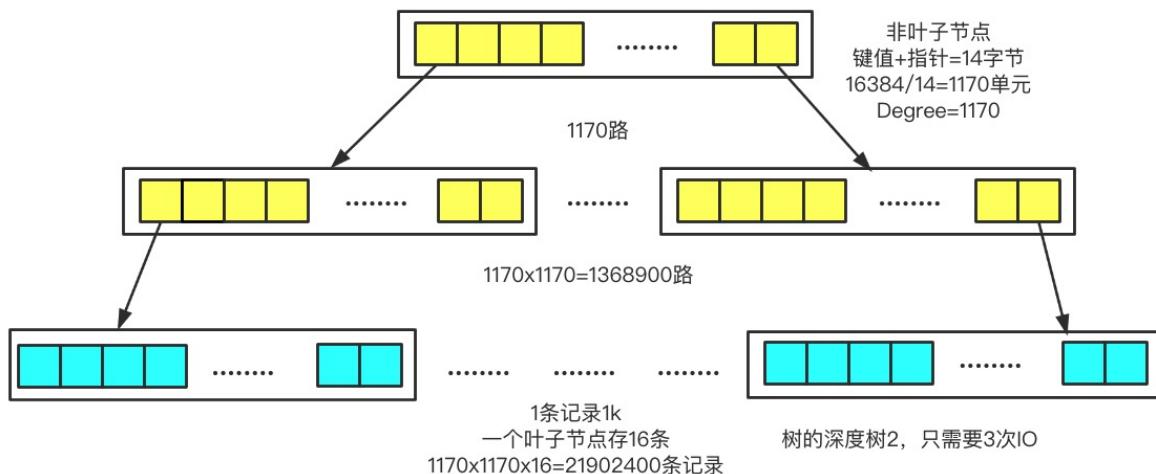
- B+树：只有叶子节点才会存储数据，非叶子节点只存储键值。叶子节点之间使用双向指针连接，最底层的叶子节点形成了一个双向有序链表。



在这张图里，有两个重点：

- 最外面的方块，我们称之为一个磁盘块，可以看到每个磁盘块包含几个数据项（粉色所示）和指针（黄色/灰色所示），如根节点磁盘包含数据项17和35，包含指针P1、P2、P3，P1表示小于17的磁盘块，P2表示在17和35之间的磁盘块，P3表示大于35的磁盘块。真实的数据存在于叶子节点即3、4、5.....、65。非叶子节点只存储真实的数据，只存储指引搜索方向的数据项，如17、35并不真实存在于数据表中。
- 叶子节点之间使用双向指针连接，最底层的叶子节点形成了一个双向有序链表，可以进行范围查询。

34.那一棵B+树能存储多少条数据呢？



假设索引字段是 bigint 类型，长度为 8 字节。指针大小在 InnoDB 源码中设置为 6 字节，这样一共 14 字节。非叶子节点(一页)可以存储 $16384/14=1170$ 个这样的单元(键值+指针)，代表有 1170 个指针。

树深度为 2 的时候，有 1170^2 个叶子节点，可以存储的数据为

$$1170 \times 1170 \times 16 = \mathbf{21902400}.$$

在查找数据时一次页的查找代表一次 IO，也就是说，一张 2000 万左右的表，查询数据最多需要访问 3 次磁盘。

所以在 InnoDB 中 B+ 树深度一般为 1-3 层，它就能满足千万级的数据存储。

35.为什么要用 B+ 树，而不用普通二叉树？

可以从几个维度去看这个问题，查询是否够快，效率是否稳定，存储数据多少，以及查找磁盘次数。

为什么不用普通二叉树？

普通二叉树存在退化的情况，如果它退化成链表，相当于全表扫描。平衡二叉树相比于二叉查找树来说，查找效率更稳定，总体的查找速度也更快。

为什么不用平衡二叉树呢？

读取数据的时候，是从磁盘读到内存。如果树这种数据结构作为索引，那每查找一次数据就需要从磁盘中读取一个节点，也就是一个磁盘块，但是平衡二叉树可是每个节点只存储一个键值和数据的，如果是 B+ 树，可以存储更多的节点数据，树的高度也会降低，因此读取磁盘的次数就降下来啦，查询效率就快。

36.为什么要用 B+ 树而不用 B 树呢？

B+相比较B树，有这些优势：

- 它是 B Tree 的变种，B Tree 能解决的问题，它都能解决。

B Tree 解决的两大问题：每个节点存储更多关键字；路数更多

- 扫库、扫表能力更强

如果我们要对表进行全表扫描，只需要遍历叶子节点就可以了，不需要遍历整棵 B+Tree 拿到所有的数据。

- B+Tree 的磁盘读写能力相对于 B Tree 来说更强，IO次数更少

根节点和枝节点不保存数据区，所以一个节点可以保存更多的关键字，一次磁盘加载的关键字更多，IO次数更少。

- 排序能力更强

因为叶子节点上有下一个数据区的指针，数据形成了链表。

- 效率更加稳定

B+Tree 永远是在叶子节点拿到数据，所以 IO 次数是稳定的。

37. Hash 索引和 B+ 树索引区别是什么？

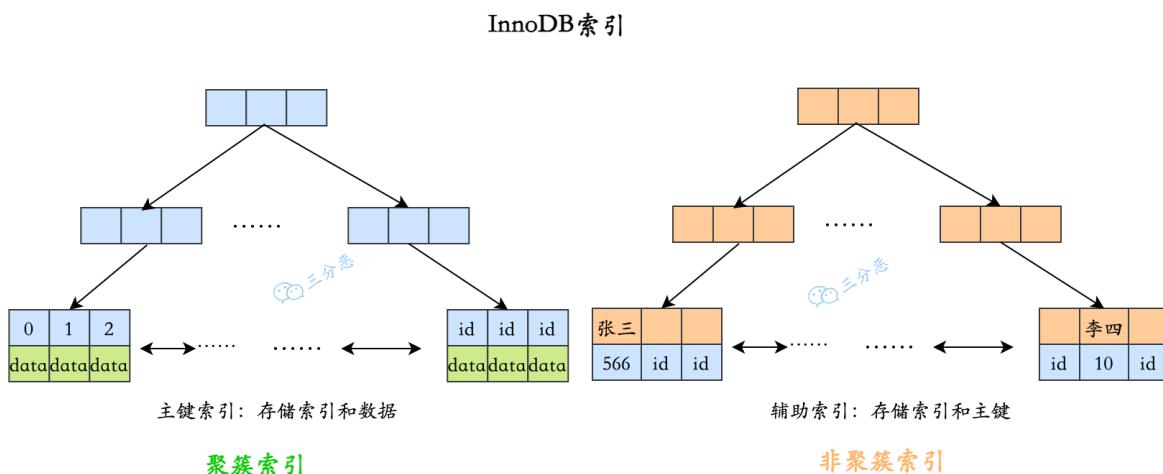
- B+ 树可以进行范围查询，Hash 索引不能。
- B+ 树支持联合索引的最左侧原则，Hash 索引不支持。
- B+ 树支持 order by 排序，Hash 索引不支持。
- Hash 索引在等值查询上比 B+ 树效率更高。
- B+ 树使用 like 进行模糊查询的时候，like 后面（比如 % 开头）的话可以起到优化的作用，Hash 索引根本无法进行模糊查询。

38. 聚簇索引与非聚簇索引的区别？

首先理解聚簇索引不是一种新的索引，而是而是一种 **数据存储方式**。聚簇表示数据行和相邻的键值紧凑地存储在一起。我们熟悉的两种存储引擎——MyISAM采用的是非聚簇索引，InnoDB采用的是聚簇索引。

可以这么说：

- 索引的数据结构是树，聚簇索引的索引和数据存储在一棵树上，树的叶子节点就是数据，非聚簇索引索引和数据不在一棵树上。

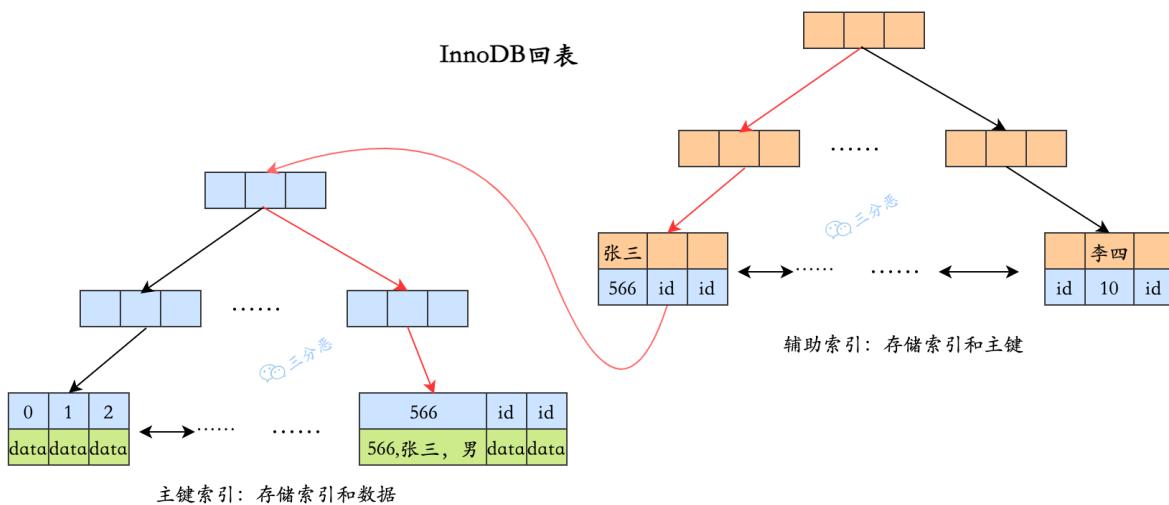


- 一个表中只能拥有一个聚簇索引，而非聚簇索引一个表可以存在多个。
- 聚簇索引，索引中键值的逻辑顺序决定了表中相应行的物理顺序；索引，索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同。
- 聚簇索引：物理存储按照索引排序；非聚集索引：物理存储不按照索引排序；

39.回表了解吗？

在InnoDB存储引擎里，利用辅助索引查询，先通过辅助索引找到主键索引的键值，再通过主键值查出主键索引里面没有符合要求的数据，它比基于主键索引的查询多扫描了一棵索引树，这个过程就叫回表。

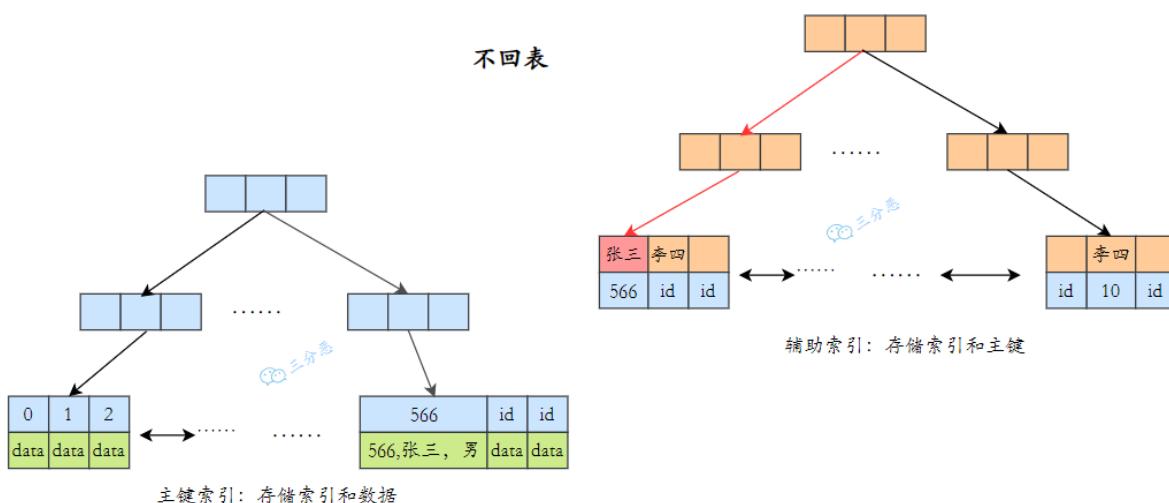
例如:select * from user where name = ‘张三’;



40.覆盖索引了解吗？

在辅助索引里面，不管是单列索引还是联合索引，如果 select 的数据列只用辅助索引中就能够取得，不用去查主键索引，这时候使用的索引就叫做覆盖索引，避免了回表。

比如，`select name from user where name = ‘张三’;`



41. 什么是最左前缀原则/最左匹配原则？

注意：最左前缀原则、最左匹配原则、最左前缀匹配原则这三个都是一个概念。

最左匹配原则：在InnoDB的联合索引中，查询的时候只有匹配了前一个/左边的值之后，才能匹配下一个。

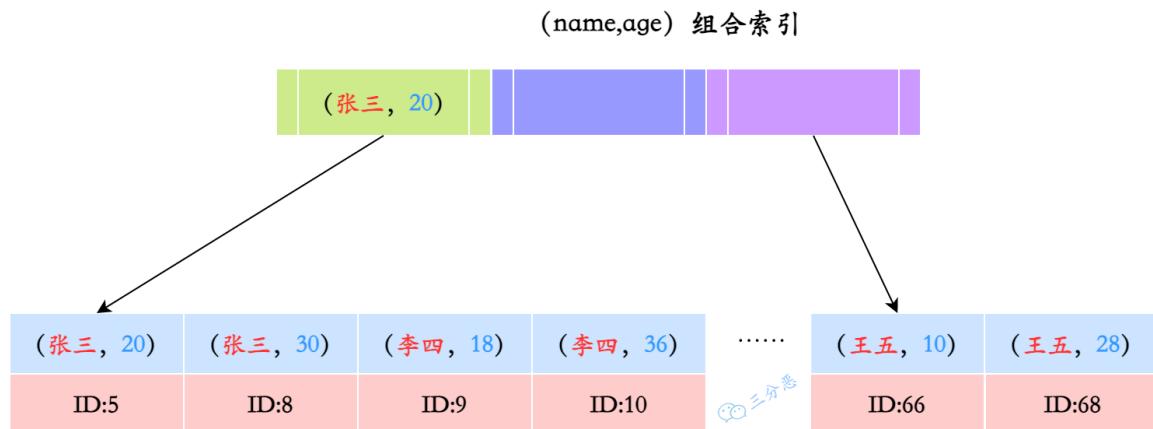
根据最左匹配原则，我们创建了一个组合索引，如 (a1,a2,a3)，相当于创建了 (a1)、(a1,a2) 和 (a1,a2,a3) 三个索引。

为什么不从最左开始查，就无法匹配呢？

比如有一个user表，我们给 name 和 age 建立了一个组合索引。

```
1 | ALTER TABLE user add INDEX comidx_name_phone (name,age);
```

组合索引在 B+Tree 中是复合的数据结构，它是按照从左到右的顺序来建立搜索树的 (name 在左边， age 在右边)。



从这张图可以看出来，name 是有序的，age 是无序的。当 name 相等的时候，age 才是有序的。

这个时候我们使用 `where name= ‘张三’ and age = ‘20 ’` 去查询数据的时候，B+Tree 会优先比较 name 来确定下一步应该搜索的方向，往左还是往右。如果 name 相同的时候再比较 age。但是如果查询条件没有 name，就不知道下一步应该查哪个节点，因为建立搜索树的时候 name 是第一个比较因子，所以就没用上索引。

42. 什么是索引下推优化？

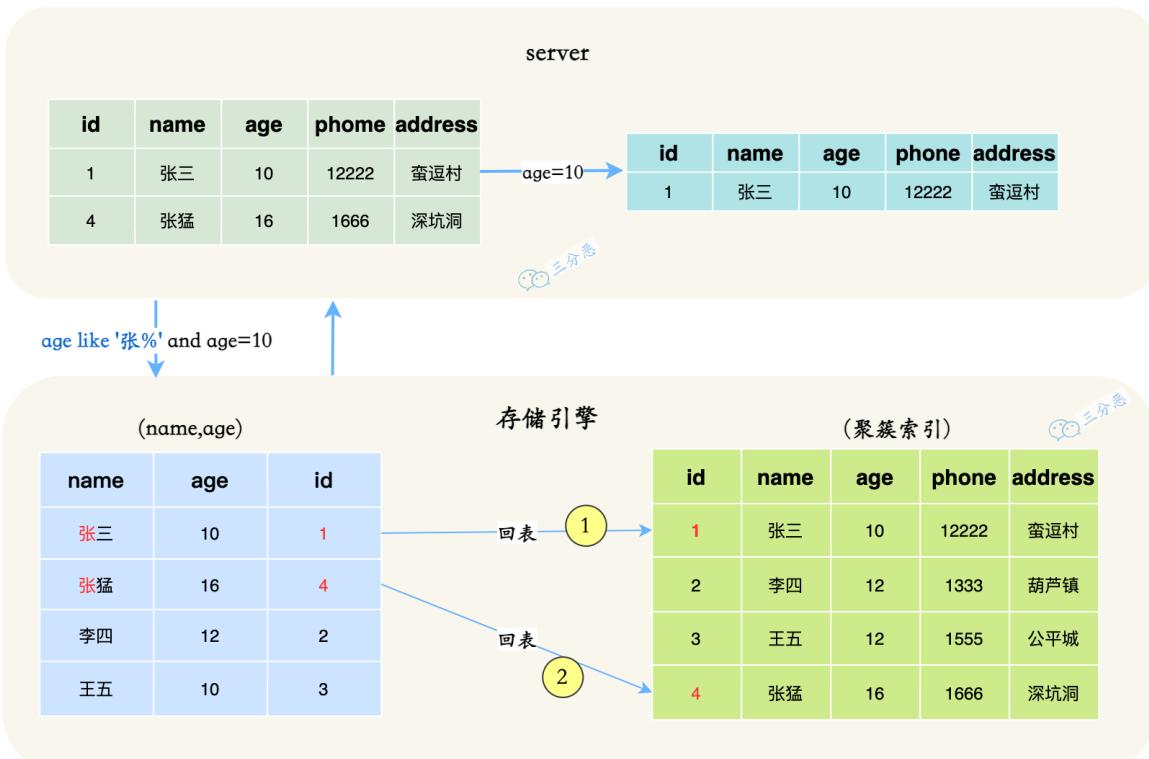
索引条件下推优化 (Index Condition Pushdown (ICP)) 是MySQL5.6添加的，用于优化数据查询。

- 不使用索引条件下推优化时存储引擎通过索引检索到数据，然后返回给MySQL Server，MySQL Server进行过滤条件的判断。
- 当使用索引条件下推优化时，如果存在某些被索引的列的判断条件时，MySQL Server将这一部分判断条件下推给存储引擎，然后由存储引擎通过判断索引是否符合MySQL Server传递的条件，只有当索引符合条件时才会将数据检索出来返回给MySQL服务器。

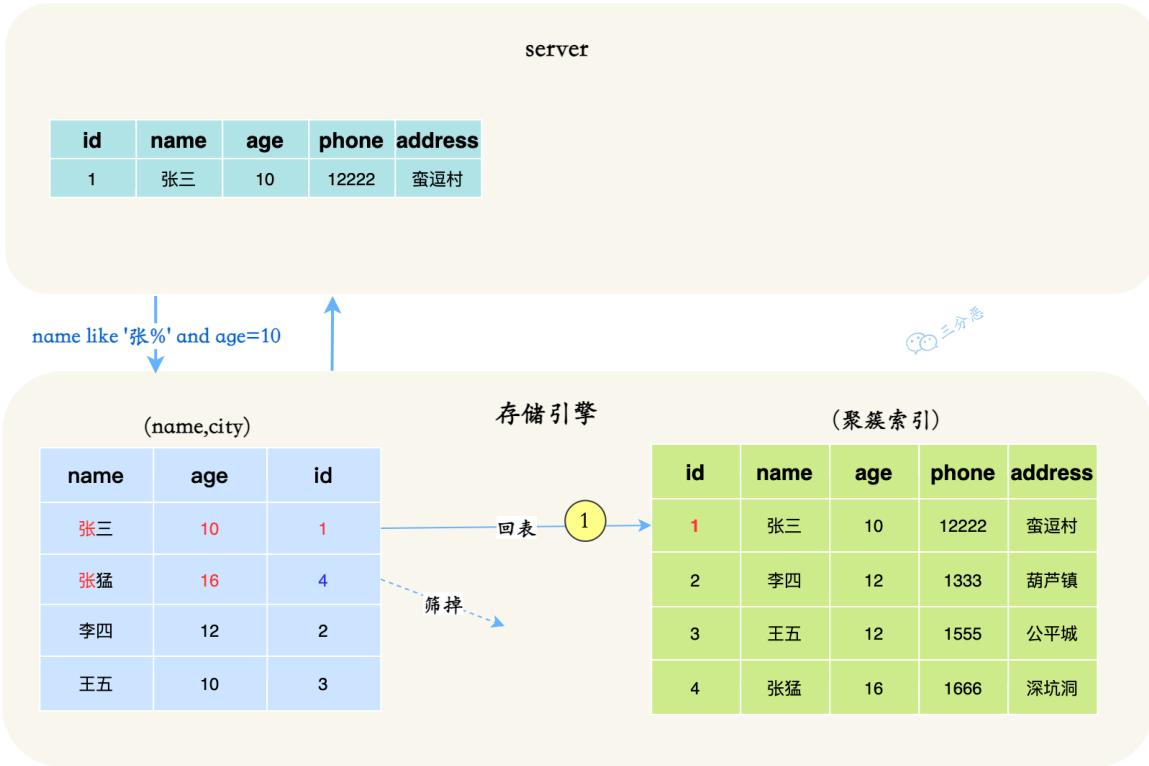
例如一张表，建了一个联合索引 (name, age)，查询语句：`select * from t_user where name like '张%' and age=10;`，由于 `name` 使用了范围查询，根据最左匹配原则：

不使用ICP，引擎层查找到 `name like '张%'` 的数据，再由Server层去过滤 `age=10` 这个条件，这样一来，就回表了两次，浪费了联合索引的另外一个字段 `age`。

没有使用ICP



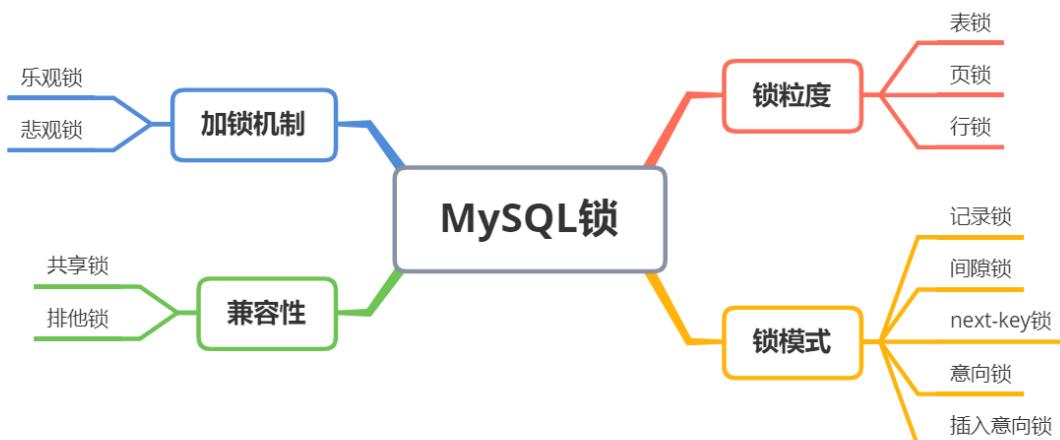
但是，使用了索引下推优化，把where的条件放到了引擎层执行，直接根据 `name like '张%' and age=10` 的条件进行过滤，减少了回表的次数。



索引条件下推优化可以减少存储引擎查询基础表的次数，也可以减少MySQL服务器从存储引擎接收数据的次数。

锁

43.MySQL中有哪几种锁，列举一下？



如果按锁粒度划分，有以下3种：

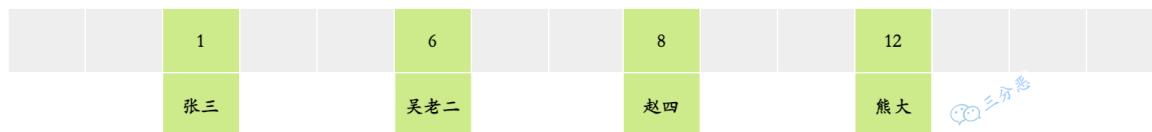
- 表锁：开销小，加锁快；锁定力度大，发生锁冲突概率高，并发度最低；不会出现死锁。
- 行锁：开销大，加锁慢；会出现死锁；锁定粒度小，发生锁冲突的概率低，并发度高。
- 页锁：开销和加锁速度介于表锁和行锁之间；会出现死锁；锁定粒度介于表锁和行锁之间，并发度一般

如果按照兼容性，有两种，

- 共享锁（S Lock），也叫读锁（read lock），相互不阻塞。
- 排他锁（X Lock），也叫写锁（write lock），排它锁是阻塞的，在一定时间内，只有一个请求能执行写入，并阻止其它锁读取正在写入的数据。

44. 说说InnoDB里的行锁实现？

我们拿这么一个用户表来表示行级锁，其中插入了4行数据，主键值分别是1,6,8,12，现在简化它的聚簇索引结构，只保留数据记录。

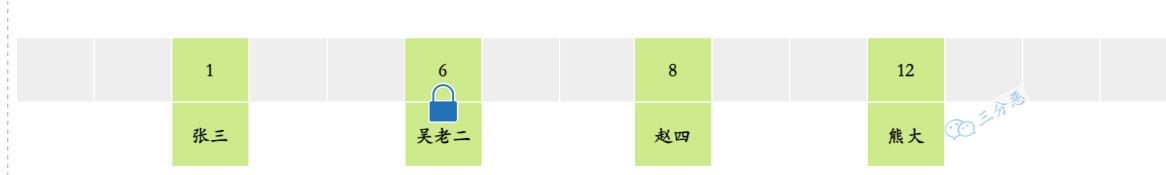


InnoDB的行锁的主要实现如下：

- Record Lock 记录锁

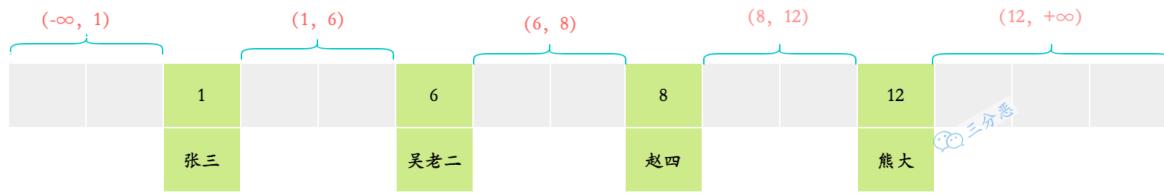
记录锁就是直接锁定某行记录。当我们使用唯一性的索引(包括唯一索引和聚簇索引)进行等值查询且精准匹配到一条记录时，此时就会直接将这条记录锁定。例如

```
select * from t where id = 6 for update; 就会将 id=6 的记录锁定。
```



- Gap Lock 间隙锁

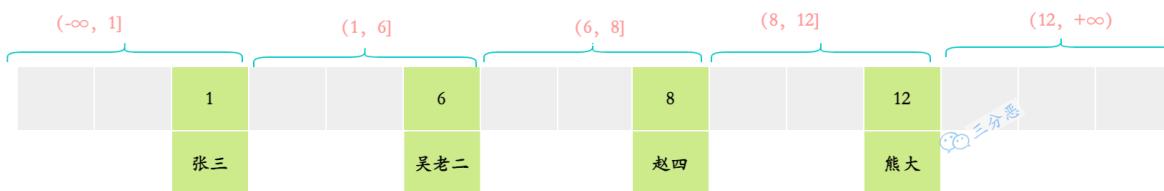
间隙锁(Gap Locks)的间隙指的是两个记录之间逻辑上尚未填入数据的部分，是一个左开右开空间。



间隙锁就是锁定某些间隙区间的。当我们使用用等值查询或者范围查询，并且没有命中任何一个 `record`，此时就会将对应的间隙区间锁定。例如 `select * from t where id = 3 for update;` 或者 `select * from t where id > 1 and id < 6 for update;` 就会将(1,6]区间锁定。

- Next-key Lock 临键锁

临键指的是间隙加上它右边的记录组成的左开右闭区间。比如上述的(1,6]、(6,8]等。



临键锁就是记录锁(Record Locks)和间隙锁(Gap Locks)的结合，即除了锁住记录本身，还要再锁住索引之间的间隙。当我们使用范围查询，并且命中了部分 `record` 记录，此时锁住的就是临键区间。注意，临键锁锁住的区间会包含最后一个record的右边的临键区间。例如 `select * from t where id > 5 and id <= 7 for update;` 会锁住(4,7]、(7,+∞)。mysql默认行锁类型就是 临键锁(Next-Key Locks)。当使用唯一性索引，等值查询匹配到一条记录的时候，临键锁(Next-Key Locks)会退化成记录锁；没有匹配到任何记录的时候，退化成间隙锁。

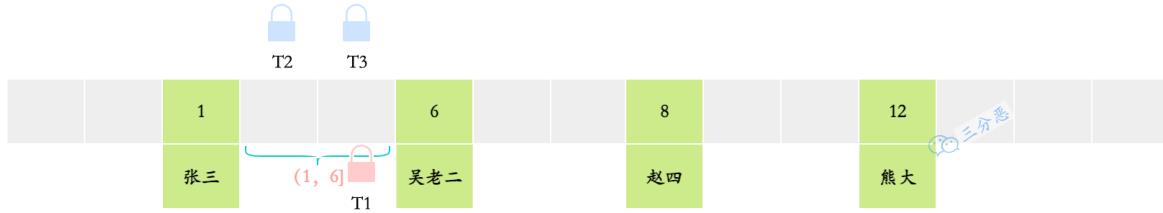
间隙锁(Gap Locks) 和 临键锁(Next-Key Locks) 都是用来解决幻读问题的，在 已提交读 (READ COMMITTED) 隔离级别下， 间隙锁(Gap Locks) 和 临键锁(Next-Key Locks) 都会失效！

上面是行锁的三种实现算法，除此之外，在行上还存在插入意向锁。

- Insert Intention Lock 插入意向锁

一个事务在插入一条记录时需要判断一下插入位置是不是被别的事务加了意向锁，如果有的话，插入操作需要等待，直到拥有 gap 锁的那个事务提交。但是事务在等待的时候也需要在内存中生成一个锁结构，表明有事务想在某个 间隙 中插入新记录，但是现在在等待。这种类型的锁命名为 Insert Intention Locks，也就是插入意向锁。

假如我们有个T1事务，给(1,6)区间加上了意向锁，现在有个T2事务，要插入一个数据，id为4，它会获取一个(1,6)区间的插入意向锁，又有有个T3事务，想要插入一个数据，id为3，它也会获取一个(1,6)区间的插入意向锁，但是，这两个插入意向锁锁不会互斥。



45. 意向锁是什么知道吗？

意向锁是一个表级锁，不要和插入意向锁搞混。

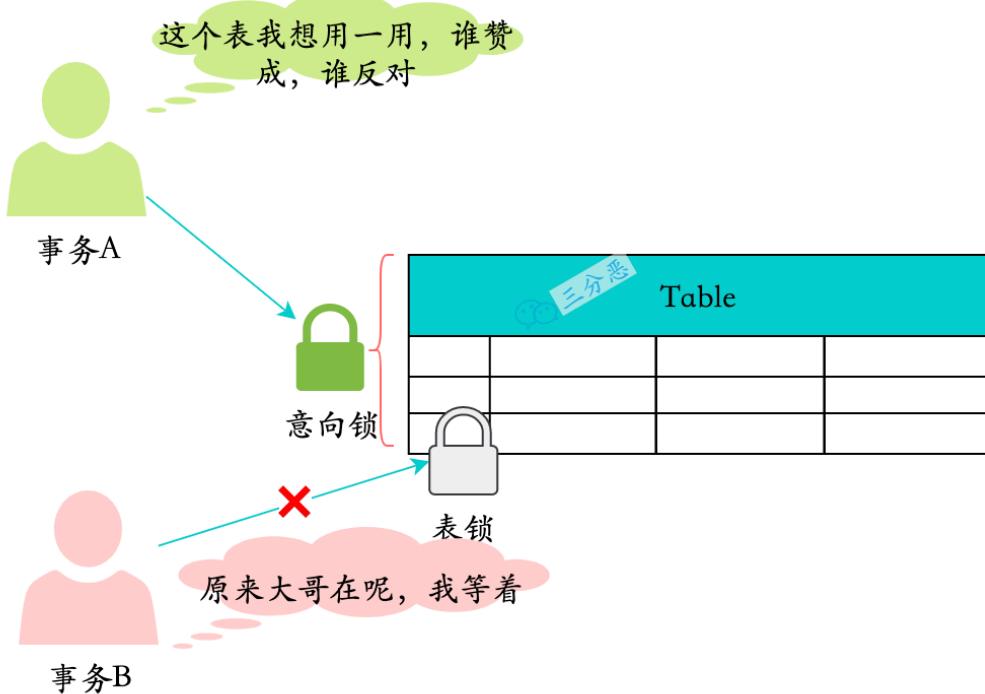
意向锁的出现是为了支持InnoDB的多粒度锁，它解决的是表锁和行锁共存的问题。

当我们需要给一个表加表锁的时候，我们需要根据去判断表中有没有数据行被锁定，以确定是否能加成功。

假如没有意向锁，那么我们就得遍历表中所有数据行来判断有没有行锁；

有了意向锁这个表级锁之后，则我们直接判断一次就知道表中是否有数据行被锁定了。

有了意向锁之后，要执行的事务A在申请行锁（写锁）之前，数据库会自动先给事务A申请表的意向排他锁。当事务B去申请表的互斥锁时就会失败，因为表上有意向排他锁之后事务B申请表的互斥锁时会被阻塞。



46.MySQL的乐观锁和悲观锁了解吗？

- 悲观锁（Pessimistic Concurrency Control）：

悲观锁认为被它保护的数据是极其不安全的，每时每刻都有可能被改动，一个事务拿到悲观锁后，其他任何事务都不能对该数据进行修改，只能等待锁被释放才可以执行。

数据库中的行锁，表锁，读锁，写锁均为悲观锁。

- 乐观锁（Optimistic Concurrency Control）

乐观锁认为数据的变动不会太频繁。

乐观锁通常是通过在表中增加一个版本(version)或时间戳(timestamp)来实现，其中，版本最为常用。

事务在从数据库中取数据时，会将该数据的版本也取出来(v1)，当事务对数据变动完毕想要将其更新到表中时，会将之前取出的版本v1与数据中最新的版本v2相对比，如果v1=v2，那么说明在数据变动期间，没有其他事务对数据进行修改，此时，就允许事务对表中的数据进行修改，并且修改时version会加1，以此来表明数据已被变动。

如果，v1不等于v2，那么说明数据变动期间，数据被其他事务改动了，此时不允许数据更新到表中，一般的处理办法是通知用户让其重新操作。不同于悲观锁，乐观锁通常是由开发者实现的。

47.MySQL 遇到过死锁问题吗，你是如何解决的？

排查死锁的一般步骤是这样的：

- (1) 查看死锁日志 `show engine innodb status;`
- (2) 找出死锁 sql
- (3) 分析 sql 加锁情况
- (4) 模拟死锁案发
- (5) 分析死锁日志
- (6) 分析死锁结果

当然，这只是一个简单的流程说明，实际上生产中的死锁千奇百怪，排查和解决起来没那么简

事务

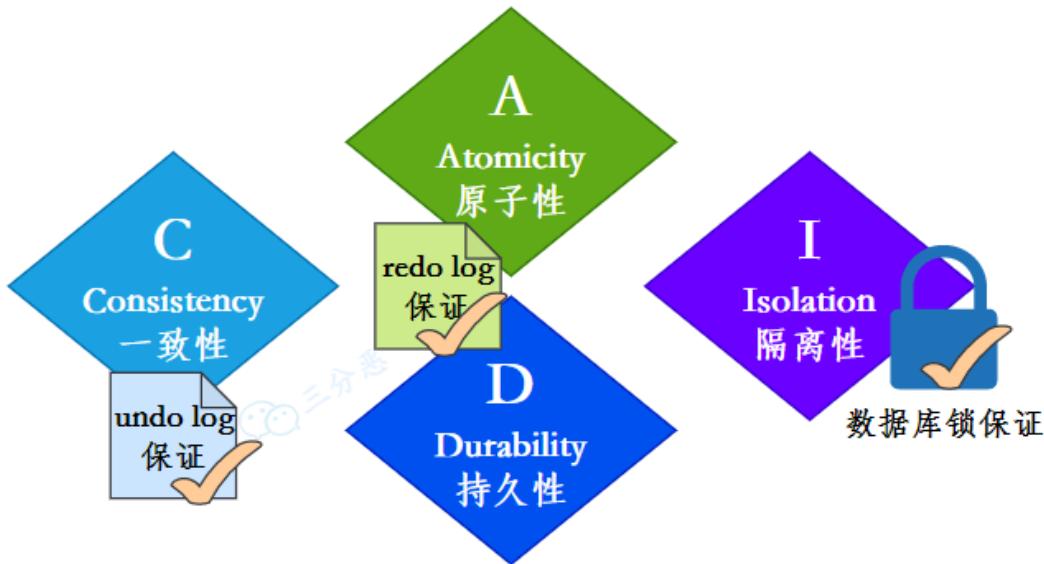
48.MySQL 事务的四大特性说一下？



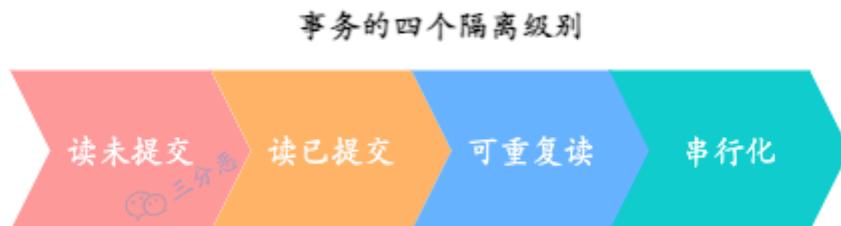
- 原子性：事务作为一个整体被执行，包含在其中的对数据库的操作要么全部被执行，要么都不执行。
- 一致性：指在事务开始之前和事务结束以后，数据不会被破坏，假如 A 账户给 B 账户转 10 块钱，不管成功与否，A 和 B 的总金额是不变的。
- 隔离性：多个事务并发访问时，事务之间是相互隔离的，即一个事务不影响其它事务运行效果。简言之，就是事务之间是进水不犯河水的。
- 持久性：表示事务完成以后，该事务对数据库所作的操作更改，将持久地保存在数据库之中。

49.那ACID靠什么保证的呢？

- 事务的 隔离性 是通过数据库锁的机制实现的。
- 事务的一致性 由undo log来保证： undo log是逻辑日志，记录了事务的insert、update、deltete操作，回滚的时候做相反的delete、update、insert操作来恢复数据。
- 事务的 原子性 和 持久性 由redo log来保证： redolog被称作重做日志，是物理日志，事务提交的时候，必须先将事务的所有日志写入redo log持久化，到事务的提交操作才算完成。



50. 事务的隔离级别有哪些？MySQL 的默认隔离级别是什么？



- 读未提交 (Read Uncommitted)
- 读已提交 (Read Committed)
- 可重复读 (Repeatable Read)
- 串行化 (Serializable)

MySQL默认的事务隔离级别是可重复读 (Repeatable Read)。

51. 什么是幻读，脏读，不可重复读呢？

- 事务 A、B 交替执行，事务 A 读取到事务 B 未提交的数据，这就是 脏读 。
- 在一个事务范围内，两个相同的查询，读取同一条记录，却返回了不同的数据，这就是 不可重复读 。
- 事务 A 查询一个范围的结果集，另一个并发事务 B 往这个范围中插入 / 删除了数据，并静悄悄地提交，然后事务 A 再次查询相同的范围，两次读取得到的结果集不一样了，这就是 幻读 。

不同的隔离级别，在并发事务下可能会发生的问题：

隔离级别	脏读	不可重复读	幻读
Read Uncommitted 读取未提交	是	是	是
Read Committed 读取已提交	否	是	否
Repeatable Read 可重复读	否	否	是
Serializable 可串行化	否	否	否

52. 事务的各个隔离级别都是如何实现的？

读未提交

读未提交，就不用多说了，采取的是读不加锁原理。

- 事务读不加锁，不阻塞其他事务的读和写
- 事务写阻塞其他事务写，但不阻塞其他事务读；

读取已提交&可重复读

读取已提交和可重复读级别利用了 **ReadView** 和 **MVCC**，也就是每个事务只能读取它能看到的版本（ReadView）。

- READ COMMITTED：每次读取数据前都生成一个ReadView
- REPEATABLE READ：在第一次读取数据时生成一个ReadView

串行化

串行化的实现采用的是读写都加锁的原理。

串行化的情况下，对于同一行事务，**写**会加**写锁**，**读**会加**读锁**。当出现读写锁冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行。

53. MVCC了解吗？怎么实现的？

MVCC(Multi Version Concurrency Control)，中文名是多版本并发控制，简单来说就是通过维护数据历史版本，从而解决并发访问情况下的读一致性问题。关于它的实现，要抓住几个关键点，**隐式字段**、**undo日志**、**版本链**、**快照读&当前读**、**Read View**。

版本链

对于InnoDB存储引擎，每一行记录都有两个隐藏列**DB_TRX_ID**、**DB_ROLL_PTR**

- **DB_TRX_ID**，事务ID，每次修改时，都会把该事务ID复制给 **DB_TRX_ID**；
- **DB_ROLL_PTR**，回滚指针，指向回滚段的undo日志。

column	column	column	DB_TRX_ID	DB_ROLL_PTR
			80	



假如有一张 **user** 表，表中只有一行记录，当时插入的事务id为80。此时，该条记录的示例图如下：

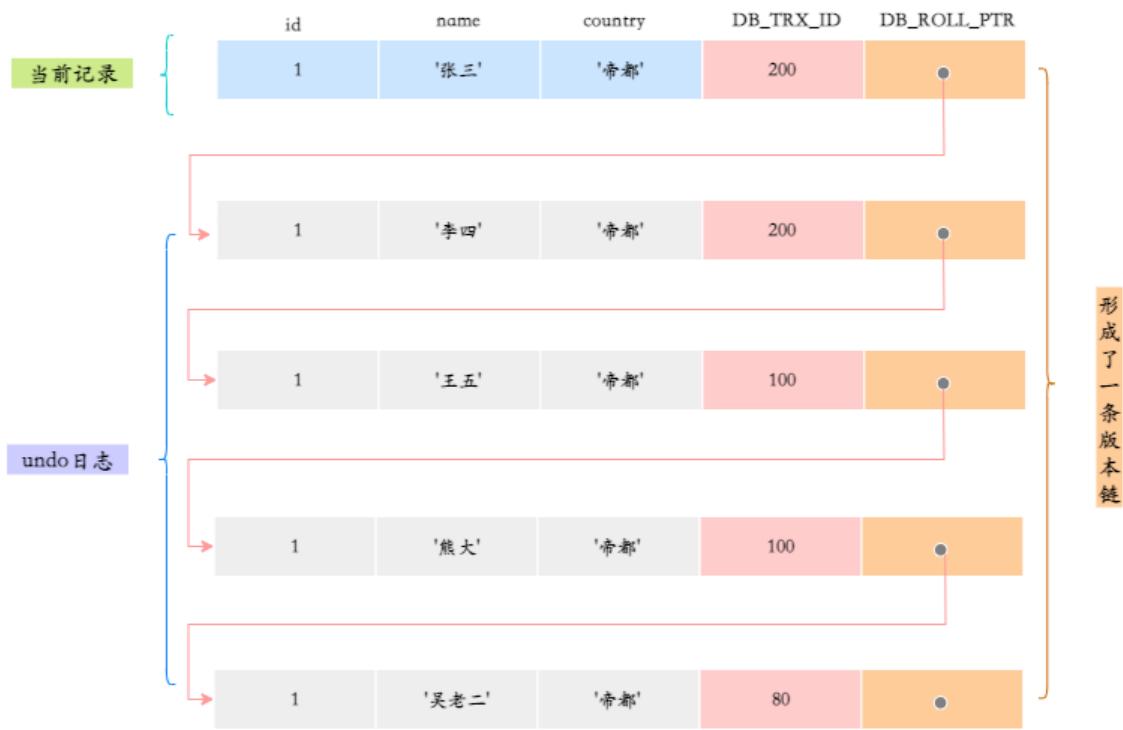
id	name	city	DB_TRX_ID	DB_ROLL_PTR
1	'张三'	'帝都'	80	

A red arrow points from the DB_ROLL_PTR column in the table to a green box labeled "insert undo".

接下来有两个 **DB_TRX_ID** 分别为 **100**、**200** 的事务对这条记录进行 **update** 操作，整个过程如下：

发生时间编号	TRX 100	TRX 200
①	BEGIN	
②		BEGIN
③		
④	UPDATE hero SET name='李四' WHERE id=1	
⑤	UPDATE hero SET name='王五' WHERE id=1	
⑥	COMMIT	UPDATE hero SET name='熊大' WHERE id=1
⑦		UPDATE hero SET name='吴老二' WHERE id=1
⑧		COMMIT

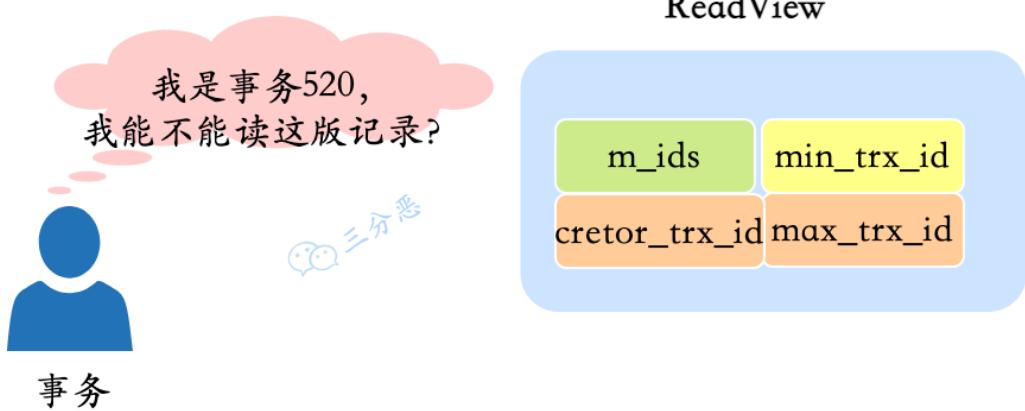
由于每次变动都会先把 **undo** 日志记录下来，并用 **DB_ROLL_PTR** 指向 **undo** 日志地址。因此可以认为，**对该条记录的修改日志串联起来就形成了一个版本链**，**版本链的头节点就是当前记录最新的值**。如下：



ReadView

对于 **Read Committed** 和 **Repeatable Read** 隔离级别来说，都需要读取已经提交的事务所修改的记录，也就是说如果版本链中某个版本的修改没有提交，那么该版本的记录时不能被读取的。所以需要确定在 **Read Committed** 和 **Repeatable Read** 隔离级别下，版本链中哪个版本是能被当前事务读取的。于是就引入了 **ReadView** 这个概念来解决这个问题。

Read View就是事务执行**快照读**时，产生的读视图，相当于某时刻表记录的一个快照，通过这个快照，我们可以获取：



- `m_ids`：表示在生成 ReadView 时当前系统中活跃的读写事务的事务id 列表。
- `min_trx_id`：表示在生成 ReadView 时当前系统中活跃的读写事务中最小的事务 id，也就是 `m_ids` 中的最小值。
- `max_trx_id`：表示生成 ReadView 时系统中应该分配给下一个事务的 id 值。
- `creator_trx_id`：表示生成该 ReadView 的事务的 事务id

有了这个 ReadView，这样在访问某条记录时，只需要按照下边的步骤判断记录的某个版本是否可见：

- 如果被访问版本的 `DB_TRX_ID` 属性值与 ReadView 中的 `creator_trx_id` 值相同，意味着当前事务在访问它自己修改过的记录，所以该版本可以被当前事务访问。
- 如果被访问版本的 `DB_TRX_ID` 属性值小于 ReadView 中的 `min_trx_id` 值，表明生成该版本的事务在当前事务生成 ReadView 前已经提交，所以该版本可以被当前事务访问。
- 如果被访问版本的 `DB_TRX_ID` 属性值大于 ReadView 中的 `max_trx_id` 值，表明生成该版本的事务在当前事务生成 ReadView 后才开启，所以该版本不可以被当前事务访问。
- 如果被访问版本的 `DB_TRX_ID` 属性值在 ReadView 的 `min_trx_id` 和 `max_trx_id` 之间，那就需要判断一下 `trx_id` 属性值是不是在 `m_ids` 列表中，如果在，说明创建 ReadView 时生成该版本的事务还是活跃的，该版本不可以被访问；如果不 在，说明创建 ReadView 时生成该版本的事务已经被提交，该版本可以被访问。

如果某个版本的数据对当前事务不可见的话，那就顺着版本链找到下一个版本的数据，继续按照上边的步骤判断可见性，依此类推，直到版本链中的最后一个版本。如果最后一个版本也不可见的话，那么就意味着该条记录对该事务完全不可见，查询结果就不包含该记录。

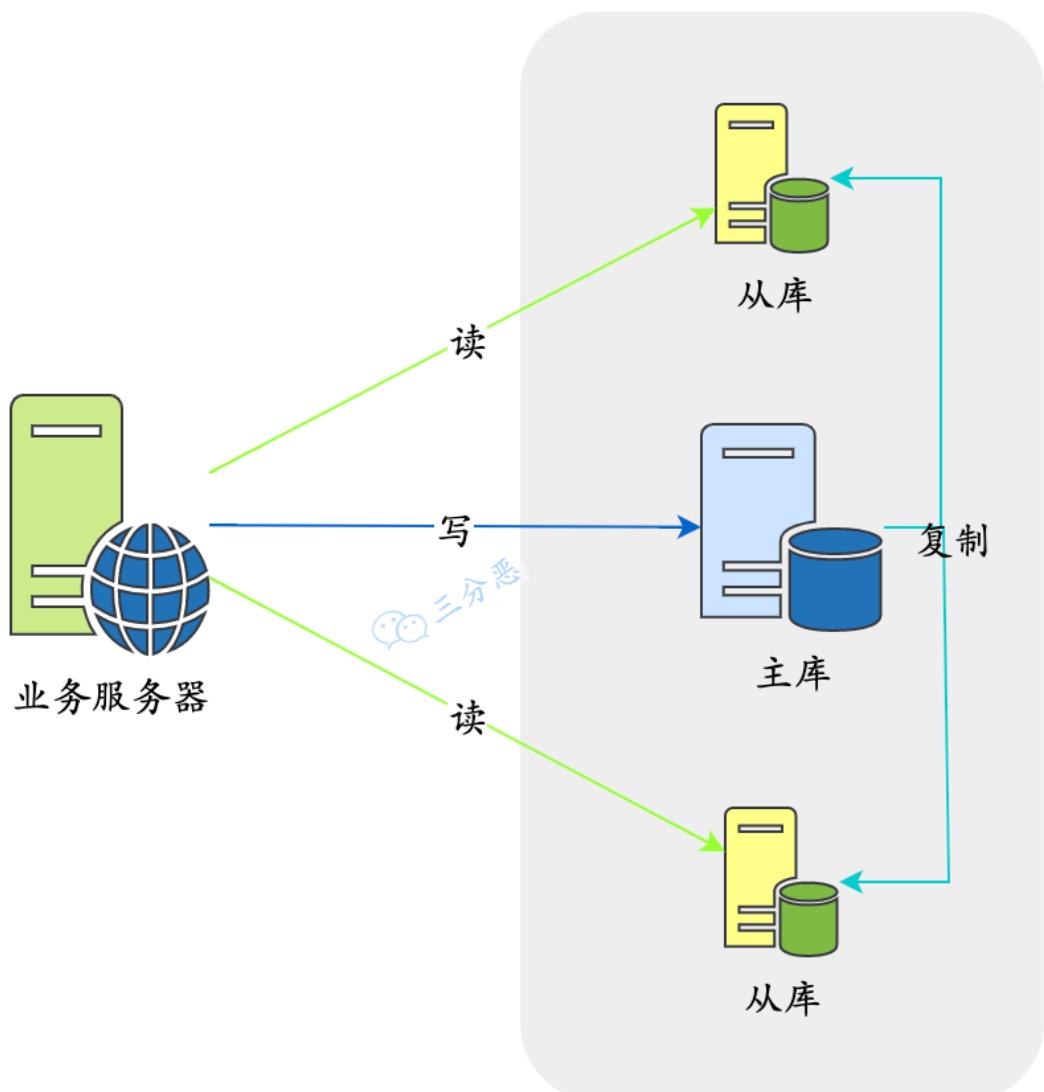
在 MySQL 中， READ COMMITTED 和 REPEATABLE READ 隔离级别的一个非常大的区别就是它们生成ReadView的时机不同。

READ COMMITTED 是每次读取数据前都生成一个**ReadView**，这样就能保证自己每次都能读到其它事务提交的数据； REPEATABLE READ 是在第一次读取数据时生成一个**ReadView**，这样就能保证后续读取的结果完全一致。

高可用/性能

54.数据库读写分离了解吗？

读写分离的基本原理是将数据库读写操作分散到不同的节点上，下面是基本架构图：



读写分离的基本实现是：

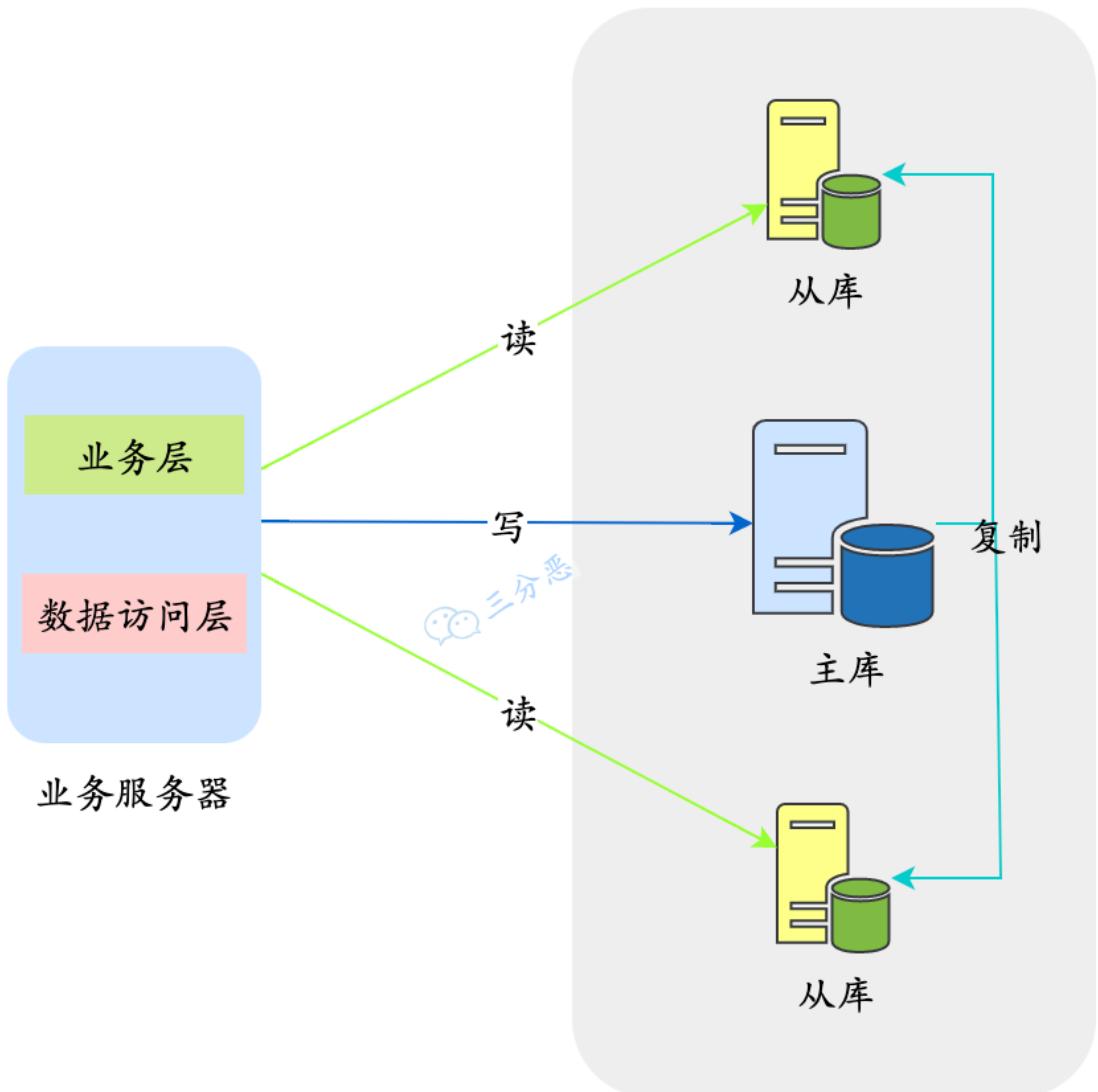
- 数据库服务器搭建主从集群，一主一从、一主多从都可以。
- 数据库主机负责读写操作，从机只负责读操作。
- 数据库主机通过复制将数据同步到从机，每台数据库服务器都存储了所有的业务数据。
- 业务服务器将写操作发给数据库主机，将读操作发给数据库从机。

55.那读写分离的分配怎么实现呢？

将读写操作区分开来，然后访问不同的数据库服务器，一般有两种方式：程序代码封装和中间件封装。

1. 程序代码封装

程序代码封装指在代码中抽象一个数据访问层（所以有的文章也称这种方式为 "中间层封装"），实现读写操作分离和数据库服务器连接的管理。例如，基于 Hibernate 进行简单封装，就可以实现读写分离：



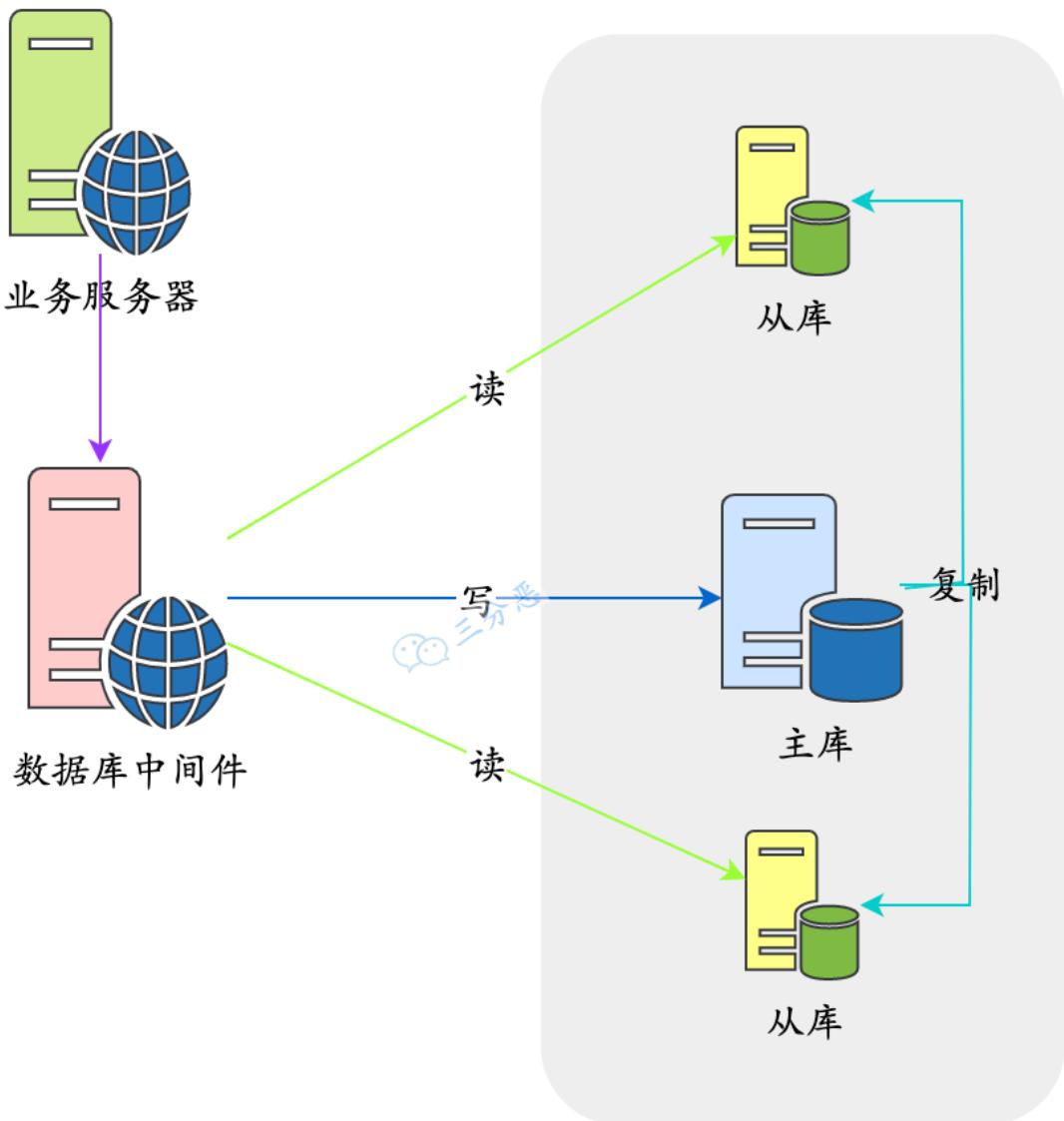
目前开源的实现方案中，淘宝的 TDDL (Taobao Distributed Data Layer, 外号：首都大了) 是比较有名的。

2. 中间件封装

中间件封装指的是独立一套系统出来，实现读写操作分离和数据库服务器连接的管理。中间件对业务服务器提供 SQL 兼容的协议，业务服务器无须自己进行读写分离。

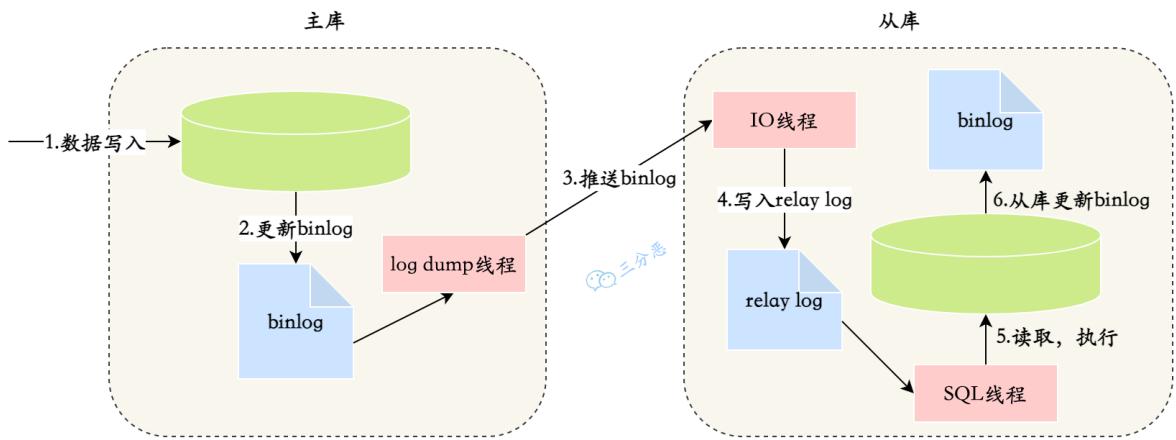
对于业务服务器来说，访问中间件和访问数据库没有区别，事实上在业务服务器看来，中间件就是一个数据库服务器。

其基本架构是：



56. 主从复制原理了解吗？

- master数据写入，更新binlog
- master创建一个dump线程向slave推送binlog
- slave连接到master的时候，会创建一个IO线程接收binlog，并记录到relay log中继日志中
- slave再开启一个sql线程读取relay log事件并在slave执行，完成同步
- slave记录自己的binglog



57. 主从同步延迟怎么处理？

主从同步延迟的原因

一个服务器开放 N 个链接给客户端来连接的，这样有会有大并发的更新操作，但是从服务器的里面读取 binlog 的线程仅有一个，当某个 SQL 在从服务器上执行的时间稍长 或者由于某个 SQL 要进行锁表就会导致，主服务器的 SQL 大量积压，未被同步到从服务器里。这就导致了主从不一致， 也就是主从延迟。

主从同步延迟的解决办法

解决主从复制延迟有几种常见的方法：

1. 写操作后的读操作指定发给数据库主服务器

例如，注册账号完成后，登录时读取账号的读操作也发给数据库主服务器。这种方式和业务强绑定，对业务的侵入和影响较大，如果哪个新来的程序员不知道这样写代码，就会导致一个bug。

2. 读从机失败后再读一次主机

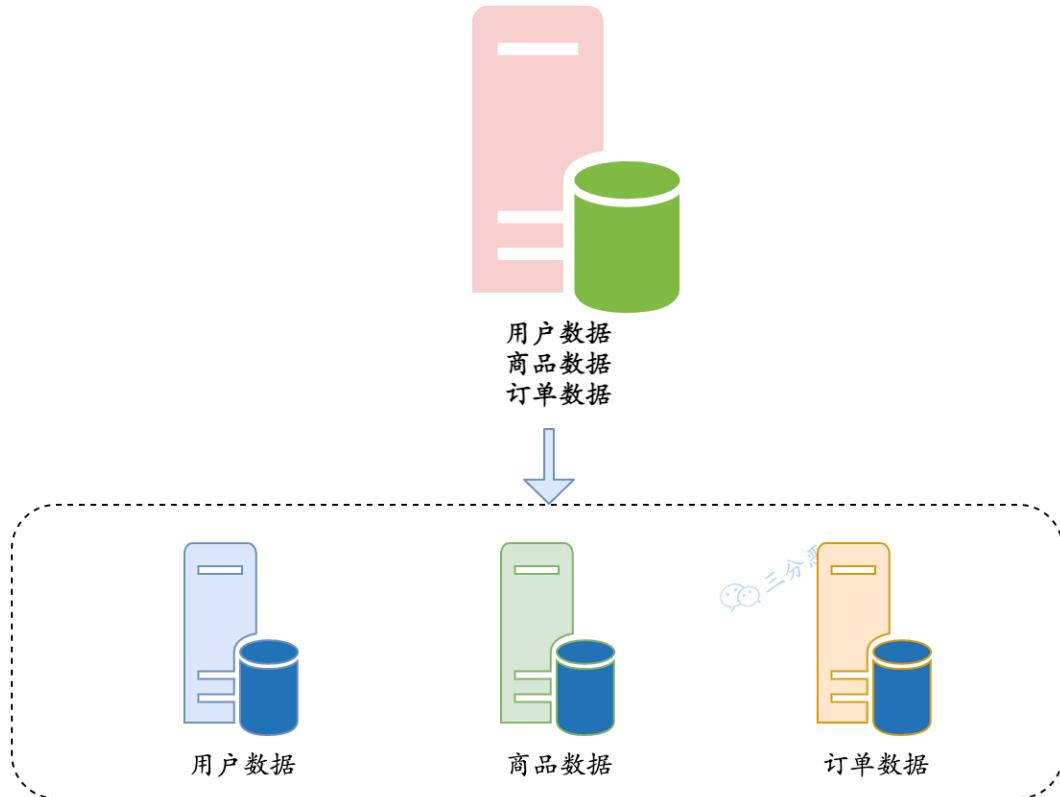
这就是通常所说的 "二次读取"，二次读取和业务无绑定，只需要对底层数据库访问的 API 进行封装即可，实现代价较小，不足之处在于如果有很多二次读取，将大大增加主机的读操作压力。例如，黑客暴力破解账号，会导致大量的二次读取操作，主机可能顶不住读操作的压力从而崩溃。

3. 关键业务读写操作全部指向主机，非关键业务采用读写分离

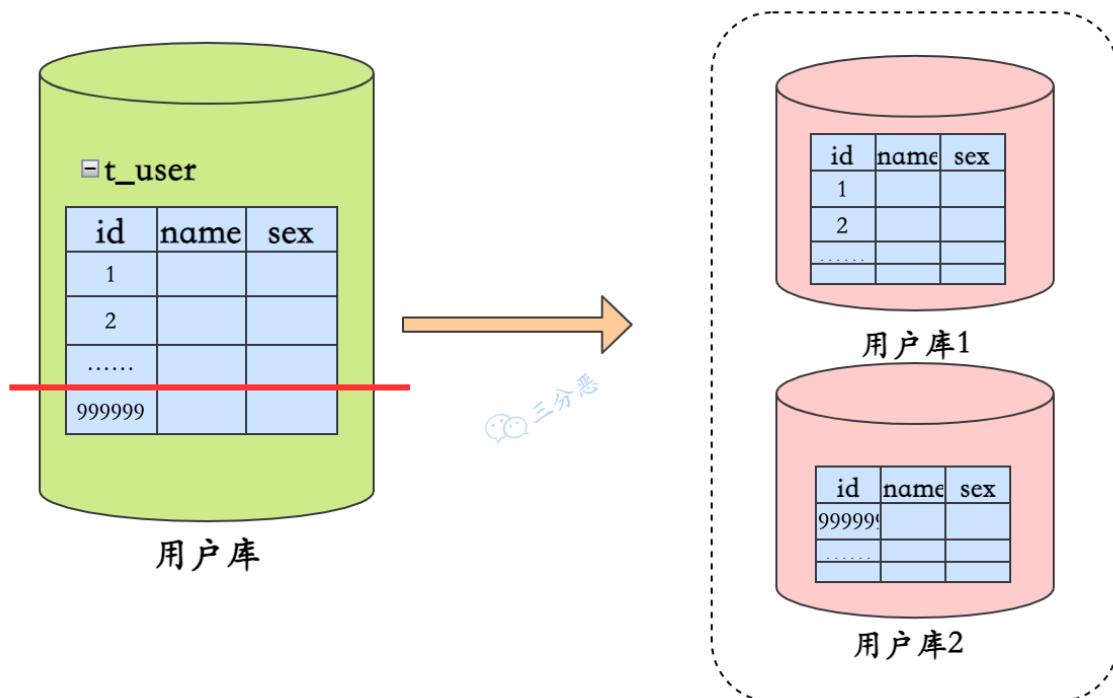
例如，对于一个用户管理系统来说，注册 + 登录的业务读写操作全部访问主机，用户的介绍、爱好、等级等业务，可以采用读写分离，因为即使用户改了自己的自我介绍，在查询时却看到了自我介绍还是旧的，业务影响与不能登录相比就小很多，还可以忍受。

58.你们一般是怎么分库的呢？

- 垂直分库：以表为依据，按照业务归属不同，将不同的表拆分到不同的库中。

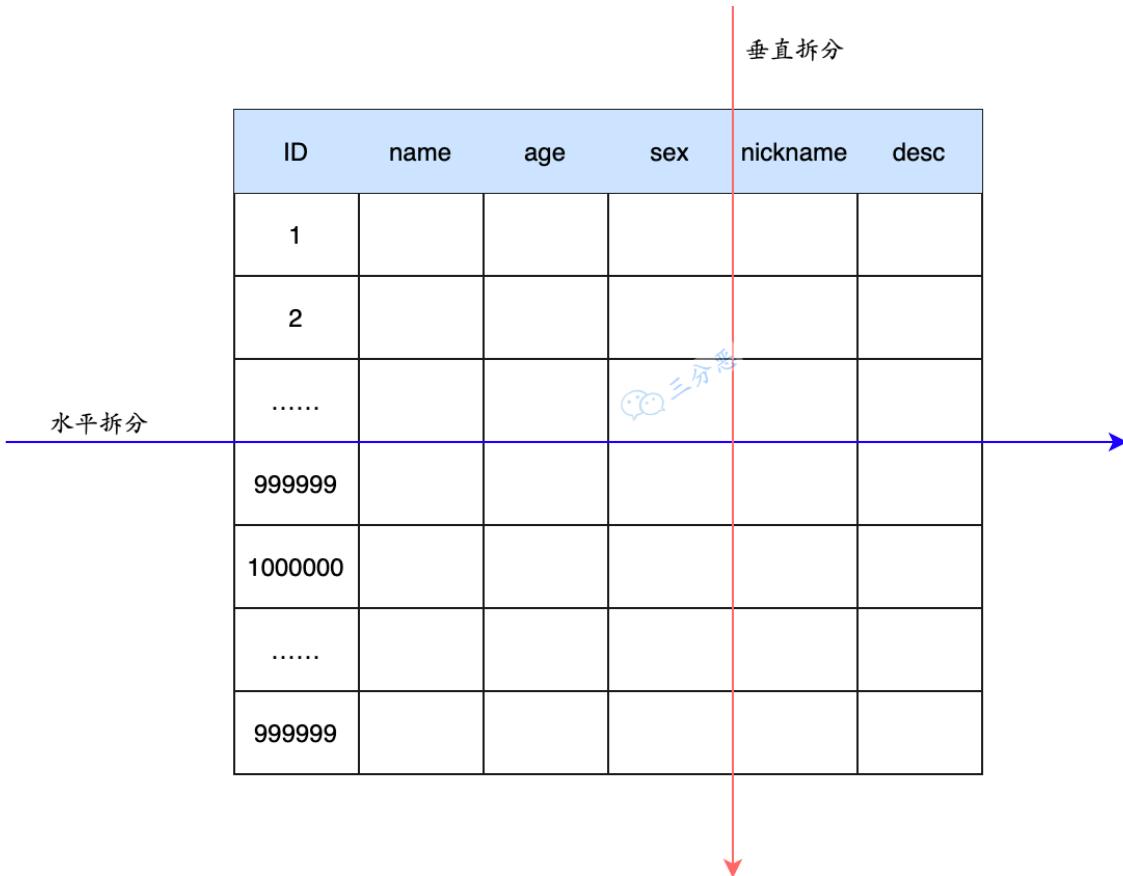


- 水平分库：以字段为依据，按照一定策略（hash、range 等），将一个库中的数据拆分到多个库中。



59.那你们是怎么分表的？

- 水平分表：以字段为依据，按照一定策略（hash、range 等），将一个表中的数据拆分到多个表中。
- 垂直分表：以字段为依据，按照字段的活跃性，将表中字段拆到不同的表（主表和扩展表）中。



60.水平分表有哪几种路由方式？

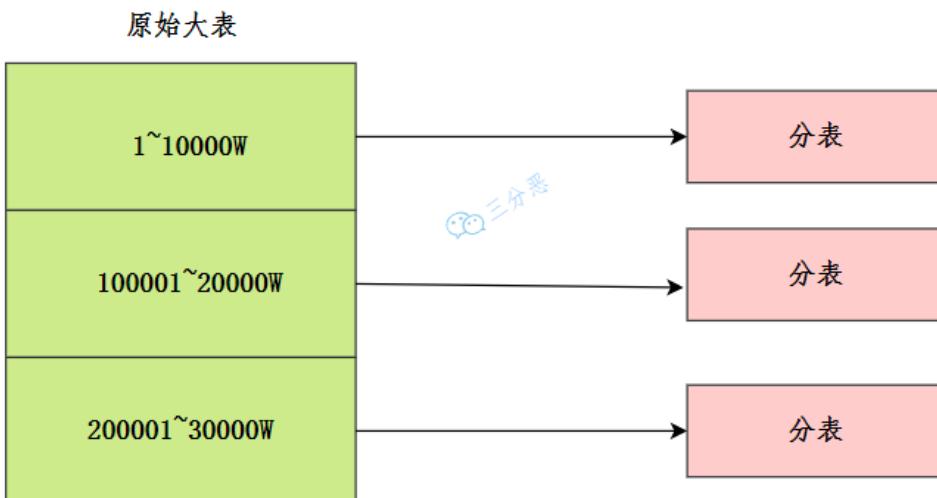
什么是路由呢？就是数据应该分到哪一张表。

水平分表主要有三种路由方式：

- 范围路由：选取有序的数据列（例如，整形、时间戳等）作为路由的条件，不同分段分散到不同的数据库表中。

我们可以观察一些支付系统，发现只能查一年范围内的支付记录，这个可能就是支付公司按照时间进行了分表。

范围路由



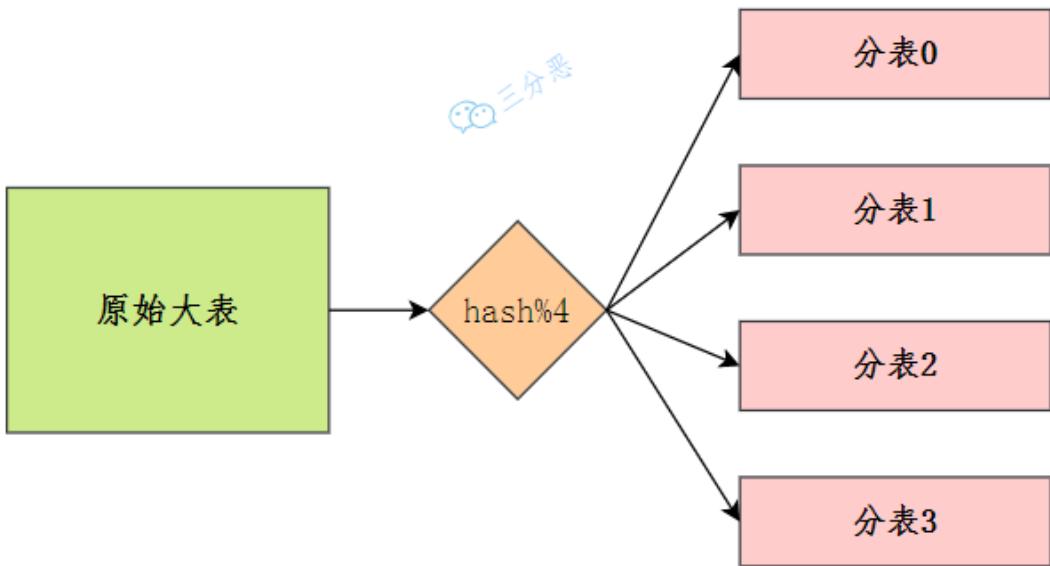
范围路由设计的复杂点主要体现在分段大小的选取上，分段太小会导致切分后子表数量过多，增加维护复杂度；分段太大可能会导致单表依然存在性能问题，一般建议分段大小在 100 万至2000 万之间，具体需要根据业务选取合适的分段大小。

范围路由的优点是可以随着数据的增加平滑地扩充新的表。例如，现在的用户是 100 万，如果增加到 1000 万，只需要增加新的表就可以了，原有的数据不需要动。范围路由的一个比较隐含的缺点是分布不均匀，假如按照 1000 万来进行分表，有可能某个分段实际存储的数据量只有 1000 条，而另外一个分段实际存储的数据量有 900 万条。

- Hash 路由：选取某个列（或者某几个列组合也可以）的值进行 Hash 运算，然后根据 Hash 结果分散到不同的数据库表中。

同样以订单 id 为例，假如我们一开始就规划了 4个数据库表，路由算法可以简单地用 $id \% 4$ 的值来表示数据所属的数据库表编号， id 为 12 的订单放到编号为 50 的子表中， id 为 13 的订单放到编号为 61 的子表中。

哈希路由

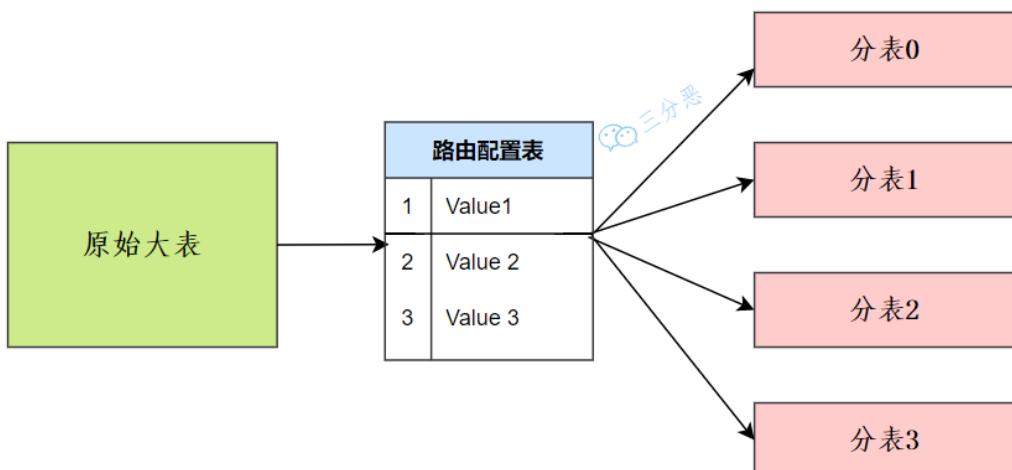


Hash 路由设计的复杂点主要体现在初始表数量的选取上，表数量太多维护比较麻烦，表数量太少又可能导致单表性能存在问题。而用了 Hash 路由后，增加子表数量是非常麻烦的，所有数据都要重分布。Hash 路由的优缺点和范围路由基本相反，Hash 路由的优点是表分布比较均匀，缺点是扩充新的表很麻烦，所有数据都要重分布。

- 配置路由：配置路由就是路由表，用一张独立的表来记录路由信息。同样以订单 id 为例，我们新增一张 `order_router` 表，这个表包含 `orderjd` 和 `tablejd` 两列，根据 `orderjd` 就可以查询对应的 `table_id`。

配置路由设计简单，使用起来非常灵活，尤其是在扩充表的时候，只需要迁移指定的数据，然后修改路由表就可以了。

配置路由



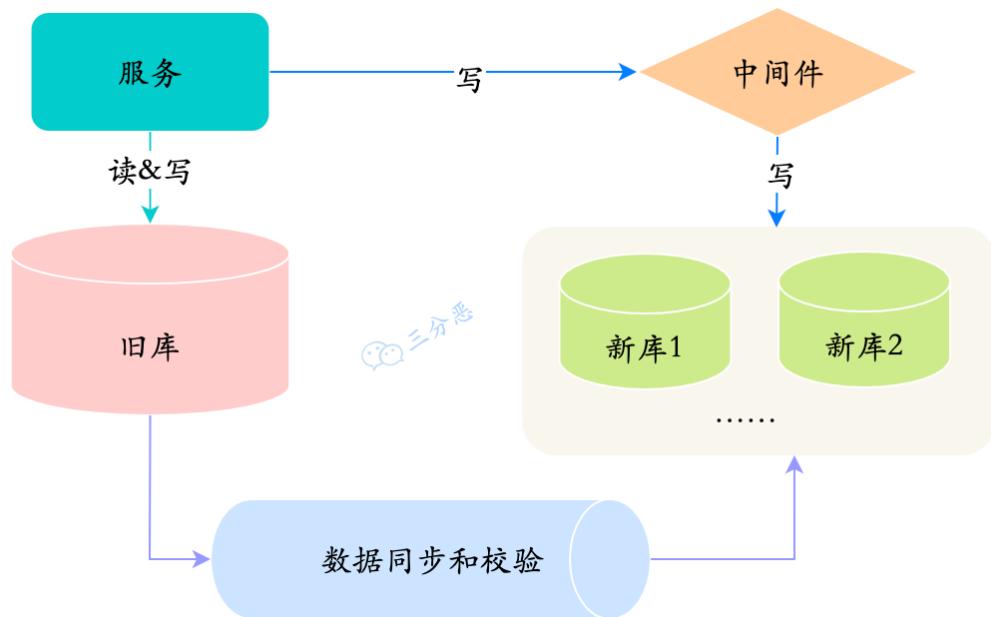
配置路由的缺点就是必须多查询一次，会影响整体性能；而且路由表本身如果太大（例如，几亿条数据），性能同样可能成为瓶颈，如果我们再次将路由表分库分表，则又面临一个死循环式的路由算法选择问题。

61. 不停机扩容怎么实现？

实际上，不停机扩容，实操起来是个非常麻烦而且很有风险的操作，当然，面试回答起来就简单很多。

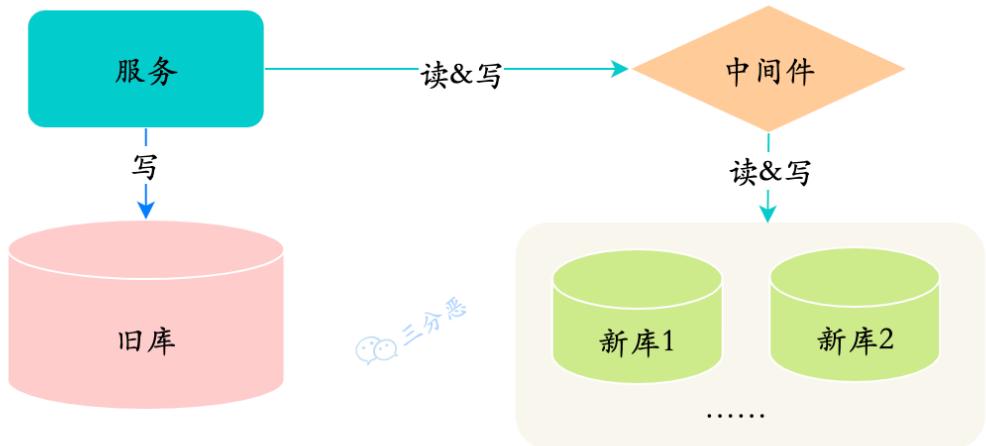
- 第一阶段：在线双写，查询走老库

1. 建立好新的库表结构，数据写入久库的同时，也写入拆分的新库
2. 数据迁移，使用数据迁移程序，将旧库中的历史数据迁移到新库
3. 使用定时任务，新旧库的数据对比，把差异补齐



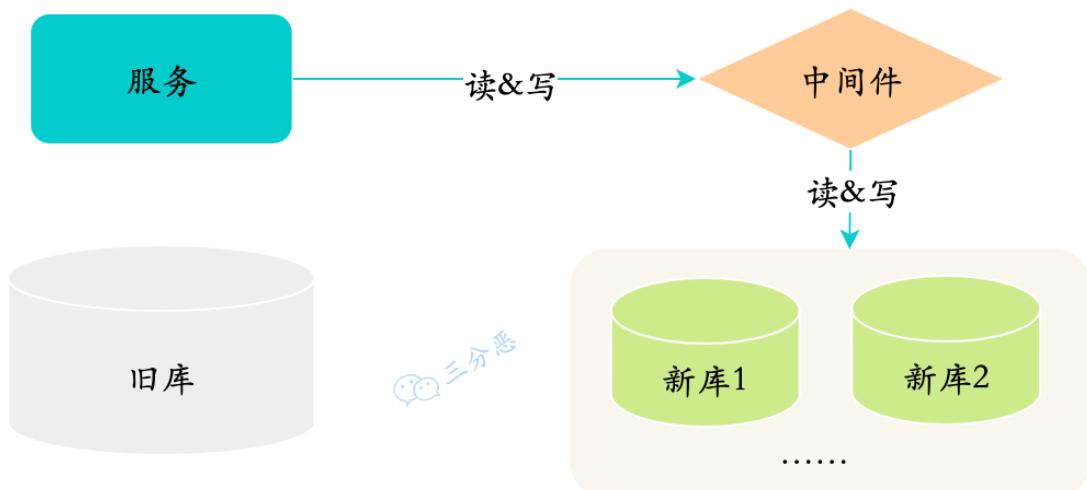
- 第二阶段：在线双写，查询走新库

1. 完成了历史数据的同步和校验
2. 把对数据的读切换到新库



- 第三阶段：旧库下线

1. 旧库不再写入新的数据
2. 经过一段时间，确定旧库没有请求之后，就可以下线老库



62. 常用的分库分表中间件有哪些？

- sharding-jdbc
- Mycat

63. 那你觉得分库分表会带来什么问题呢？

从分库的角度来讲：

- 事务的问题

使用关系型数据库，有很大一点在于它保证事务完整性。

而分库之后单机事务就用不上了，必须使用分布式事务来解决。

- 跨库 JOIN 问题

在一个库中的时候我们还可以利用 JOIN 来连表查询，而跨库了之后就无法使用 JOIN 了。

此时的解决方案就是在业务代码中进行关联，也就是先把一个表的数据查出来，然后通过得到的结果再去查另一张表，然后利用代码来关联得到最终的结果。

这种方式实现起来稍微比较复杂，不过也是可以接受的。

还有可以适当的冗余一些字段。比如以前的表就存储一个关联 ID，但是业务时常常要求返回对应的 Name 或者其他字段。这时候就可以把这些字段冗余到当前表中，来去除需要关联的操作。

还有一种方式就是数据异构，通过binlog同步等方式，把需要跨库join的数据异构到ES等存储结构中，通过ES进行查询。

从分表的角度来看：

- 跨节点的 count,order by,group by 以及聚合函数问题

只能由业务代码来实现或者用中间件将各表中的数据汇总、排序、分页然后返回。

- 数据迁移，容量规划，扩容等问题

数据的迁移，容量如何规划，未来是否可能再次需要扩容，等等，都是需要考虑的问题。

- ID 问题

数据库表被切分后，不能再依赖数据库自身的主键生成机制，所以需要一些手段来保证全局主键唯一。

1. 还是自增，只不过自增步长设置一下。比如现在有三张表，步长设置为3，三张表 ID 初始值分别是1、2、3。这样第一张表的 ID 增长是 1、4、7。第二张表是 2、5、8。第三张表是3、6、9，这样就不会重复了。
2. UUID，这种最简单，但是不连续的主键插入会导致严重的页分裂，性能比较差。
3. 分布式 ID，比较出名的就是 Twitter 开源的 sonwflake 雪花算法

64. 百万级别以上的数据如何删除？

关于索引：由于索引需要额外的维护成本，因为索引文件是单独存在的文件，所以当我们对数据的增加、修改、删除，都会产生额外的对索引文件的操作，这些操作需要消耗额外的IO，会降低增/改/删的执行效率。

所以，在我们删除数据库百万级别数据的时候，查询MySQL官方手册得知删除数据的速度和创建的索引数量是成正比的。

1. 所以我们想要删除百万数据的时候可以先删除索引
2. 然后删除其中无用数据
3. 删除完成后重新创建索引创建索引也非常快

65. 百万千万级大表如何添加字段？

当线上的数据库数据量到达几百万、上千万的时候，加一个字段就没那么简单，因为可能会长时间锁表。

大表添加字段，通常有这些做法：

- 通过中间表转换过去

创建一个临时的新表，把旧表的结构完全复制过去，添加字段，再把旧表数据复制过去，删除旧表，新表命名为旧表的名称，这种方式可能回丢掉一些数据。

- 用pt-online-schema-change

`pt-online-schema-change` 是percona公司开发的一个工具，它可以在线修改表结构，它的原理也是通过中间表。

- 先在从库添加 再进行主从切换

如果一张表数据量大且是热表（读写特别频繁），则可以考虑先在从库添加，再进行主从切换，切换后再将其他几个节点上添加字段。

66. MySQL 数据库 cpu 飙升的话，要怎么处理呢？

排查过程：

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

处理：

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

其他情况：

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

参考：

- [1]. 《高性能MySQL》
- [2]. 《MySQL技术内幕 InnoDB存储引擎》
- [3]. 《MySQL实战45讲》
- [4]. 《MySQL 是怎样运行的：从根儿上理解 MySQL》
- [5]. [快问快答，MySQL 面试夺命 20 问](#)
- [6]. 艾小仙 《我想进大厂》
- [7]. [100道MySQL数据库经典面试题解析（收藏版）](#)
- [8]. [MySQL索引从基础到原理，看这一篇就够了](#)
- [9]. [图文并茂的带你彻底理解悲观锁与乐观锁](#)

- [10]. [最完整MySQL数据库面试题（2020最新版）](#)
- [11]. [MySQL提升笔记（2）：存储引擎盘点](#)
- [13]. [不会看 Explain 执行计划，劝你简历别写熟悉 SQL 优化](#)
- [14]. [为了让你彻底弄懂 MySQL 事务日志，我通宵肝出了这份图解！](#)
- [15]. 《极客时间 高并发系统设计40问》
- [16]. 《极客时间 从零开始学架构》
- [17]. [在面试时被问到，为什么 MySQL 数据库数据量大了要进行分库分表？](#)
- [18]. [为了让你彻底弄懂 MySQL 事务日志，我通宵肝出了这份图解！](#)
- [19]. [MySQL 三万字精华总结 + 面试100 问，和面试官扯皮绰绰有余（收藏系列）](#)
- [20]. [面试中的老大难-mysql事务和锁，一次性讲清楚！](#)
- [21]. [一文彻底读懂MySQL事务的四大隔离级别](#)
- [22]. [MySQL存储引擎——MyISAM与InnoDB区别](#)
- [23]. [解决死锁之路（终结篇） - 再见死锁](#)
- [24]. [大众点评订单系统分库分表实践](#)

关注公众号：三分恶

手册更新动态
即刻送达



添加个人微信：ThirdFighter

技术交流
加大佬云集微信群



二、Redis

基础

1.说说什么是Redis?



Redis是一种基于键值对（key-value）的NoSQL数据库。

比一般键值对数据库强大的地方，Redis中的value支持string（字符串）、hash（哈希）、list（列表）、set（集合）、zset（有序集合）、Bitmaps（位图）、HyperLogLog、GEO（地理信息定位）等多种数据结构，因此Redis可以满足很多的应用场景。

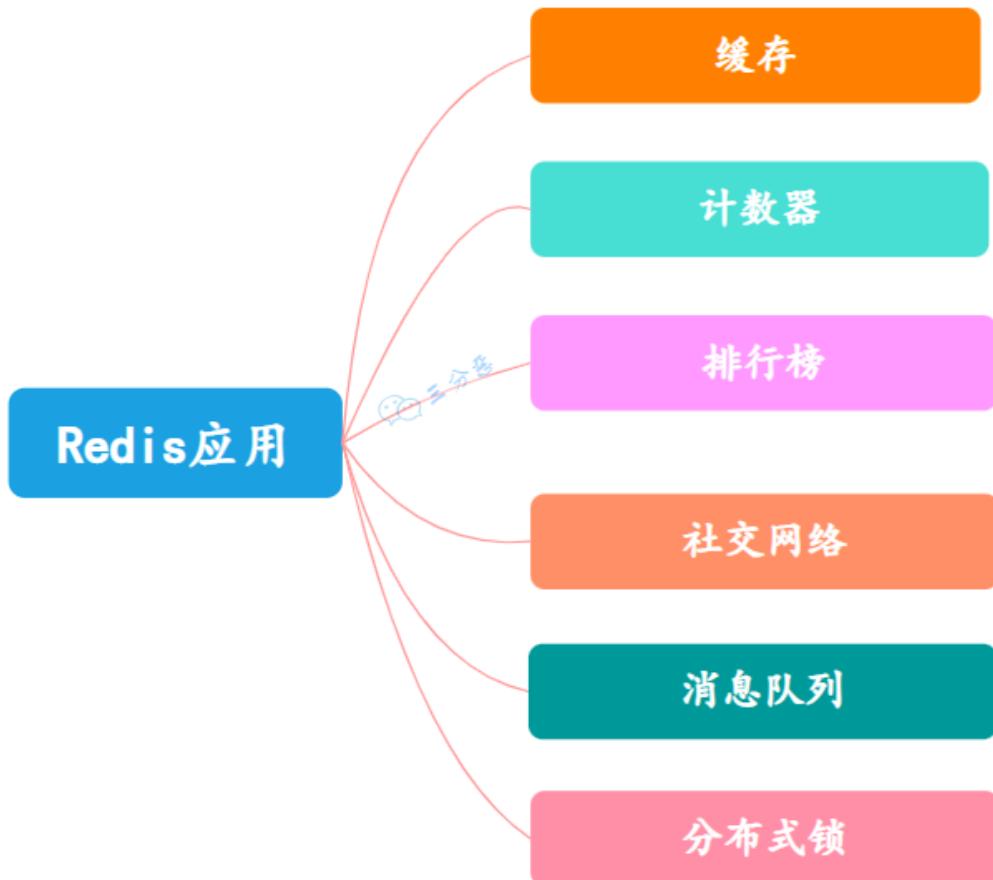
而且因为Redis会将所有数据都存放在内存中，所以它的读写性能非常出色。

不仅如此，Redis还可以将内存的数据利用快照和日志的形式保存到硬盘上，这样在发生类似断电或者机器故障的时候，内存中的数据不会“丢失”。

除了上述功能以外，Redis还提供了键过期、发布订阅、事务、流水线、Lua脚本等附加功能。

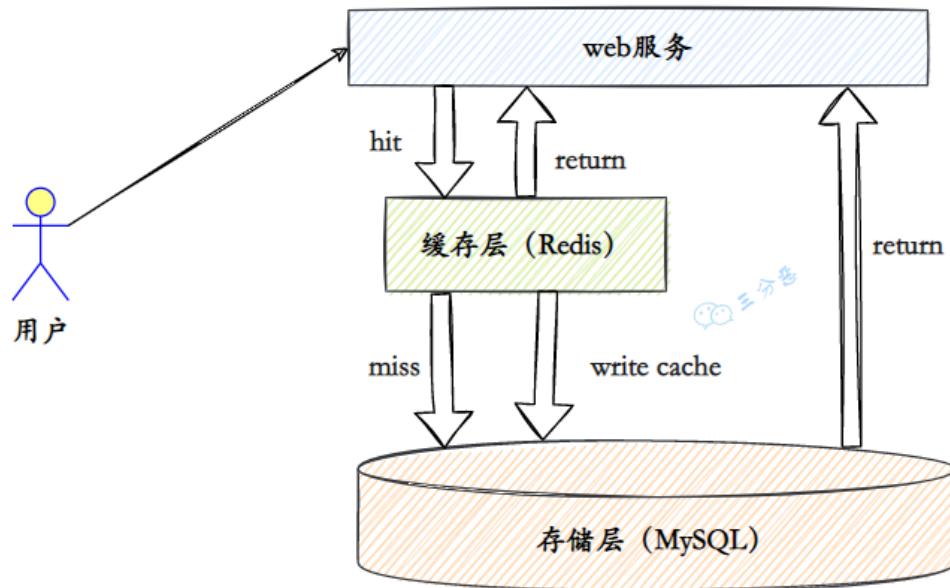
总之，Redis是一款强大的性能利器。

2.Redis可以用来干什么？



1. 缓存

这是Redis应用最广泛地方，基本所有的Web应用都会使用Redis作为缓存，来降低数据源压力，提高响应速度。



2. 计数器

Redis天然支持计数功能，而且计数性能非常好，可以用来记录浏览量、点赞量等等。

3. 排行榜

Redis提供了列表和有序集合数据结构，合理地使用这些数据结构可以很方便地构建各种排行榜系统。

4. 社交网络

赞/踩、粉丝、共同好友/喜好、推送、下拉刷新。

5. 消息队列

Redis提供了发布订阅功能和阻塞队列的功能，可以满足一般消息队列功能。

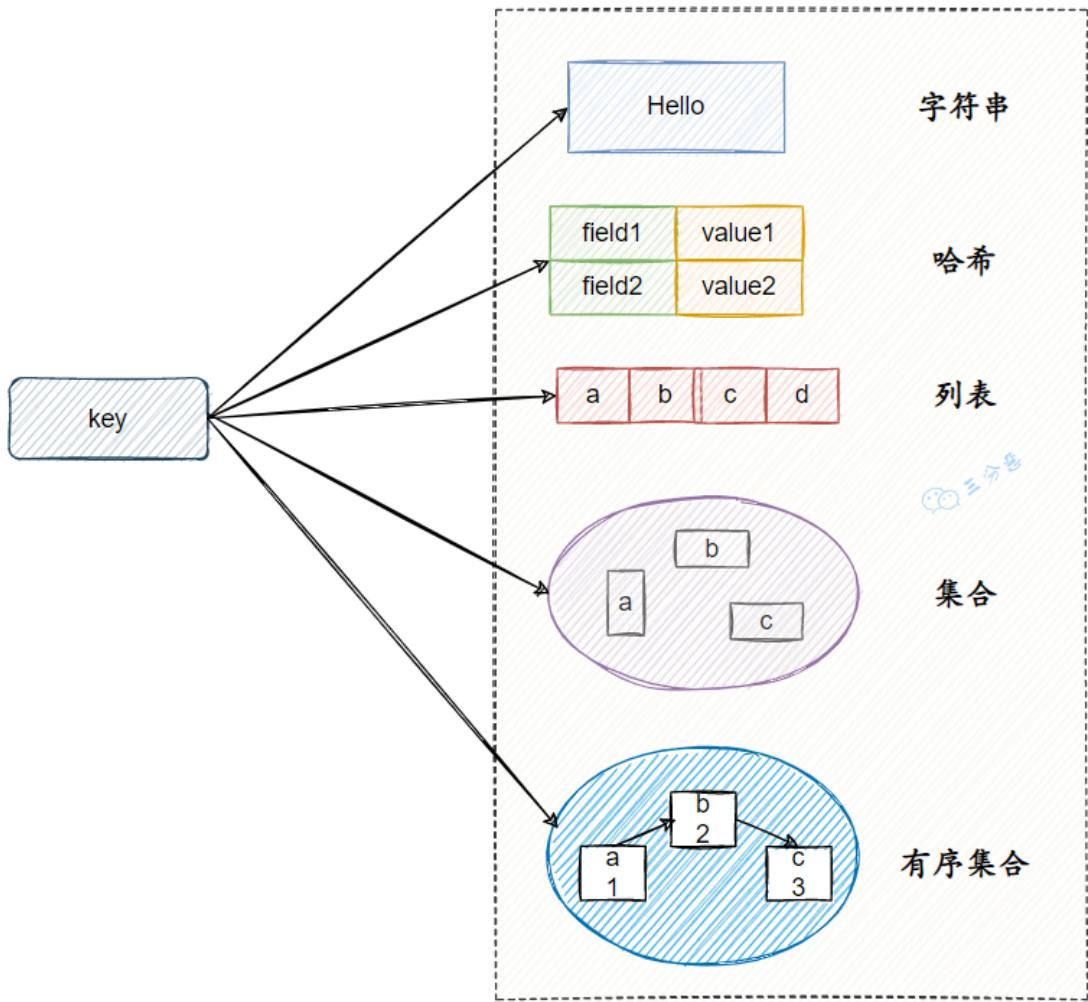
6. 分布式锁

分布式环境下，利用Redis实现分布式锁，也是Redis常见的应用。

Redis的应用一般会结合项目去问，以一个电商项目的用户服务为例：

- Token存储：用户登录成功之后，使用Redis存储Token
- 登录失败次数计数：使用Redis计数，登录失败超过一定次数，锁定账号
- 地址缓存：对省市区数据的缓存
- 分布式锁：分布式环境下登录、注册等操作加分布式锁
-

3.Redis 有哪些数据结构？



Redis有五种基本数据结构。

string

字符串最基础的数据结构。字符串类型的值实际可以是字符串（简单的字符串、复杂的字符串（例如JSON、XML））、数字（整数、浮点数），甚至是二进制（图片、音频、视频），但是值最大不能超过512MB。

字符串主要有以下几个典型使用场景：

- 缓存功能
- 计数
- 共享Session
- 限速

hash

哈希类型是指键值本身又是一个键值对结构。

哈希主要有以下典型应用场景：

- 缓存用户信息

- 缓存对象

list

列表（list）类型是用来存储多个有序的字符串。列表是一种比较灵活的数据结构，它可以充当栈和队列的角色

列表主要有以下几种使用场景：

- 消息队列
- 文章列表

set

集合（set）类型也是用来保存多个的字符串元素，但和列表类型不一样的是，集合中不允许有重复元素，并且集合中的元素是无序的。

集合主要有如下使用场景：

- 标签（tag）
- 共同关注

sorted set

有序集合中的元素可以排序。但是它和列表使用索引下标作为排序依据不同的是，它给每个元素设置一个权重（score）作为排序的依据。

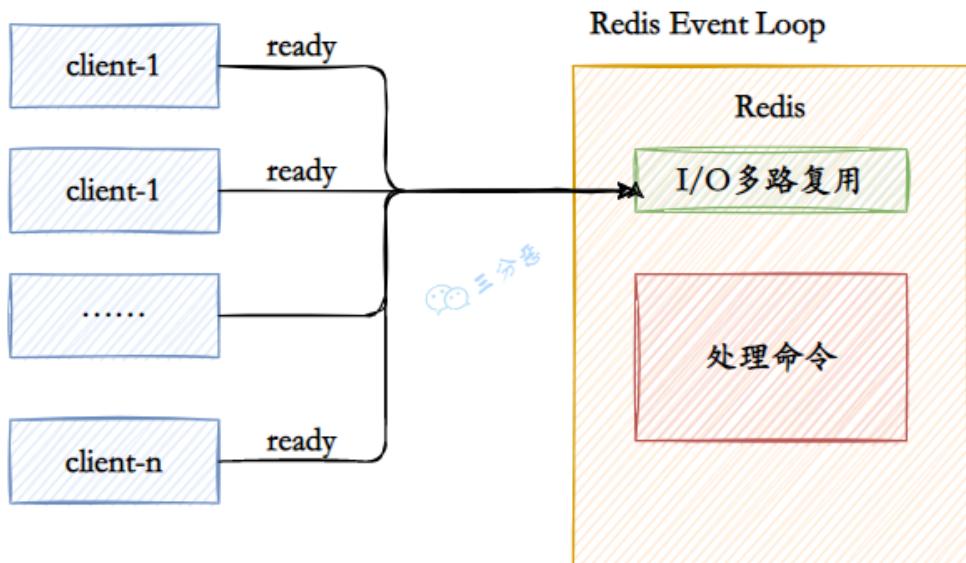
有序集合主要应用场景：

- 用户点赞统计
- 用户排序

4.Redis为什么快呢？

Redis的速度非常的快，单机的Redis就可以支撑每秒十几万的并发，相对于MySQL来说，性能是MySQL的几十倍。速度快的原因主要有几点：

1. 完全基于内存操作
2. 使用单线程，避免了线程切换和竞态产生的消耗
3. 基于非阻塞的IO多路复用机制
4. C语言实现，优化过的数据结构，基于几种基础的数据结构，redis做了大量的优化，性能极高



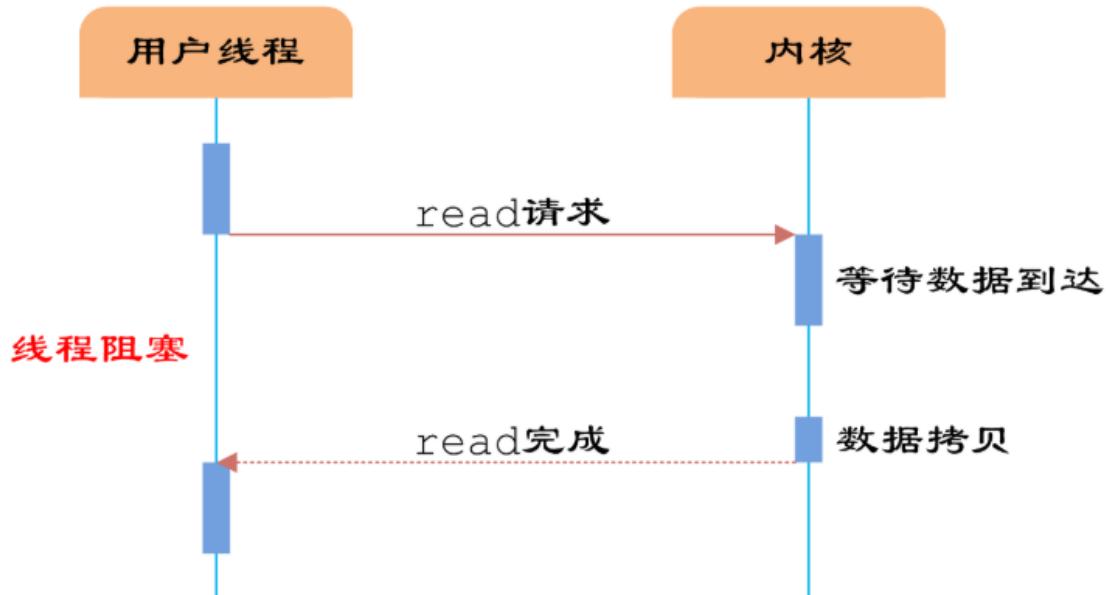
5. 能说一下I/O多路复用吗？

引用知乎上一个高赞的回答来解释什么是I/O多路复用。假设你是一个老师，让30个学生解答一道题目，然后检查学生做的是否正确，你有下面几个选择：

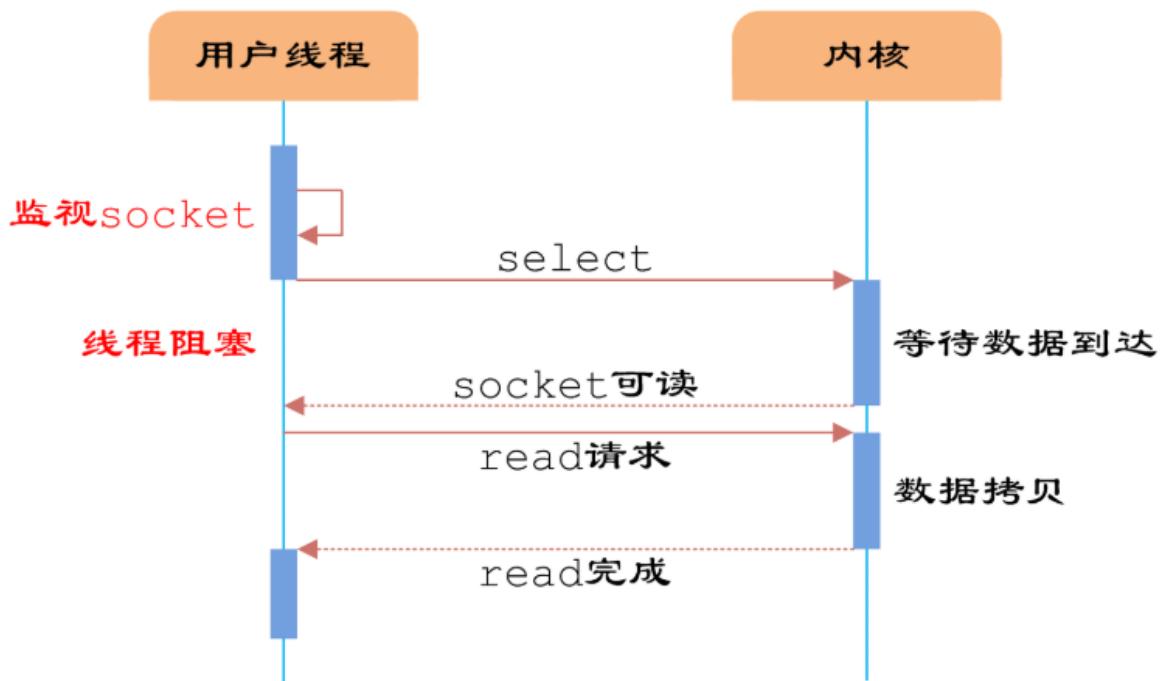
- 第一种选择：按顺序逐个检查，先检查A，然后是B，之后是C、D。。。这中间如果有一个学生卡住，全班都会被耽误。这种模式就好比，你用循环挨个处理socket，根本不具有并发能力。
- 第二种选择：你创建30个分身，每个分身检查一个学生的答案是否正确。这种类似于为每一个用户创建一个进程或者- 线程处理连接。
- 第三种选择，你站在讲台上等，谁解答完谁举手。这时C、D举手，表示他们解答问题完毕，你下去依次检查C、D的答案，然后继续回到讲台上等。此时E、A又举手，然后去处理E和A。

第一种就是阻塞IO模型，第三种就是I/O复用模型。

阻塞I/O模型



I/O多路复用模型



Linux系统有三种方式实现IO多路复用：select、poll和epoll。

例如epoll方式是将用户socket对应的fd注册进epoll，然后epoll帮你监听哪些socket上有消息到达，这样就避免了大量的无用操作。此时的socket应该采用非阻塞模式。

这样，整个过程只在进行select、poll、epoll这些调用的时候才会阻塞，收发客户消息是不会阻塞的，整个进程或者线程就被充分利用起来，这就是事件驱动，所谓的reactor模式。

6. Redis为什么早期选择单线程？

官方解释：<https://redis.io/topics/faq>

Redis is single threaded. How can I exploit multiple CPU / cores?

It's not very frequent that CPU becomes your bottleneck with Redis, as usually Redis is either memory or network bound. For instance, using pipelining Redis running on an average Linux system can deliver even 1 million requests per second, so if your application mainly uses O(N) or O(log(N)) commands, it is hardly going to use too much CPU. However, to maximize CPU usage you can start multiple instances of Redis in the same box and treat them as different servers. At some point a single box may not be enough anyway, so if you want to use multiple CPUs you can start thinking of some way to shard earlier.

You can find more information about using multiple Redis instances in the [Partitioning page](#).

However with Redis 4.0 we started to make Redis more threaded. For now this is limited to deleting objects in the background, and to blocking commands implemented via Redis modules. For future releases, the plan is to make Redis more and more threaded.

官方FAQ表示，因为Redis是基于内存的操作，CPU成为Redis的瓶颈的情况很少见，Redis的瓶颈最有可能是内存的大小或者网络限制。

如果想要最大程度利用CPU，可以在一台机器上启动多个Redis实例。

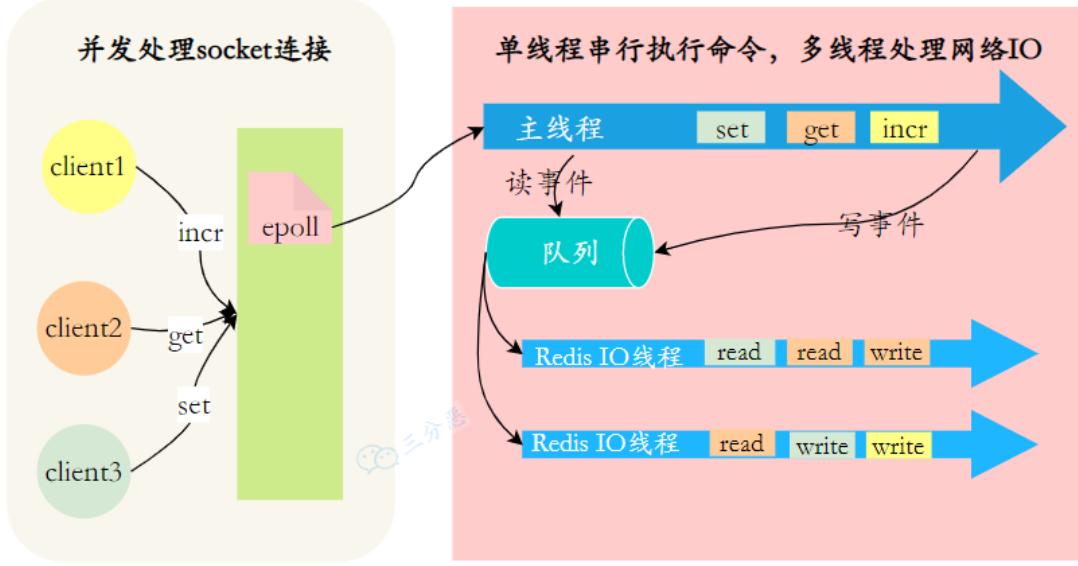
PS：网上有这样的回答，吐槽官方的解释有些敷衍，其实就是历史原因，开发者嫌多线程麻烦，后来这个CPU的利用问题就被抛给了使用者。

同时FAQ里还提到了，Redis 4.0 之后开始变成多线程，除了主线程外，它也有后台线程在处理一些较为缓慢的操作，例如清理脏数据、无用连接的释放、大 Key 的删除等等。

7.Redis6.0使用多线程是怎么回事？

Redis不是说用单线程的吗？怎么6.0成了多线程的？

Redis6.0的多线程是用多线程来处理数据的读写和协议解析，但是Redis执行命令还是单线程的。



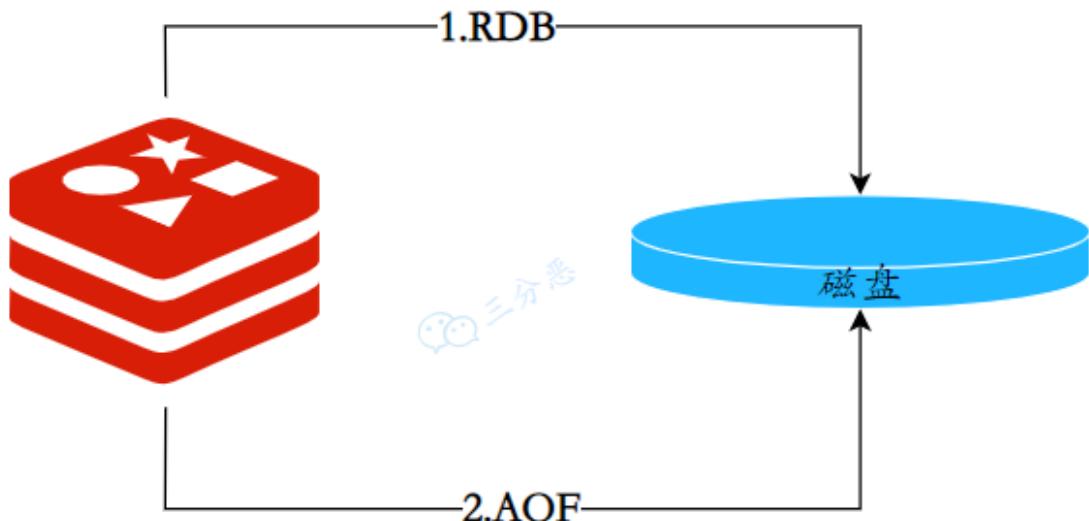
这样做的目的是因为Redis的性能瓶颈在于网络IO而非CPU，使用多线程能提升IO读写的效率，从而整体提高Redis的性能。

持久化

8.Redis持久化方式有哪些？有什么区别？

Redis持久化方案分为RDB和AOF两种。

Redis持久化的两种方式

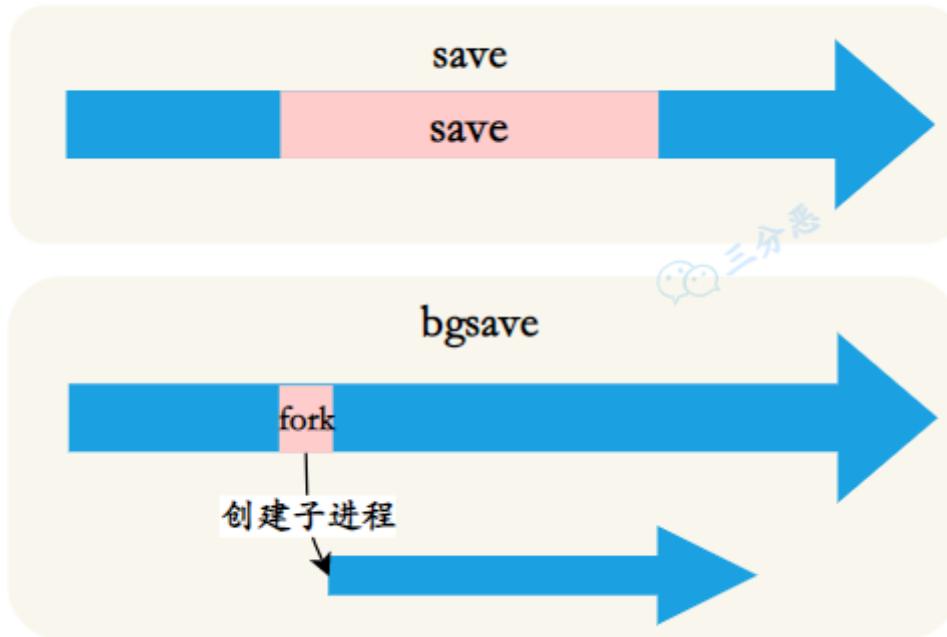


RDB

RDB持久化是把当前进程数据生成快照保存到硬盘的过程，触发RDB持久化过程分为手动触发和自动触发。

RDB文件是一个压缩的二进制文件，通过它可以还原某个时刻数据库的状态。由于RDB文件是保存在硬盘上的，所以即使Redis崩溃或者退出，只要RDB文件存在，就可以用它来恢复还原数据库的状态。

手动触发分别对应save和bgsave命令：



- save命令：阻塞当前Redis服务器，直到RDB过程完成为止，对于内存比较大的实例会造成长时间阻塞，线上环境不建议使用。
- bgsave命令：Redis进程执行fork操作创建子进程，RDB持久化过程由子进程负责，完成后自动结束。阻塞只发生在fork阶段，一般时间很短。

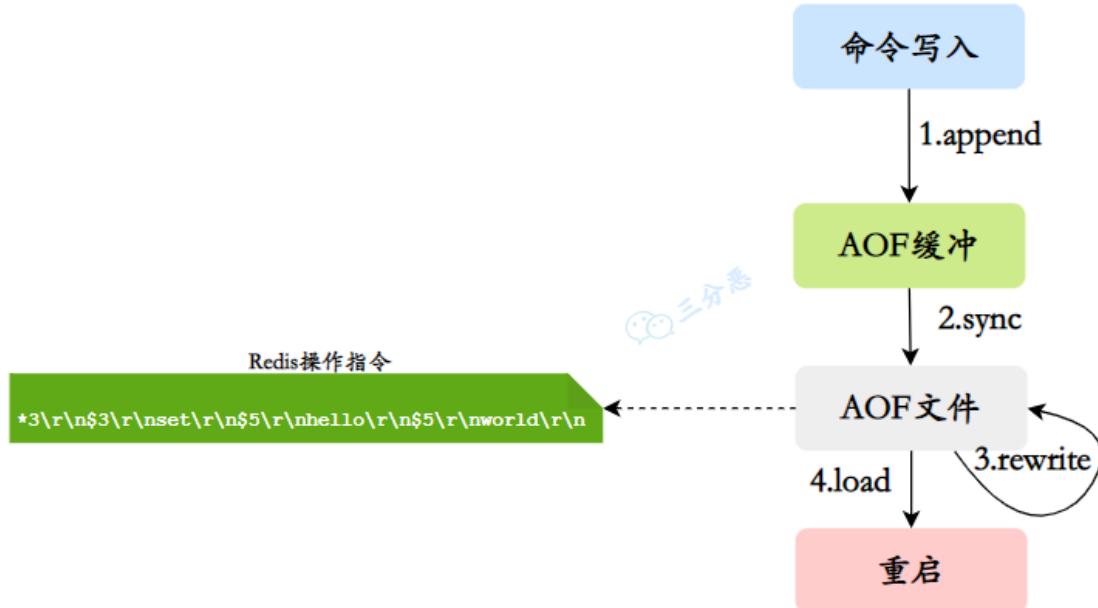
以下场景会自动触发RDB持久化：

- 使用save相关配置，如“save m n”。表示m秒内数据集存在n次修改时，自动触发bgsave。
- 如果从节点执行全量复制操作，主节点自动执行bgsave生成RDB文件并发送给从节点
- 执行debug reload命令重新加载Redis时，也会自动触发save操作
- 默认情况下执行shutdown命令时，如果没有开启AOF持久化功能则自动执行bgsave。

AOF

AOF (append only file) 持久化：以独立日志的方式记录每次写命令，重启时再重新执行AOF文件中的命令达到恢复数据的目的。AOF的主要作用是解决了数据持久化的实时性，目前已经是Redis持久化的主流方式。

AOF的工作流程操作：命令写入（append）、文件同步（sync）、文件重写（rewrite）、重启加载（load）



流程如下：

- 1) 所有的写入命令会追加到aof_buf（缓冲区）中。
- 2) AOF缓冲区根据对应的策略向硬盘做同步操作。
- 3) 随着AOF文件越来越大，需要定期对AOF文件进行重写，达到压缩的目的。
- 4) 当Redis服务器重启时，可以加载AOF文件进行数据恢复。

9.RDB 和 AOF 各自有什么优缺点？

RDB | 优点

1. 只有一个紧凑的二进制文件 `dump.rdb`，非常适合备份、全量复制的场景。
2. 容灾性好，可以把RDB文件拷贝到远程机器或者文件系统上，用于容灾恢复。
3. 恢复速度快，RDB恢复数据的速度远远快于AOF的方式

RDB | 缺点

1. 实时性低，RDB是间隔一段时间进行持久化，没法做到实时持久化/秒级持久化。如果在这一间隔事件发生故障，数据会丢失。

- 存在兼容问题，Redis演进过程存在多个格式的RDB版本，存在老版本Redis无法兼容新版本RDB的问题。

AOF | 优点

- 实时性好，aof持久化可以配置 `appendfsync` 属性，有 `always`，每进行一次命令操作就记录到 aof 文件中一次。
- 通过 append 模式写文件，即使中途服务器宕机，可以通过 `redis-check-aof` 工具解决数据一致性问题。

AOF | 缺点

- AOF 文件比 RDB 文件大，且恢复速度慢。
- 数据集大的时候，比 RDB 启动效率低。

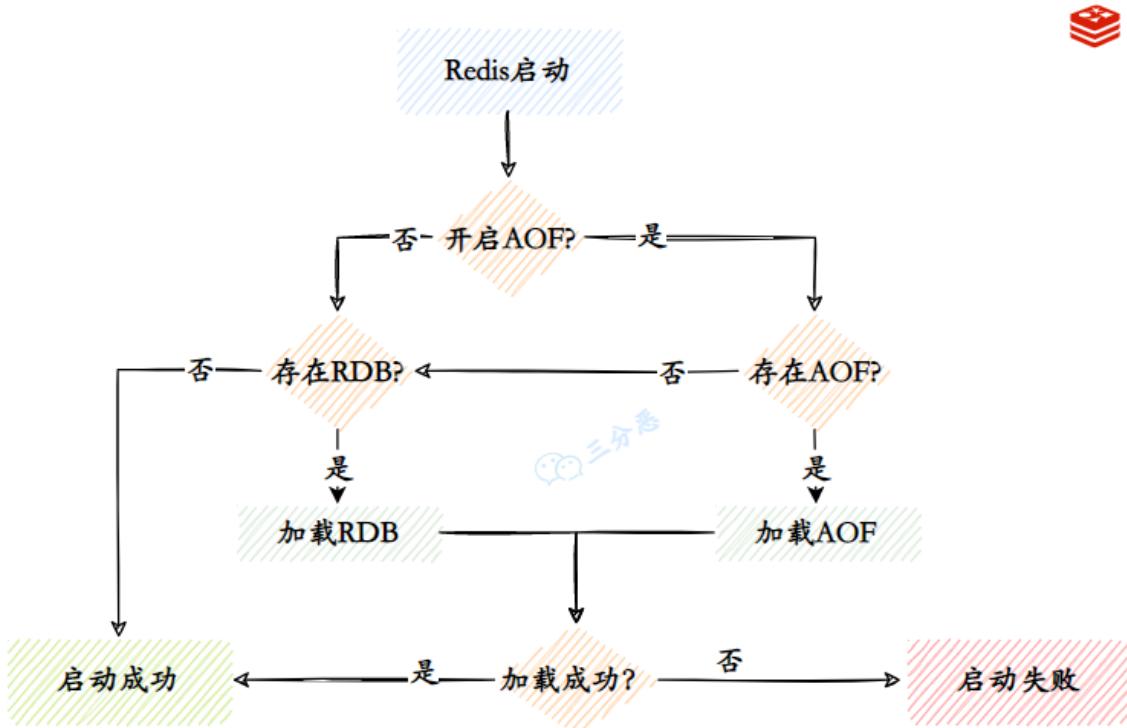
10.RDB和AOF如何选择？

- 一般来说，如果想达到足以媲美数据库的数据安全性，应该同时使用两种持久化功能。在这种情况下，当 Redis 重启的时候会优先载入 AOF 文件来恢复原始的数据，因为在通常情况下 AOF 文件保存的数据集要比 RDB 文件保存的数据集要完整。
- 如果可以接受数分钟以内的数据丢失，那么可以只使用 RDB 持久化。
- 有很多用户都只使用 AOF 持久化，但并不推荐这种方式，因为定时生成 RDB 快照（snapshot）非常便于进行数据备份，并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快，除此之外，使用 RDB 还可以避免 AOF 程序的 bug。
- 如果只需要数据在服务器运行的时候存在，也可以不使用任何持久化方式。

11.Redis的数据恢复？

当Redis发生了故障，可以从RDB或者AOF中恢复数据。

恢复的过程也很简单，把RDB或者AOF文件拷贝到Redis的数据目录下，如果使用AOF恢复，配置文件开启AOF，然后启动redis-server即可。



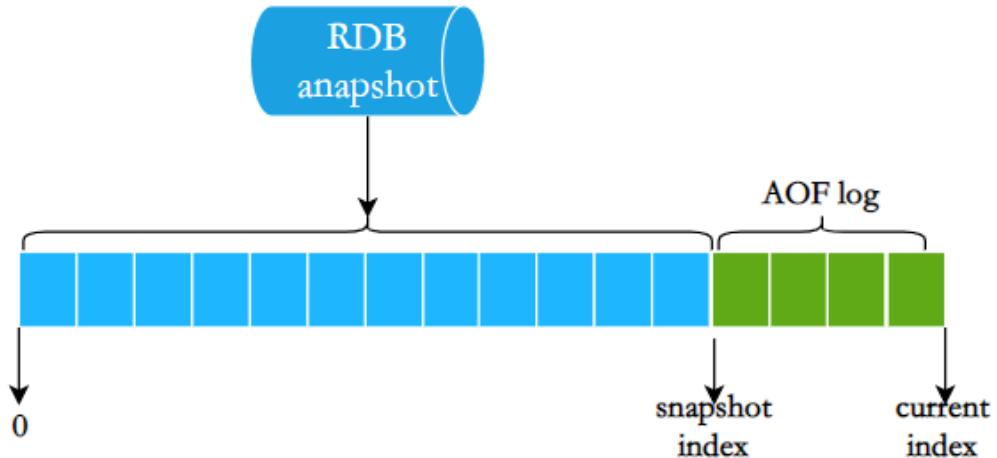
Redis 启动时加载数据的流程：

1. AOF持久化开启且存在AOF文件时，优先加载AOF文件。
2. AOF关闭或者AOF文件不存在时，加载RDB文件。
3. 加载AOF/RDB文件成功后，Redis启动成功。
4. AOF/RDB文件存在错误时，Redis启动失败并打印错误信息。

12.Redis 4.0 的混合持久化了解吗？

重启 Redis 时，我们很少使用 **RDB** 来恢复内存状态，因为会丢失大量数据。我们通常使用 AOF 日志重放，但是重放 AOF 日志性能相对 **RDB** 来说要慢很多，这样在 Redis 实例很大的情况下，启动需要花费很长的时间。

Redis 4.0 为了解决这个问题，带来了一个新的持久化选项——**混合持久化**。将 **rdb** 文件的内容和增量的 AOF 日志文件存在一起。这里的 AOF 日志不再是全量的日志，而是 **自持久化开始到持久化结束** 的这段时间发生的增量 AOF 日志，通常这部分 AOF 日志很小：



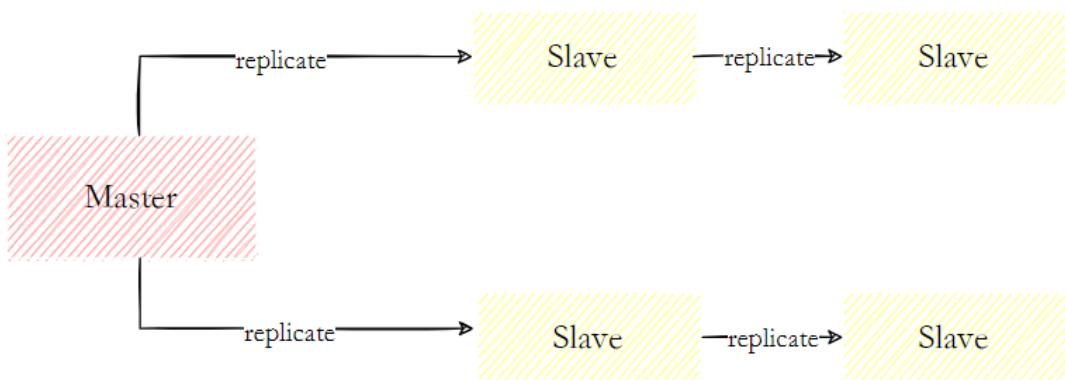
于是在 Redis 重启的时候，可以先加载 `rdb` 的内容，然后再重放增量 AOF 日志就可以完全替代之前的 AOF 全量文件重放，重启效率因此大幅得到提升。

高可用

Redis保证高可用主要有三种方式：主从、哨兵、集群。

13. 主从复制了解吗？

Redis主从复制



主从复制，是指将一台 Redis 服务器的数据，复制到其他的 Redis 服务器。前者称为 **主节点(master)**，后者称为 **从节点(slave)**。且数据的复制是 **单向** 的，只能由主节点到从节点。Redis 主从复制支持 **主从同步** 和 **从从同步** 两种，后者是 Redis 后续版本新增的功能，以减轻主节点的同步负担。

主从复制主要的作用？

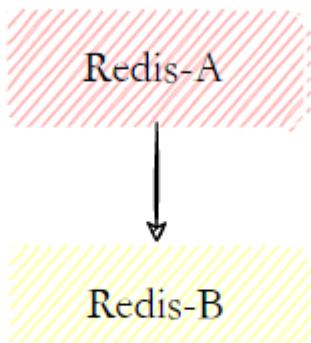
- 数据冗余： 主从复制实现了数据的热备份，是持久化之外的一种数据冗余方式。
- 故障恢复： 当主节点出现问题时，可以由从节点提供服务，实现快速的故障恢复（实际上是一种服务的冗余）。
- 负载均衡： 在主从复制的基础上，配合读写分离，可以由主节点提供写服务，由从节点提供读服务（即写 Redis 数据时应用连接主节点，读 Redis 数据时应用连接从节点），分担服务器负载。尤其是在写少读多的场景下，通过多个从节点分担读负载，可以大大提高 Redis 服务器的并发量。
- 高可用基石： 除了上述作用以外，主从复制还是哨兵和集群能够实施的基础，因此说主从复制是 Redis 高可用的基础。

14.Redis主从有几种常见的拓扑结构？

Redis的复制拓扑结构可以支持单层或多层复制关系，根据拓扑复杂性可以分为以下三种：一主一从、一主多从、树状主从结构。

1.一主一从结构

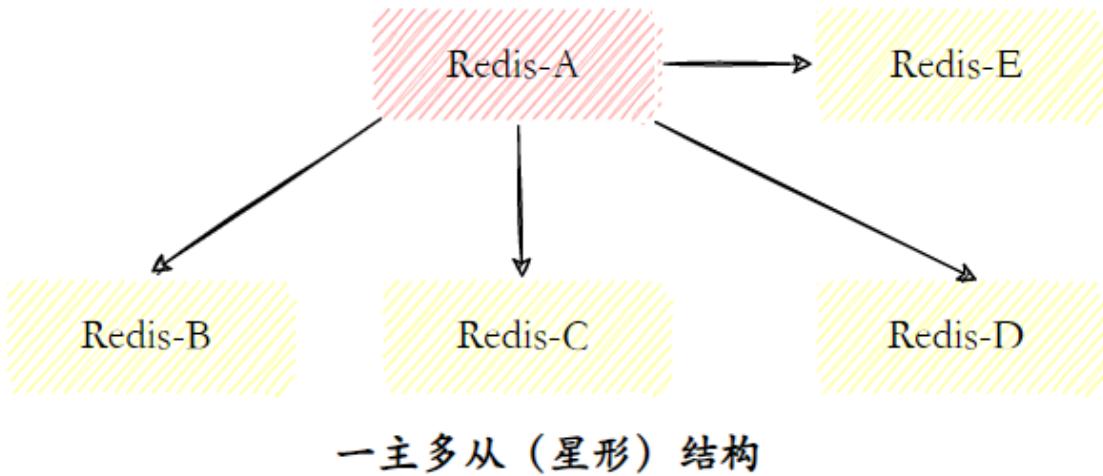
一主一从结构是最简单的复制拓扑结构，用于主节点出现宕机时从节点提供故障转移支持。



一主一从结构

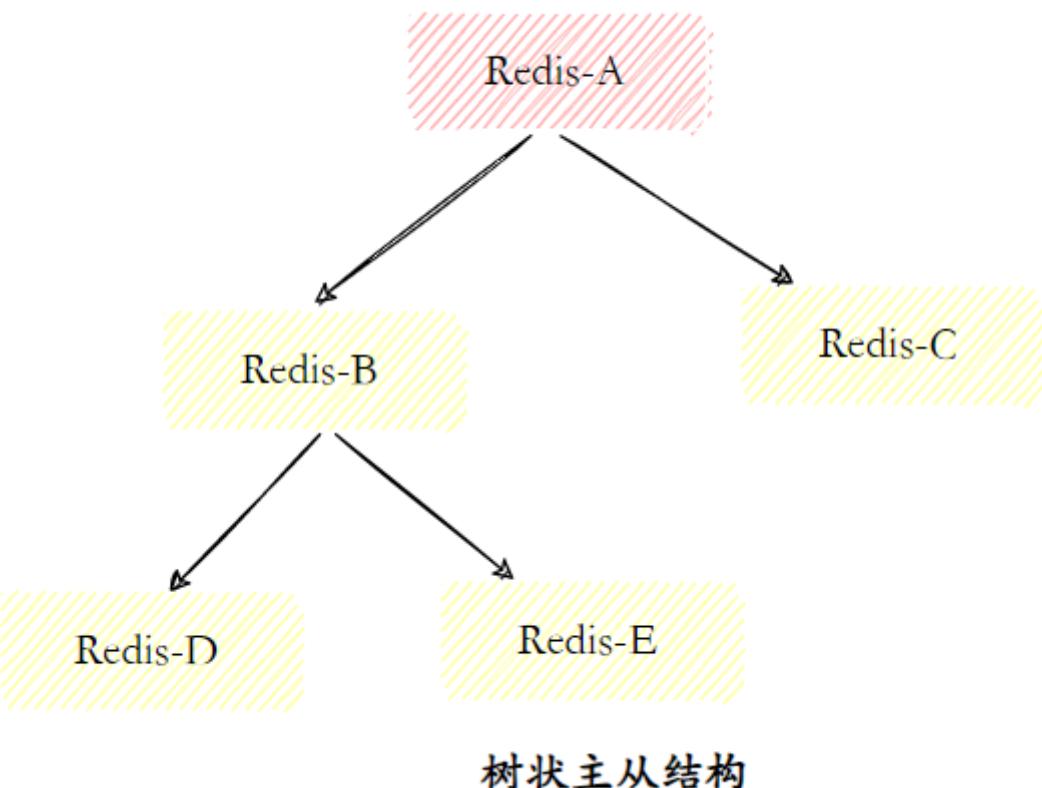
2.一主多从结构

一主多从结构（又称为星形拓扑结构）使得应用端可以利用多个从节点实现读写分离（见图6-5）。对于读占比较大的场景，可以把读命令发送到从节点来分担主节点压力。



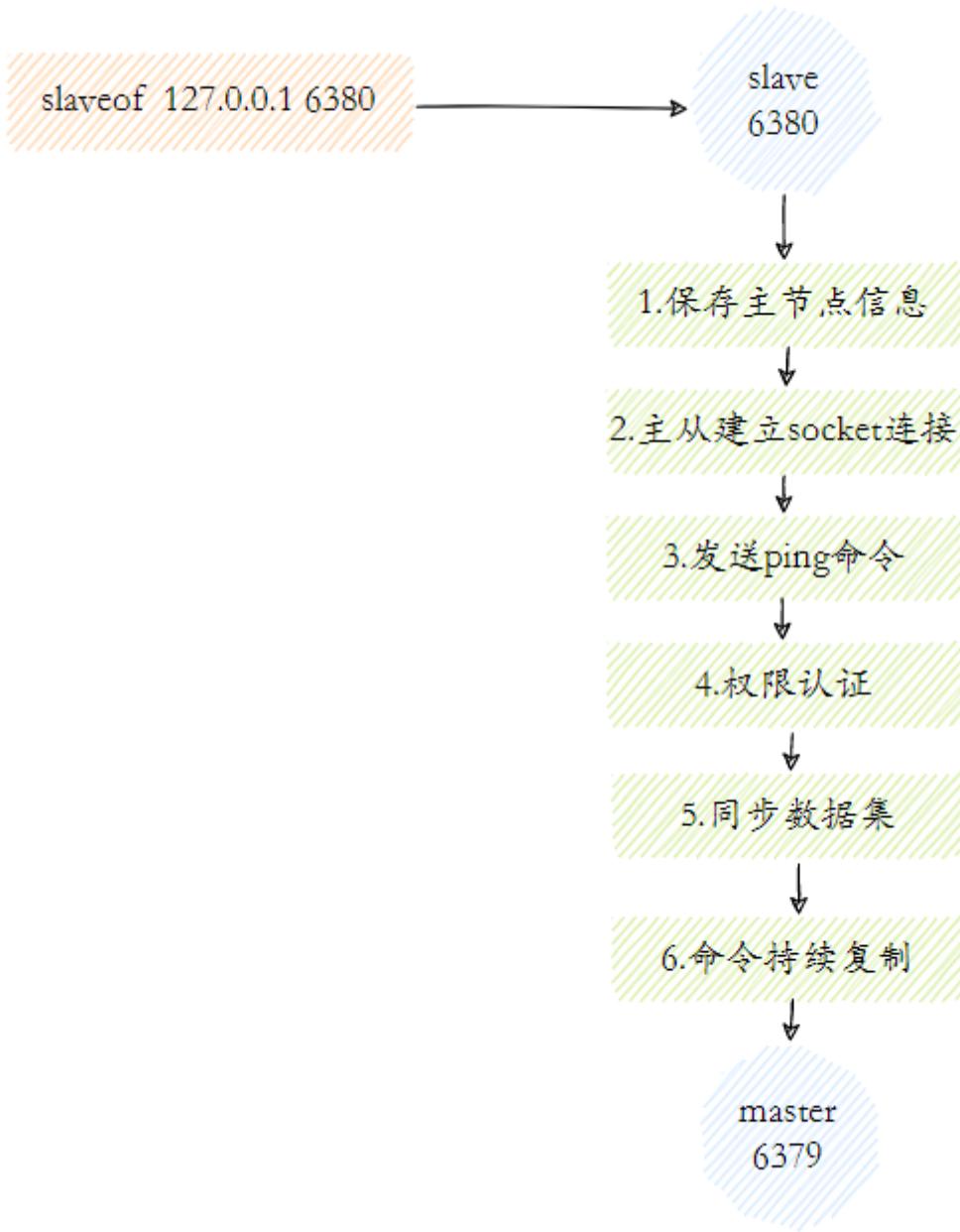
3. 树状主从结构

树状主从结构（又称为树状拓扑结构）使得从节点不但可以复制主节点数据，同时可以作为其他从节点的主节点继续向下层复制。通过引入复制中间层，可以有效降低主节点负载和需要传送给从节点的数据量。



15.Redis的主从复制原理了解吗？

Redis主从复制的工作流程大概可以分为如下几步：



1. 保存主节点（master）信息

这一步只是保存主节点信息，保存主节点的ip和port。

2. 主从建立连接

从节点（slave）发现新的主节点后，会尝试和主节点建立网络连接。

3. 发送ping命令

连接建立成功后从节点发送ping请求进行首次通信，主要是检测主从之间网络套接字是否可用、主节点当前是否可接受处理命令。

4. 权限验证

如果主节点要求密码验证，从节点必须正确的密码才能通过验证。

5. 同步数据集

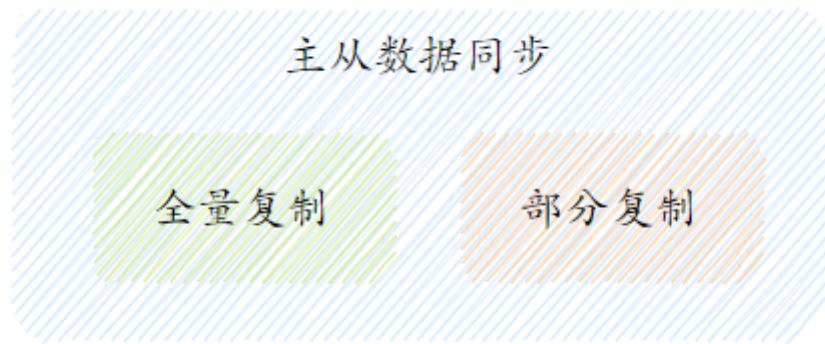
主从复制连接正常通信后，主节点会把持有的数据全部发送给从节点。

6. 命令持续复制

接下来主节点会持续地把写命令发送给从节点，保证主从数据一致性。

16. 说说主从数据同步的方式？

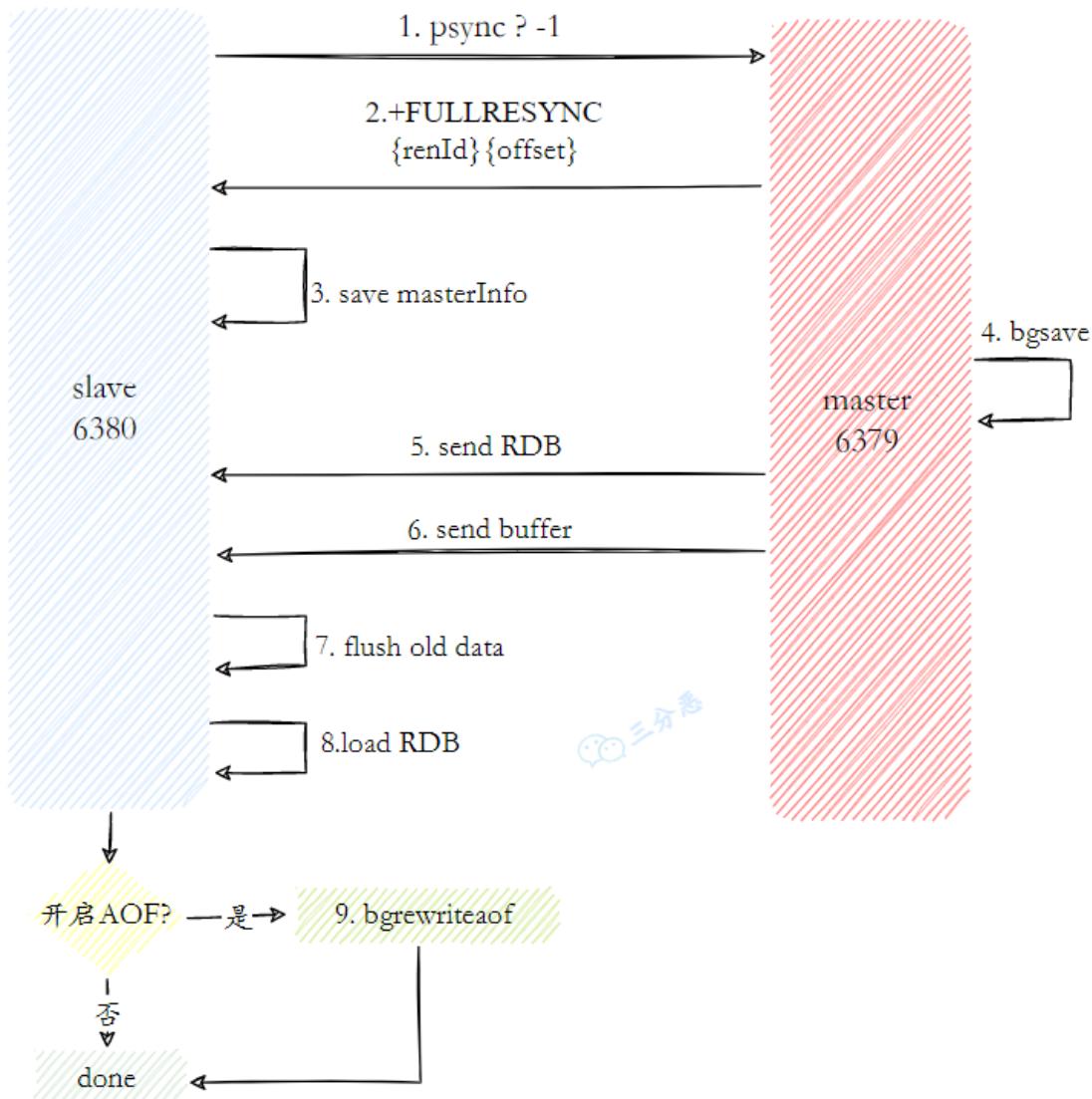
Redis在2.8及以上版本使用psync命令完成主从数据同步，同步过程分为：全量复制和部分复制。



全量复制

一般用于初次复制场景，Redis早期支持的复制功能只有全量复制，它会把主节点全部数据一次性发送给从节点，当数据量较大时，会对主从节点和网络造成很大的开销。

全量复制的完整运行流程如下：

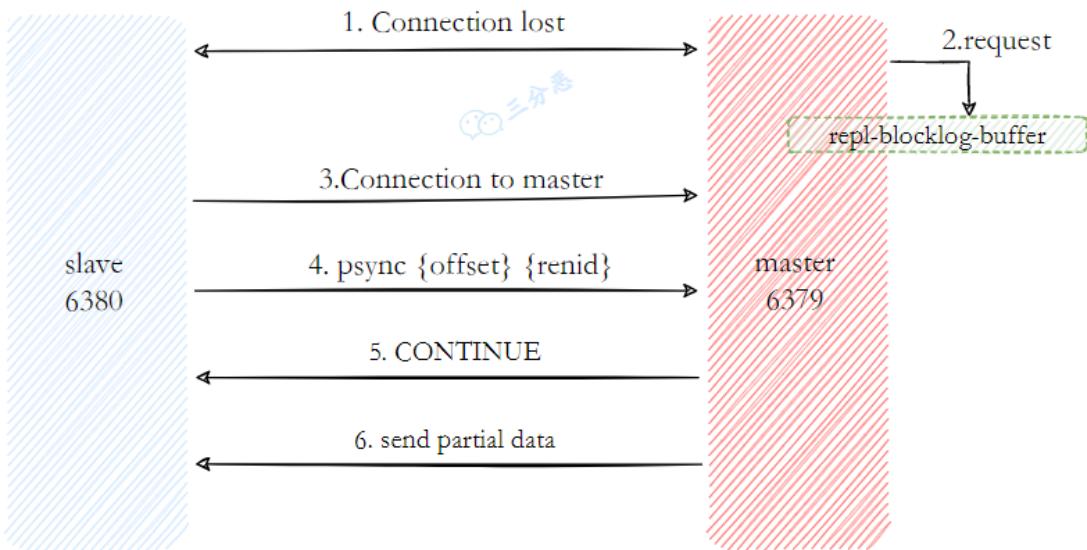


1. 发送psync命令进行数据同步，由于是第一次进行复制，从节点没有复制偏移量和主节点的运行ID，所以发送psync-1。
2. 主节点根据psync-1解析出当前为全量复制，回复+FULLRESYNC响应。
3. 从节点接收主节点的响应数据保存运行ID和偏移量offset
4. 主节点执行bgsave保存RDB文件到本地
5. 主节点发送RDB文件给从节点，从节点把接收的RDB文件保存在本地并直接作为从节点的数据文件
6. 对于从节点开始接收RDB快照到接收完成期间，主节点仍然响应读写命令，因此主节点会把这期间写命令数据保存在复制客户端缓冲区内，当从节点加载完RDB文件后，主节点再把缓冲区内的数据发送给从节点，保证主从之间数据一致性。
7. 从节点接收完主节点传送来的全部数据后会清空自身旧数据
8. 从节点清空数据后开始加载RDB文件

9. 从节点成功加载完RDB后，如果当前节点开启了AOF持久化功能，它会立刻做bgrewriteaof操作，为了保证全量复制后AOF持久化文件立刻可用。

部分复制

部分复制主要是Redis针对全量复制的过高水平做出的一种优化措施，使用psync{runId}{offset}命令实现。当从节点（slave）正在复制主节点（master）时，如果出现网络闪断或者命令丢失等异常情况时，从节点会向主节点要求补发丢失的命令数据，如果主节点的复制积压缓冲区内存在这部分数据则直接发送给从节点，这样就可以保持主从节点复制的一致性。



1. 当主从节点之间网络出现中断时，如果超过repl-timeout时间，主节点会认为从节点故障并中断复制连接
2. 主从连接中断期间主节点依然响应命令，但因复制连接中断命令无法发送给从节点，不过主节点内部存在的复制积压缓冲区，依然可以保存最近一段时间的写命令数据，默认最大缓存1MB。
3. 当主从节点网络恢复后，从节点会再次连上主节点
4. 当主从连接恢复后，由于从节点之前保存了自身已复制的偏移量和主节点的运行ID。因此会把它们当作psync参数发送给主节点，要求进行部分复制操作。
5. 主节点接到psync命令后首先核对参数runId是否与自身一致，如果一致，说明之前复制的是当前主节点；之后根据参数offset在自身复制积压缓冲区查找，如果偏移量之后的数据存在缓冲区中，则对从节点发送+CONTINUE响应，表示可以进行部分复制。
6. 主节点根据偏移量把复制积压缓冲区里的数据发送给从节点，保证主从复制进入正常状态。

17. 主从复制存在哪些问题呢？

主从复制虽好，但也存在一些问题：

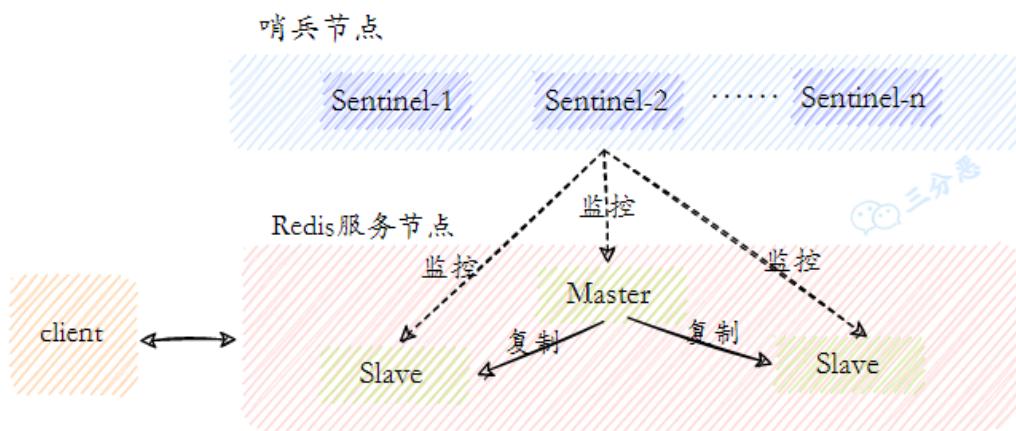
- 一旦主节点出现故障，需要手动将一个从节点晋升为主节点，同时需要修改应用方的主节点地址，还需要命令其他从节点去复制新的主节点，整个过程都需要人工干预。
- 主节点的写能力受到单机的限制。
- 主节点的存储能力受到单机的限制。

第一个问题是Redis的高可用问题，第二、三个问题属于Redis的分布式问题。

18.Redis Sentinel（哨兵）了解吗？

主从复制存在一个问题，没法完成自动故障转移。所以我们需要一个方案来完成自动故障转移，它就是Redis Sentinel（哨兵）。

Redis Sentinel



Redis Sentinel，它由两部分组成，哨兵节点和数据节点：

- 哨兵节点： 哨兵系统由一个或多个哨兵节点组成，哨兵节点是特殊的Redis节点，不存储数据，对数据节点进行监控。
- 数据节点： 主节点和从节点都是数据节点；

在复制的基础上，哨兵实现了 **自动化的故障恢复** 功能，下面是官方对于哨兵功能的描述：

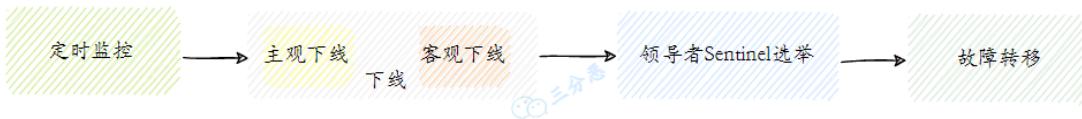
- 监控（Monitoring）： 哨兵会不断地检查主节点和从节点是否运作正常。
- 自动故障转移（Automatic failover）： 当 主节点 不能正常工作时，哨兵会开始 自动故障转移操作，它会将失效主节点的其中一个 从节点升级为新的主节点，并让其他从节点改为复制新的主节点。

- 配置提供者（Configuration provider）： 客户端在初始化时，通过连接哨兵来获得当前 Redis 服务的主节点地址。
- 通知（Notification）： 哨兵可以将故障转移的结果发送给客户端。

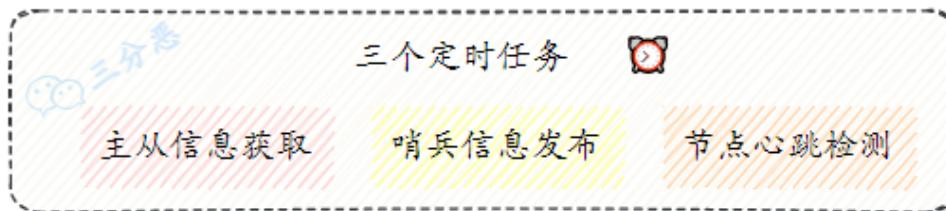
其中，监控和自动故障转移功能，使得哨兵可以及时发现主节点故障并完成转移。而配置提供者和通知功能，则需要在与客户端的交互中才能体现。

19.Redis Sentinel（哨兵）实现原理知道吗？

哨兵模式是通过哨兵节点完成对数据节点的监控、下线、故障转移。



• 定时监控

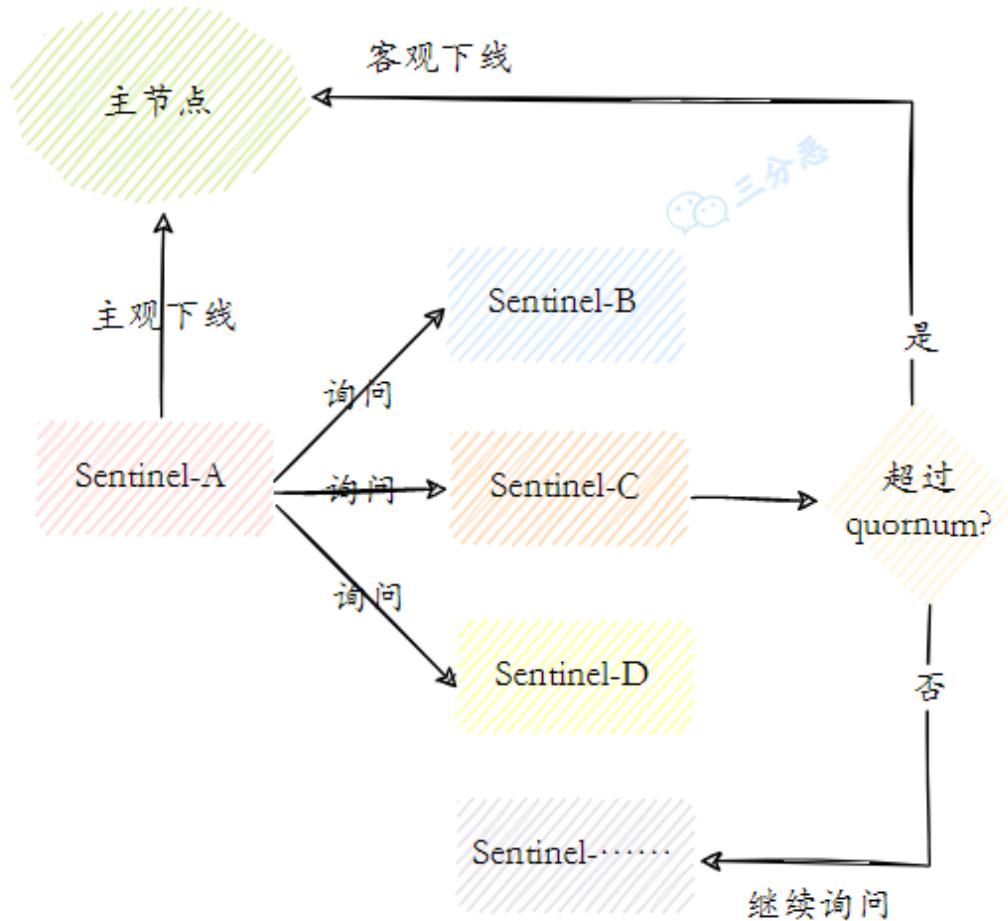


Redis Sentinel通过三个定时监控任务完成对各个节点发现和监控：

1. 每隔10秒，每个Sentinel节点会向主节点和从节点发送info命令获取最新的拓扑结构
2. 每隔2秒，每个Sentinel节点会向Redis数据节点的 sentinel : hello 频道上发送该Sentinel节点对于主节点的判断以及当前Sentinel节点的信息
3. 每隔1秒，每个Sentinel节点会向主节点、从节点、其余Sentinel节点发送一条ping命令做一次心跳检测，来确认这些节点当前是否可达

- 主观下线和客观下线

主观下线就是哨兵节点认为某个节点有问题，客观下线就是超过一定数量的哨兵节点认为主节点有问题。



1. 主观下线

每个Sentinel节点会每隔1秒对主节点、从节点、其他Sentinel节点发送ping命令做心跳检测，当这些节点超过 `down-after-milliseconds` 没有进行有效回复，Sentinel 节点就会对该节点做失败判定，这个行为叫做主观下线。

2. 客观下线

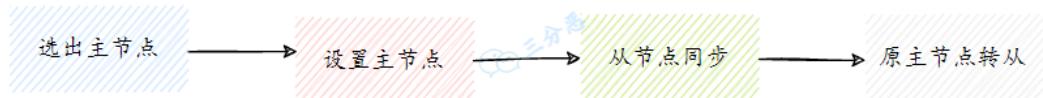
当Sentinel主观下线的节点是主节点时，该Sentinel节点会通过`sentinel is- master-down-by-addr`命令向其他Sentinel节点询问对主节点的判断，当超过 `<quorum>` 个数，Sentinel节点认为主节点确实有问题，这时该Sentinel节点会做出客观下线的决定

- 领导者Sentinel节点选举

Sentinel节点之间会做一个领导者选举的工作，选出一个Sentinel节点作为领导者进行故障转移的工作。Redis使用了Raft算法实现领导者选举。

- 故障转移

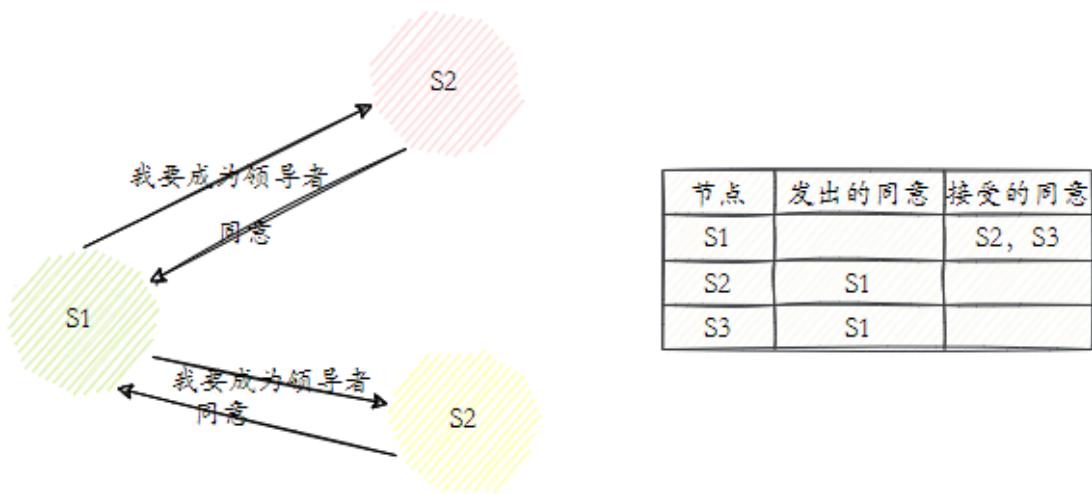
领导者选举出的Sentinel节点负责故障转移，过程如下：



1. 在从节点列表中选出一个节点作为新的主节点，这一步是相对复杂一些的一步
2. Sentinel领导者节点会对第一步选出来的从节点执行slaveof no one命令让其成为主节点
3. Sentinel领导者节点会向剩余的从节点发送命令，让它们成为新主节点的从节点
4. Sentinel节点集合会将原来的主节点更新为从节点，并保持着对其关注，当其恢复后命令它去复制新的主节点

20. 领导者Sentinel节点选举了解吗？

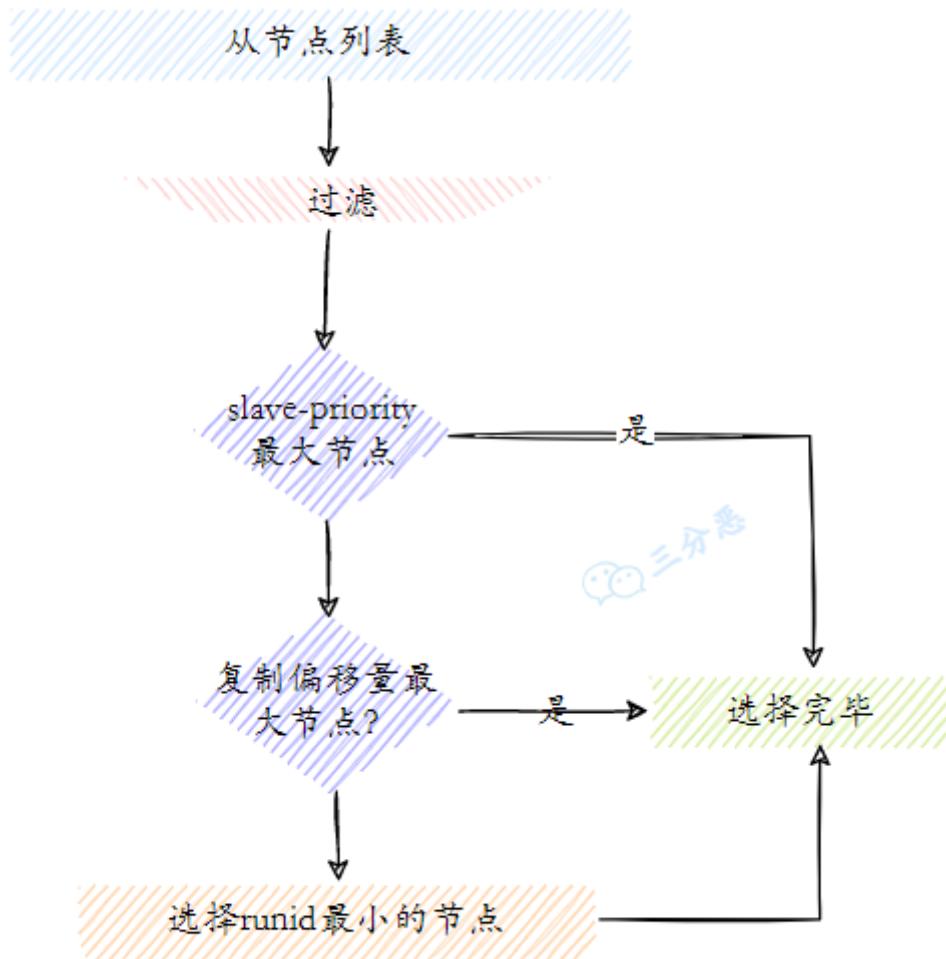
Redis使用了Raft算法实现领导者选举，大致流程如下：



1. 每个在线的Sentinel节点都有资格成为领导者，当它确认主节点主观下线时候，会向其他Sentinel节点发送sentinel is-master-down-by-addr命令，要求将自己设置为领导者。
2. 收到命令的Sentinel节点，如果没有同意过其他Sentinel节点的sentinel is-master-down-by-addr命令，将同意该请求，否则拒绝。
3. 如果该Sentinel节点发现自己的票数已经大于等于 $\text{max}(\text{quorum}, \text{num}(\text{sentinels}) / 2 + 1)$ ，那么它将成为领导者。
4. 如果此过程没有选举出领导者，将进入下一次选举。

21.新的主节点是怎样被挑选出来的？

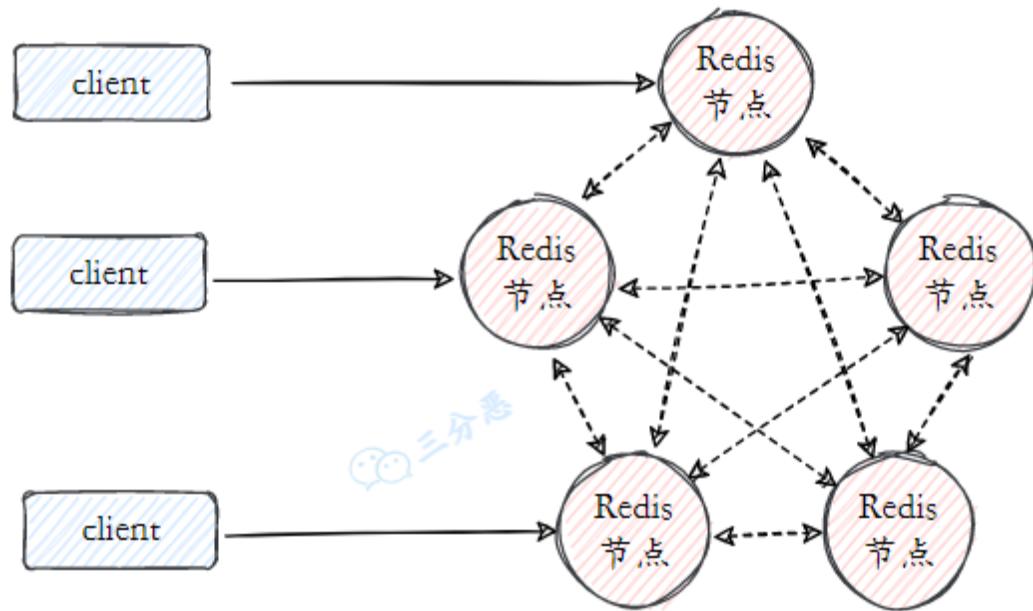
选出新的主节点，大概分为这么几步：



1. 过滤：“不健康”（主观下线、断线）、5秒内没有回复过Sentinel节点ping响应、与主节点失联超过down-after-milliseconds*10秒。
2. 选择slave-priority（从节点优先级）最高的从节点列表，如果存在则返回，不存在则继续。
3. 选择复制偏移量最大的从节点（复制的最完整），如果存在则返回，不存在则继续。
4. 选择runid最小的从节点。

22.Redis 集群了解吗？

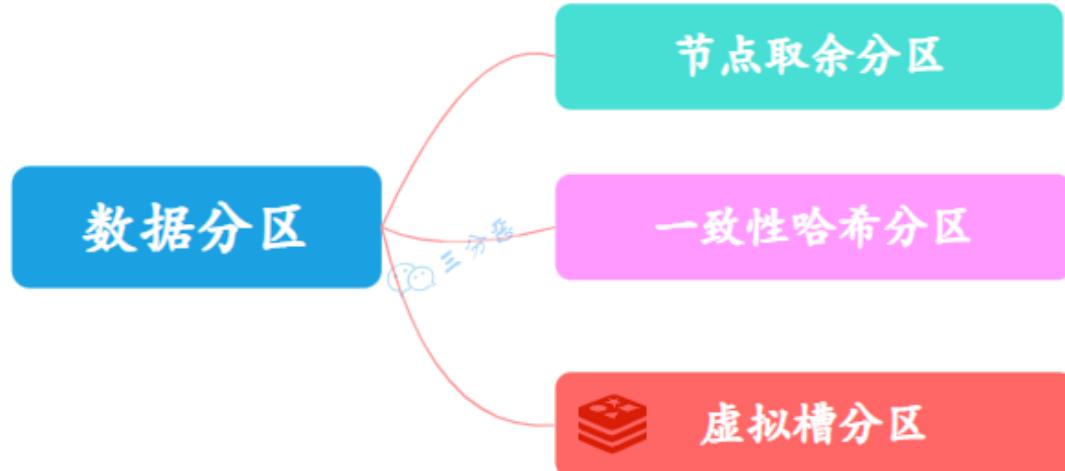
前面说到了主从存在高可用和分布式的问题，哨兵解决了高可用的问题，而集群就是终极方案，一举解决高可用和分布式问题。



1. 数据分区：数据分区（或称数据分片）是集群最核心的功能。集群将数据分散到多个节点，一方面突破了 Redis 单机内存大小的限制，存储容量大大增加；另一方面 每个主节点都可以对外提供读服务和写服务，大大提高了集群的响应能力。
2. 高可用：集群支持主从复制和主节点的 自动故障转移 （与哨兵类似），当任一节点发生故障时，集群仍然可以对外提供服务。

23. 集群中数据如何分区？

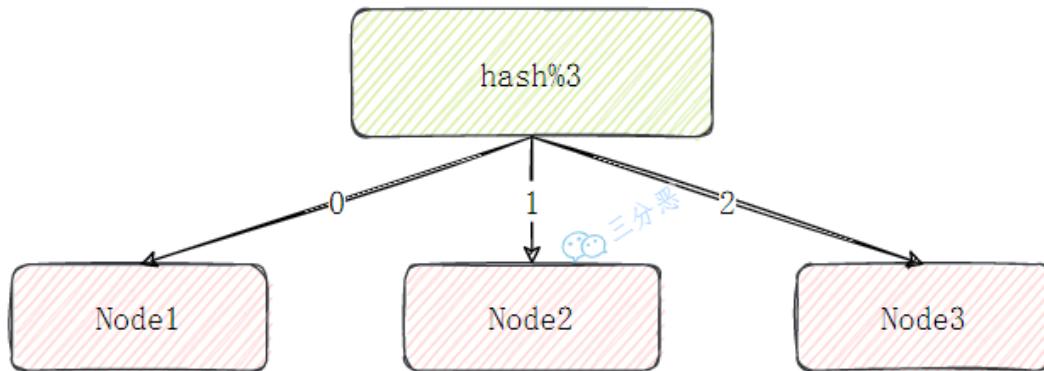
分布式的存储中，要把数据集按照分区规则映射到多个节点，常见的数据分区规则三种：



H6 方案一：节点取余分区

节点取余分区，非常好理解，使用特定的数据，比如Redis的键，或者用户ID之类，对响应的hash值取余： $\text{hash}(\text{key}) \% N$ ，来确定数据映射到哪一个节点上。

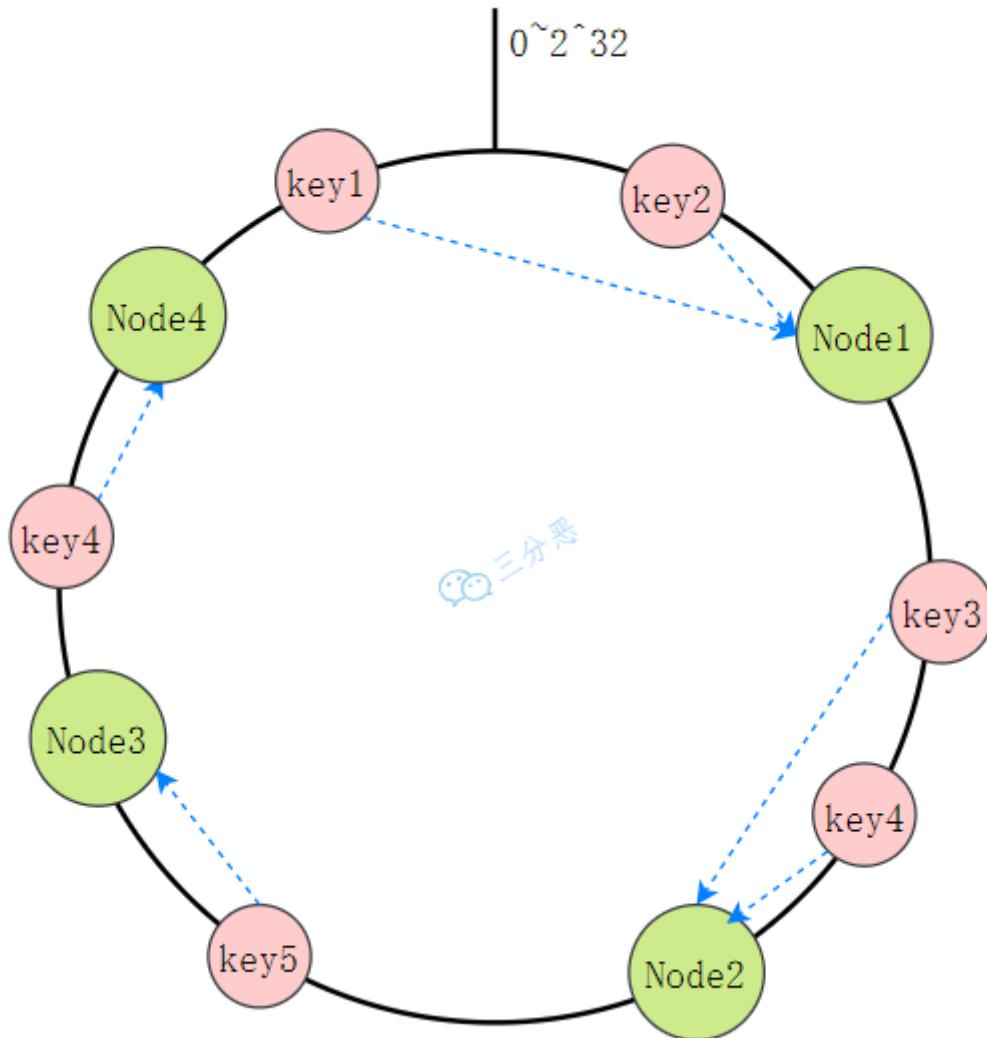
不过该方案最大的问题是，当节点数量变化时，如扩容或收缩节点，数据节点映射关系需要重新计算，会导致数据的重新迁移。



H6 方案二：一致性哈希分区

将整个 Hash 值空间组织成一个虚拟的圆环，然后将缓存节点的 IP 地址或者主机名做 Hash 取值后，放置在这个圆环上。当我们需要确定某一个 Key 需要存取到哪个节点上的时候，先对这个 Key 做同样的 Hash 取值，确定在环上的位置，然后按照顺时针方向在环上“行走”，遇到的第一个缓存节点就是要访问的节点。

比如说下面这张图里面，Key 1 和 Key 2 会落入到 Node 1 中，Key 3、Key 4 会落入到 Node 2 中，Key 5 落入到 Node 3 中，Key 6 落入到 Node 4 中。



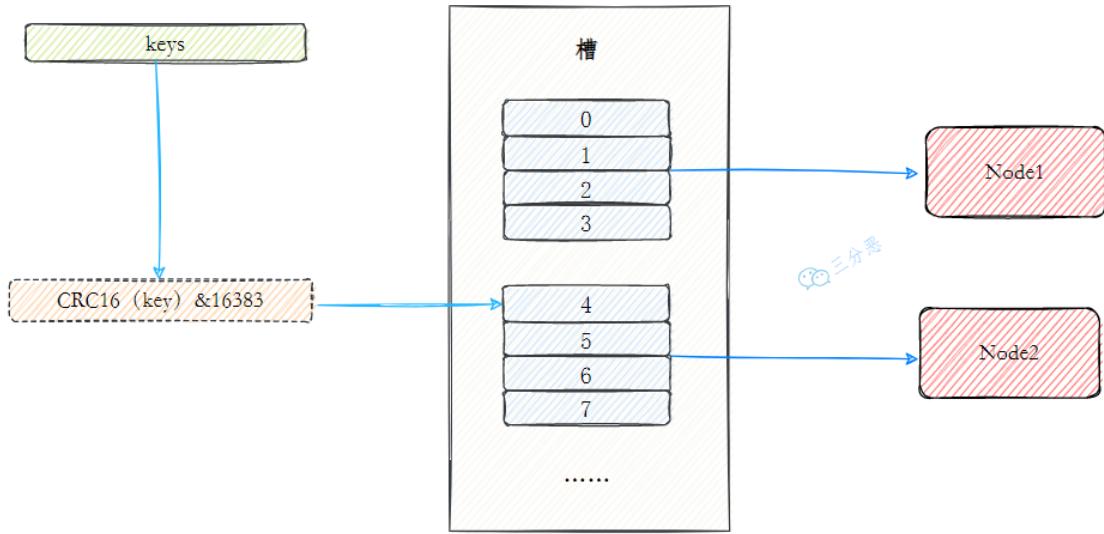
这种方式相比节点取余最大的好处在于加入和删除节点只影响哈希环中 相邻的节点，对其他节点无影响。

但它还是存在问题：

- 缓存节点在圆环上分布不平均，会造成部分缓存节点的压力较大
- 当某个节点故障时，这个节点所要承担的所有访问都会被顺移到另一个节点上，会对后面这个节点造成压力。

H6 方案三：虚拟槽分区

这个方案一致性哈希分区的基础上，引入了 **虚拟节点** 的概念。Redis 集群使用的便是该方案，其中的虚拟节点称为 **槽（slot）**。槽是介于数据和实际节点之间的虚拟概念，每个实际节点包含一定数量的槽，每个槽包含哈希值在一定范围内的数据。



在使用了槽的一致性哈希分区中，槽是数据管理和迁移的基本单位。槽解耦了数据和实际节点之间的关系，增加或删除节点对系统的影响很小。仍以上图为例，系统中有 4 个实际节点，假设为其分配 16 个槽(0-15)；

- 槽 0-3 位于 node1；4-7 位于 node2；以此类推....

如果此时删除 node2，只需要将槽 4-7 重新分配即可，例如槽 4-5 分配给 node1，槽 6 分配给 node3，槽 7 分配给 node4，数据在其他节点的分布仍然较为均衡。

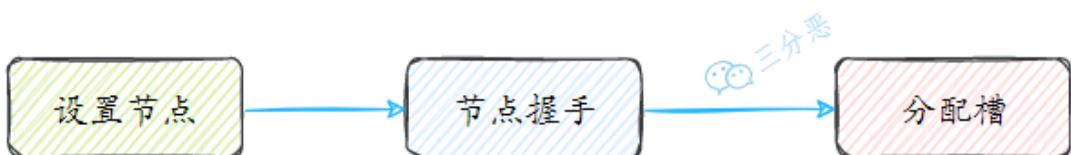
24.能说说Redis集群的原理吗？

Redis集群通过数据分区来实现数据的分布式存储，通过自动故障转移实现高可用。

H6 集群创建

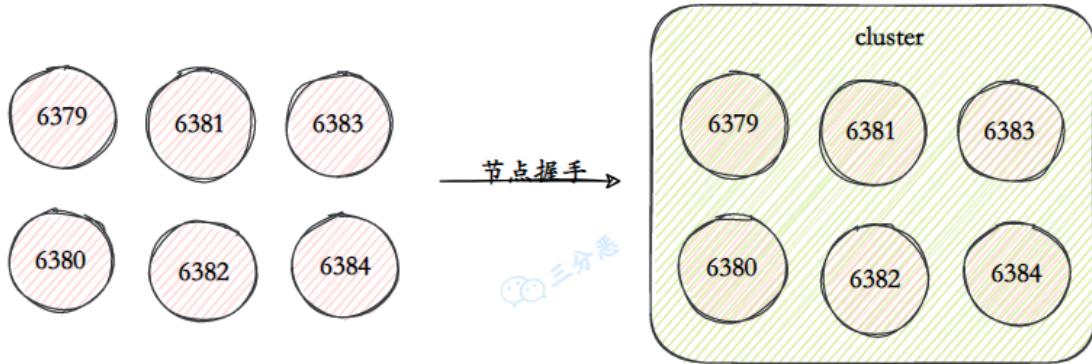
数据分区是在集群创建的时候完成的。

Redis集群创建



设置节点

Redis集群一般由多个节点组成，节点数量至少为6个才能保证组成完整高可用的集群。每个节点需要开启配置cluster-enabled yes，让Redis运行在集群模式下。

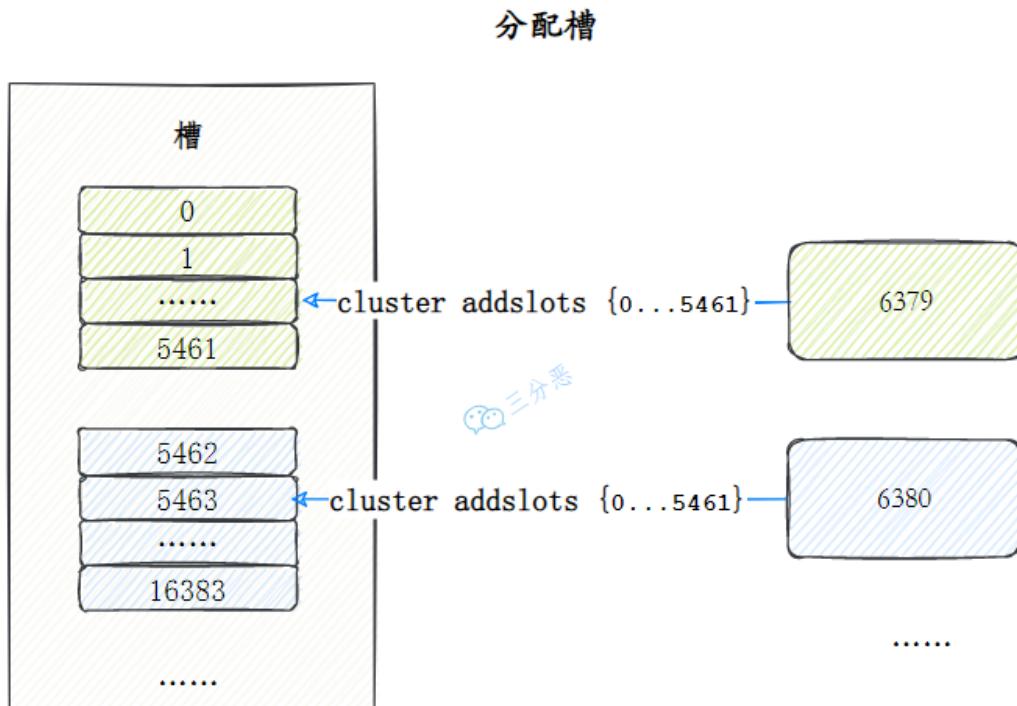


节点握手

节点握手是指一批运行在集群模式下的节点通过Gossip协议彼此通信，达到感知对方的过程。节点握手是集群彼此通信的第一步，由客户端发起命令：cluster meet{ip} {port}。完成节点握手之后，一个个的Redis节点就组成了一个多节点的集群。

分配槽（slot）

Redis集群把所有的数据映射到16384个槽中。每个节点对应若干个槽，只有当节点分配了槽，才能响应和这些槽关联的键命令。通过 cluster addslots命令为节点分配槽。

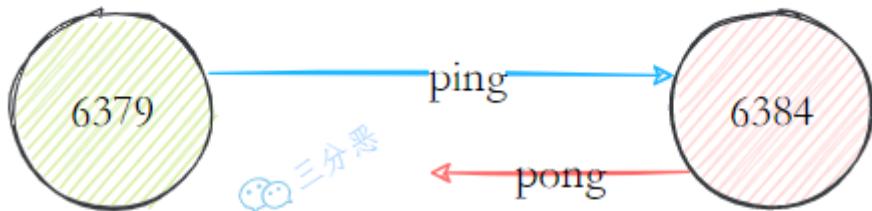


H6 故障转移

Redis集群的故障转移和哨兵的故障转移类似，但是Redis集群中所有的节点都要承担状态维护的任务。

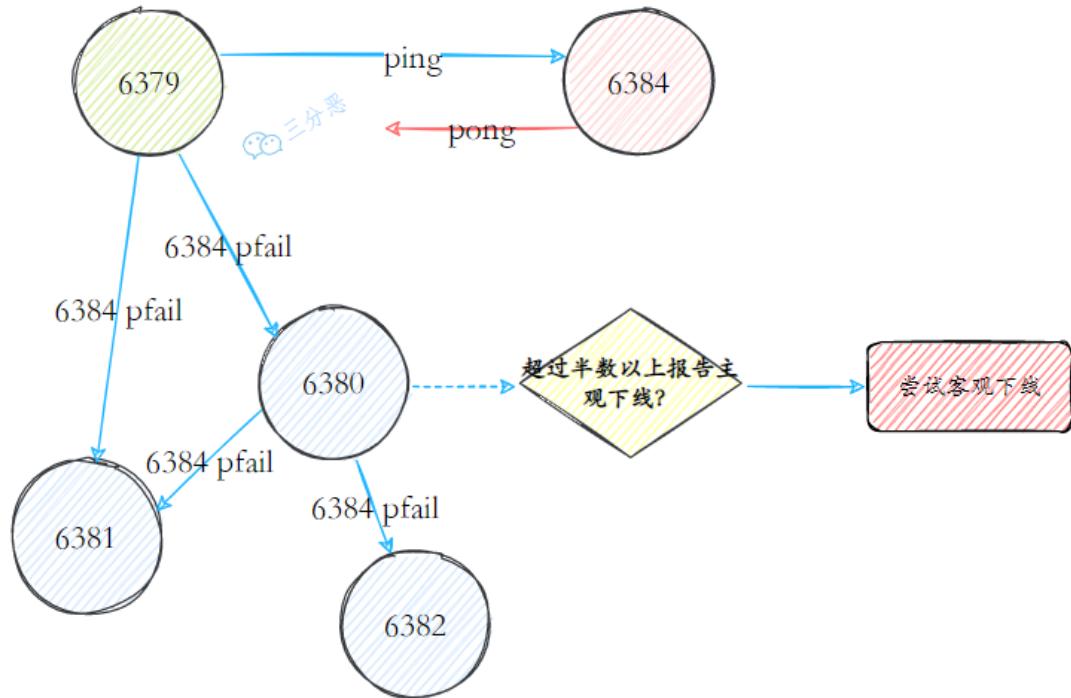
故障发现

Redis集群内节点通过ping/pong消息实现节点通信，集群中每个节点都会定期向其他节点发送ping消息，接收节点回复pong消息作为响应。如果在cluster-node-timeout时间内通信一直失败，则发送节点会认为接收节点存在故障，把接收节点标记为主观下线（pfail）状态。



主观下线

当某个节点判断另一个节点主观下线后，相应的节点状态会跟随消息在集群内传播。通过Gossip消息传播，集群内节点不断收集到故障节点的下线报告。当半数以上持有槽的主节点都标记某个节点是主观下线时。触发客观下线流程。



故障恢复

故障节点变为客观下线后，如果下线节点是持有槽的主节点则需要在它的从节点中选出一个替换它，从而保证集群的高可用。

故障恢复流程



1. 资格检查

每个从节点都要检查最后与主节点断线时间，判断是否有资格替换故障的主节点。

2. 准备选举时间

当从节点符合故障转移资格后，更新触发故障选举的时间，只有到达该时间后才能执行后续流程。

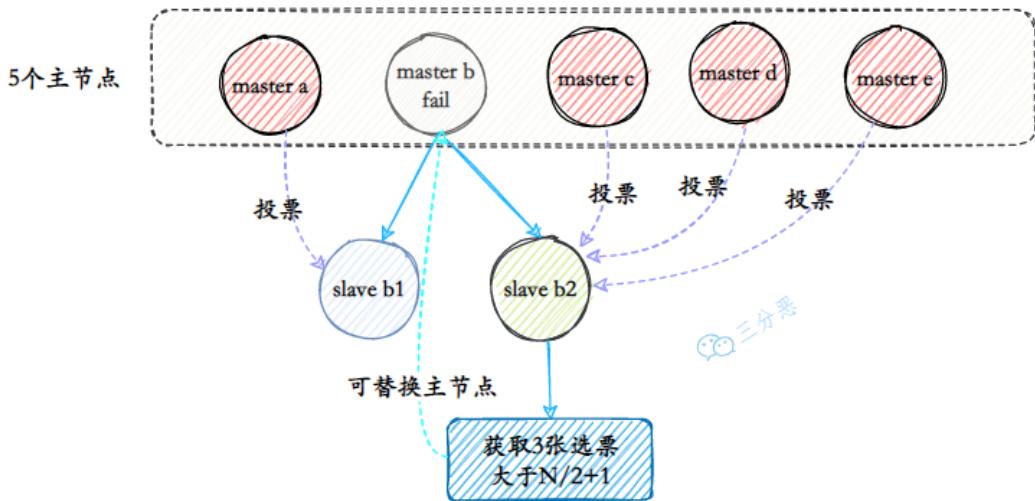
3. 发起选举

当从节点定时任务检测到达故障选举时间（`failover_auth_time`）到达后，发起选举流程。

4. 选举投票

持有槽的主节点处理故障选举消息。投票过程其实是一个领导者选举的过程，如集群内有N个持有槽的主节点代表有N张选票。由于在每个配置纪元内持有槽的主节点只能投票给一个从节点，因此只能有一个从节点获得 $N/2+1$ 的选票，保证

能够找出唯一的从节点。



5. 替换主节点

当从节点收集到足够的选票之后，触发替换主节点操作。

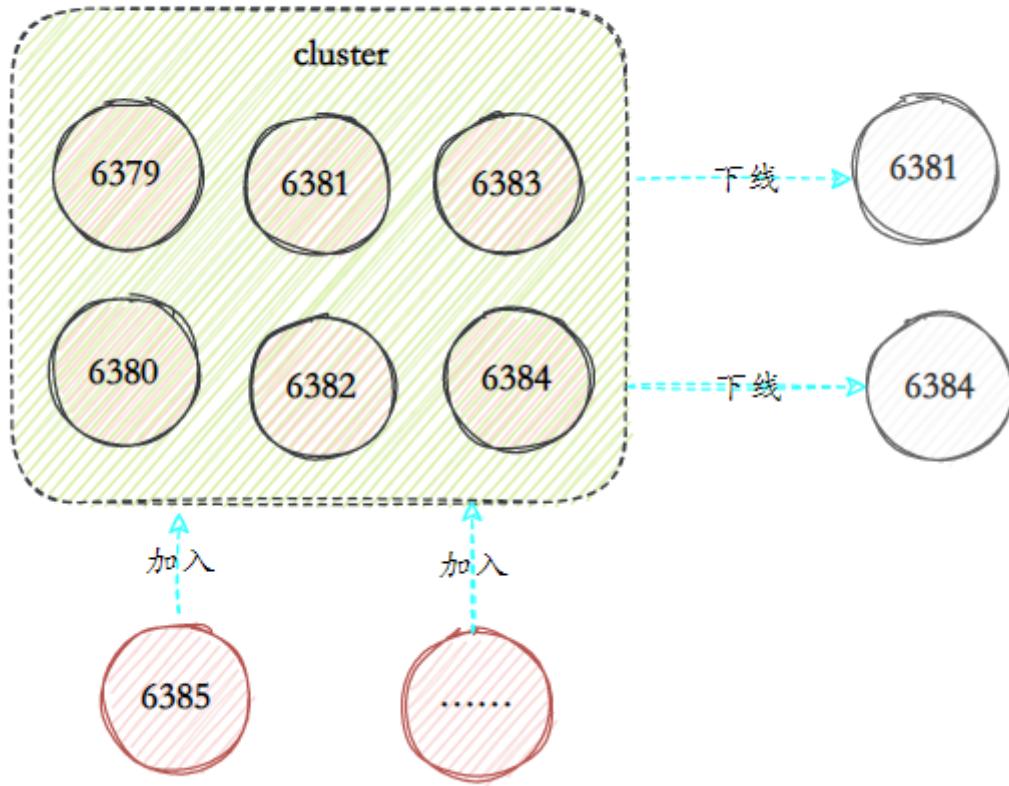
部署Redis集群至少需要几个物理节点？

在投票选举的环节，故障主节点也算在投票数内，假设集群内节点规模是3主3从，其中有2个主节点部署在一台机器上，当这台机器宕机时，由于从节点无法收集到 $3/2+1$ 个主节点选票将导致故障转移失败。这个问题也适用于故障发现环节。因此部署集群时所有主节点最少需要部署在3台物理机上才能避免单点问题。

25. 说说集群的伸缩？

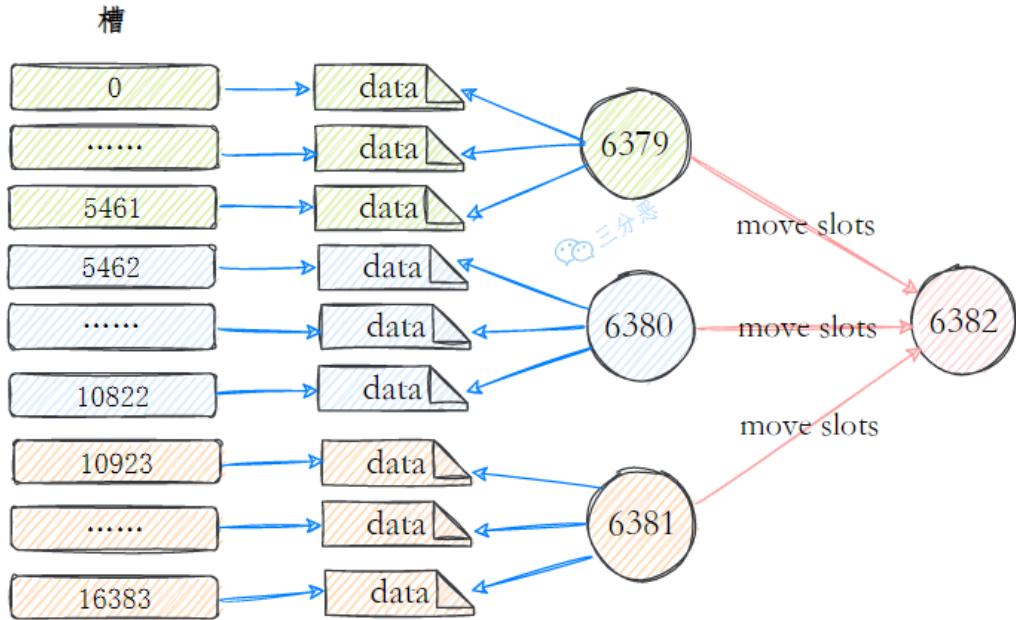
Redis集群提供了灵活的节点扩容和收缩方案，可以在不影响集群对外服务的情况下，为集群添加节点进行扩容也可以下线部分节点进行缩容。

集群节点上下线



其实，集群扩容和缩容的关键点，就在于槽和节点的对应关系，扩容和缩容就是将一部分 **槽** 和 **数据** 迁移给新节点。

例如下面一个集群，每个节点对应若干个槽，每个槽对应一定的数据，如果希望加入1个节点希望实现集群扩容时，需要通过相关命令把一部分槽和内容迁移给新节点。



缩容也是类似，先把槽和数据迁移到其它节点，再把对应的节点下线。

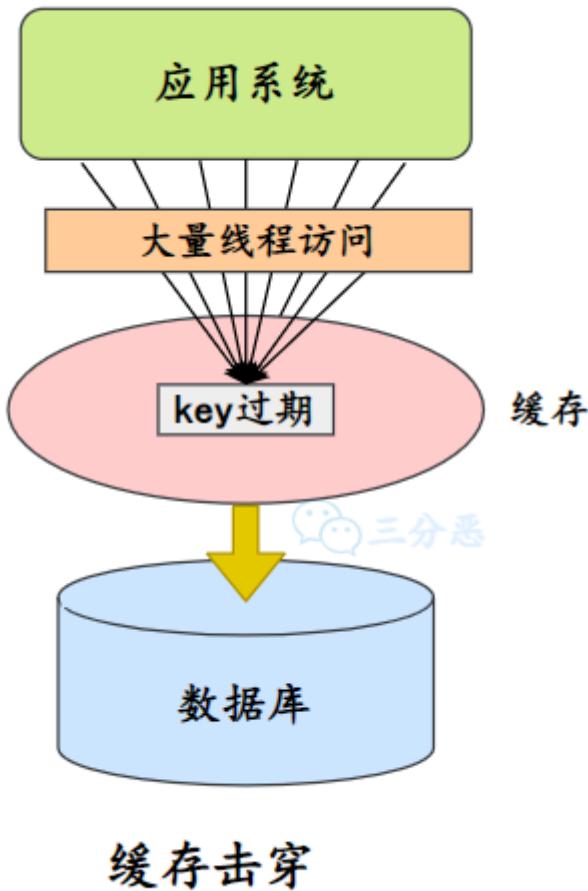
缓存设计

26.什么是缓存击穿、缓存穿透、缓存雪崩？

PS:这是多年黄历的老八股了，一定要理解清楚。

H6 缓存击穿

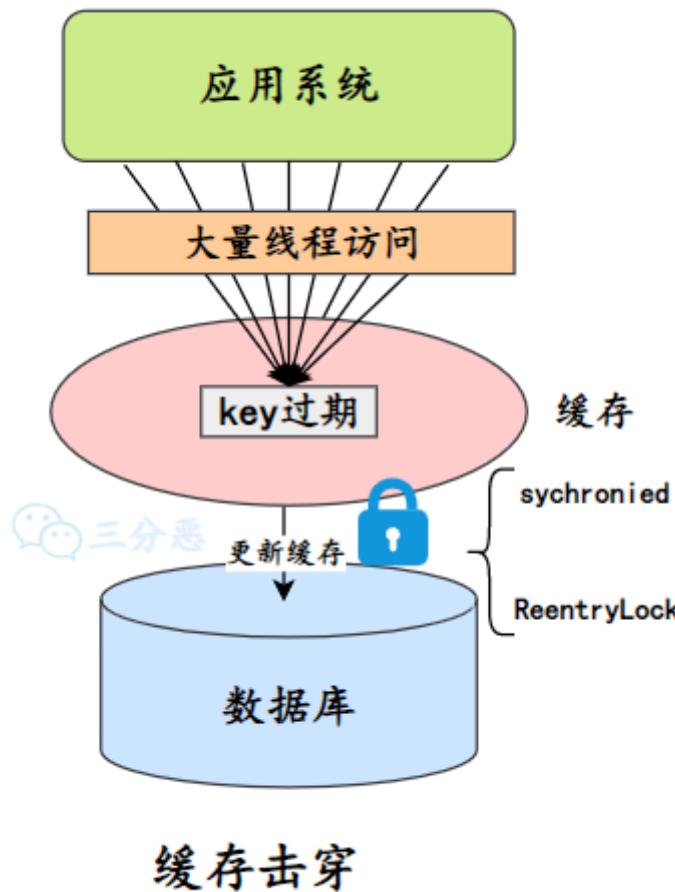
一个并发访问量比较大的key在某个时间过期，导致所有的请求直接打在DB上。



解决方案：

1. 加锁更新，比如请求查询A，发现缓存中没有，对A这个key加锁，同时去数据库查询数据，写入缓存，再返回给用户，这样后面的请求就可以从缓存中拿到数据

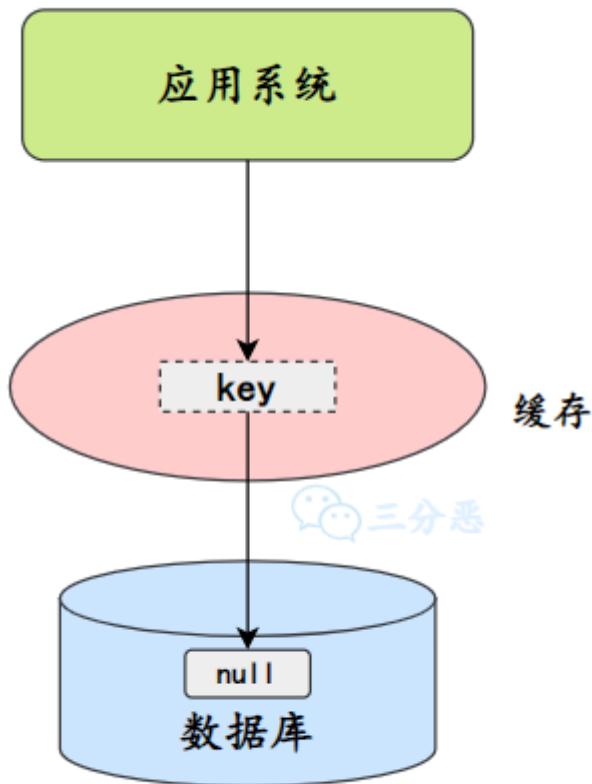
了。



- 将过期时间组合写在value中，通过异步的方式不断的刷新过期时间，防止此类现象。

H6 缓存穿透

缓存穿透指的查询缓存和数据库中都不存在的数据，这样每次请求直接打到数据库，就好像缓存不存在一样。



缓存穿透

缓存穿透将导致不存在的数据每次请求都要到存储层去查询，失去了缓存保护后端存储的意义。

缓存穿透可能会使后端存储负载加大，如果发现大量存储层空命中，可能就是出现了缓存穿透问题。

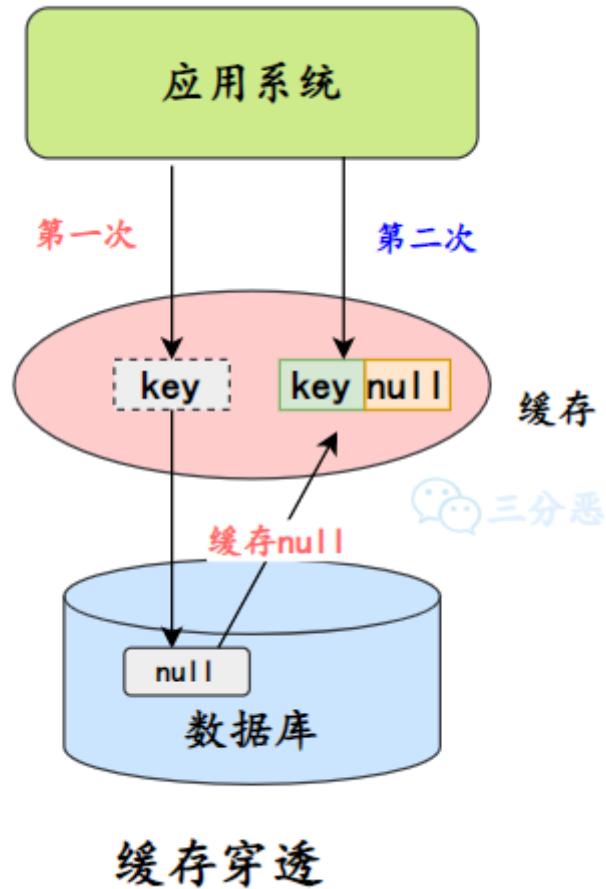
缓存穿透可能有两种原因：

1. 自身业务代码问题
2. 恶意攻击，爬虫造成空命中

它主要有两种解决办法：

- 缓存空值/默认值

一种方式是在数据库不命中之后，把一个空对象或者默认值保存到缓存，之后再访问这个数据，就会从缓存中获取，这样就保护了数据库。



缓存空值有两大问题：

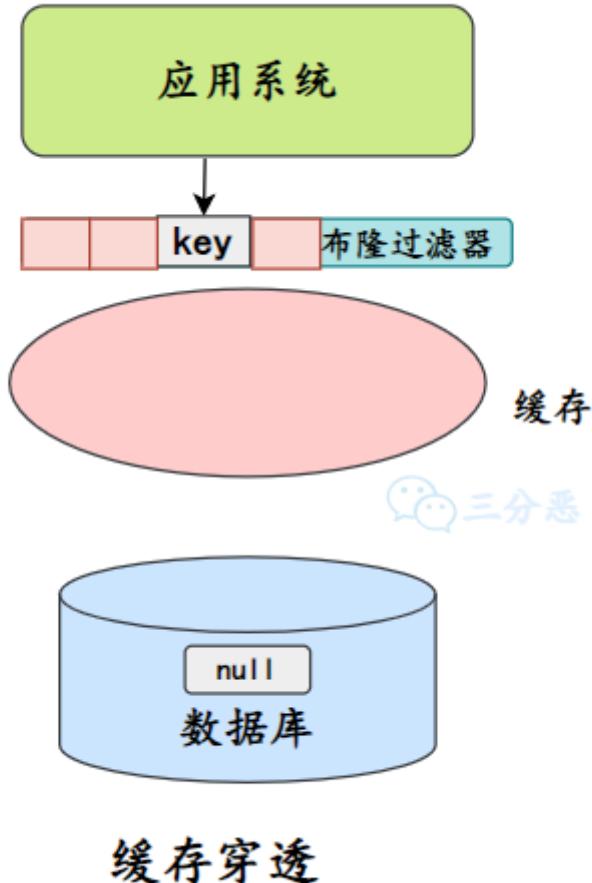
1. 空值做了缓存，意味着缓存层中存了更多的键，需要更多的内存空间（如果是攻击，问题更严重），比较有效的方法是针对这类数据设置一个较短的过期时间，让其自动剔除。
2. 缓存层和存储层的数据会有一段时间窗口的不一致，可能会对业务有一定影响。例如过期时间设置为5分钟，如果此时存储层添加了这个数据，那此段时间就会出现缓存层和存储层数据的不一致。

这时候可以利用消息队列或者其它异步方式清理缓存中的空对象。

- 布隆过滤器

除了缓存空对象，我们还可以在存储和缓存之前，加一个布隆过滤器，做一层过滤。

布隆过滤器里会保存数据是否存在，如果判断数据不能再，就不会访问存储。

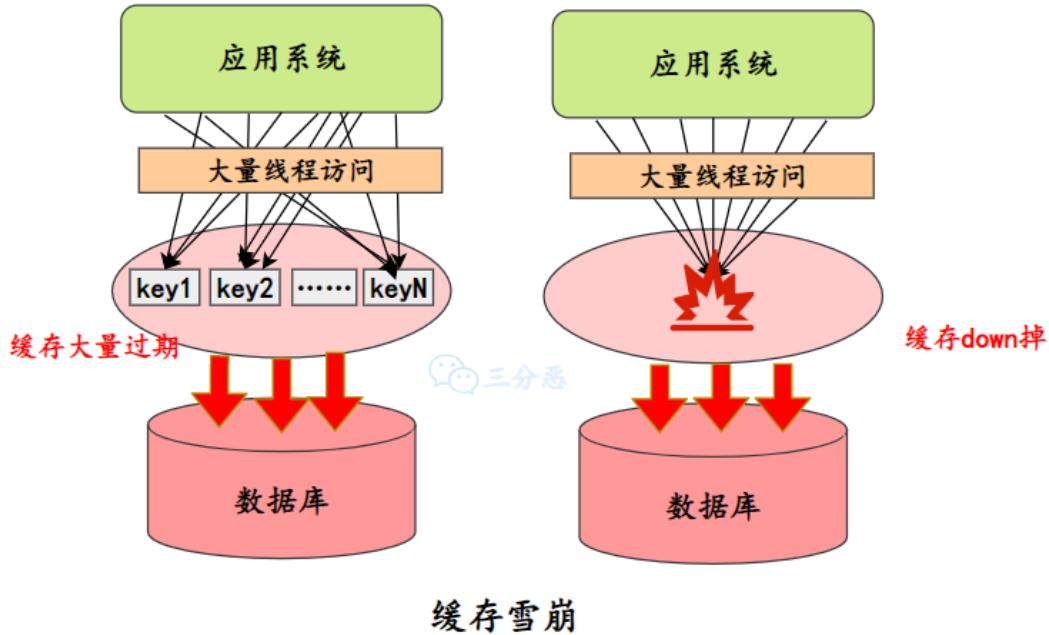


两种解决方案的对比：

解决缓存穿透	适用场景	维护成本
缓存空对象	<ul style="list-style-type: none">• 数据命中不高• 数据频繁实时性高	<ul style="list-style-type: none">• 代码维护简单• 需要较多的缓存空间• 数据不一致
布隆过滤器	<ul style="list-style-type: none">• 数据命中不高• 数据相对固定实时性低	<ul style="list-style-type: none">• 代码维护复杂• 缓存空间占用少

H6 缓存雪崩

某一时刻发生大规模的缓存失效的情况，例如缓存服务宕机、大量key在同一时间过期，这样的后果就是大量的请求进来直接打到DB上，可能导致整个系统的崩溃，称为雪崩。



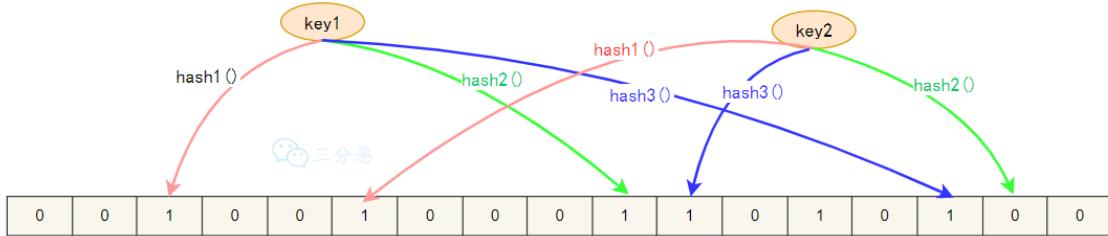
缓存雪崩是三大缓存问题里最严重的一种，我们来看看怎么预防和处理。

- 提高缓存可用性
 1. 集群部署：通过集群来提升缓存的可用性，可以利用Redis本身的Redis Cluster或者第三方集群方案如Codis等。
 2. 多级缓存：设置多级缓存，第一级缓存失效的基础上，访问二级缓存，每一级缓存的失效时间都不同。
- 过期时间
 1. 均匀过期：为了避免大量的缓存在同一时间过期，可以把不同的 key 过期时间随机生成，避免过期时间太过集中。
 2. 热点数据永不过期。
- 熔断降级
 1. 服务熔断：当缓存服务器宕机或超时响应时，为了防止整个系统出现雪崩，暂时停止业务服务访问缓存系统。
 2. 服务降级：当出现大量缓存失效，而且处在高并发高负荷的情况下，在业务系统内部暂时舍弃对一些非核心的接口和数据的请求，而直接返回一个提前准备好的 fallback（退路）错误处理信息。

27.能说说布隆过滤器吗？

布隆过滤器，它是一个连续的数据结构，每个存储位存储都是一个 **bit**，即 **0** 或者 **1**，来标识数据是否存在。

存储数据的时候，使用K个不同的哈希函数将这个变量映射为bit列表的的K个点，把它们置为1。



我们判断缓存key是否存在，同样，K个哈希函数，映射到bit列表上的K个点，判断是不是1：

- 如果全不是1，那么key不存在；
- 如果都是1，也只是表示key可能存在。

布隆过滤器也有一些缺点：

1. 它在判断元素是否在集合中时是有一定错误几率，因为哈希算法有一定的碰撞的概率。
2. 不支持删除元素。

28.如何保证缓存和数据库数据的一致性？

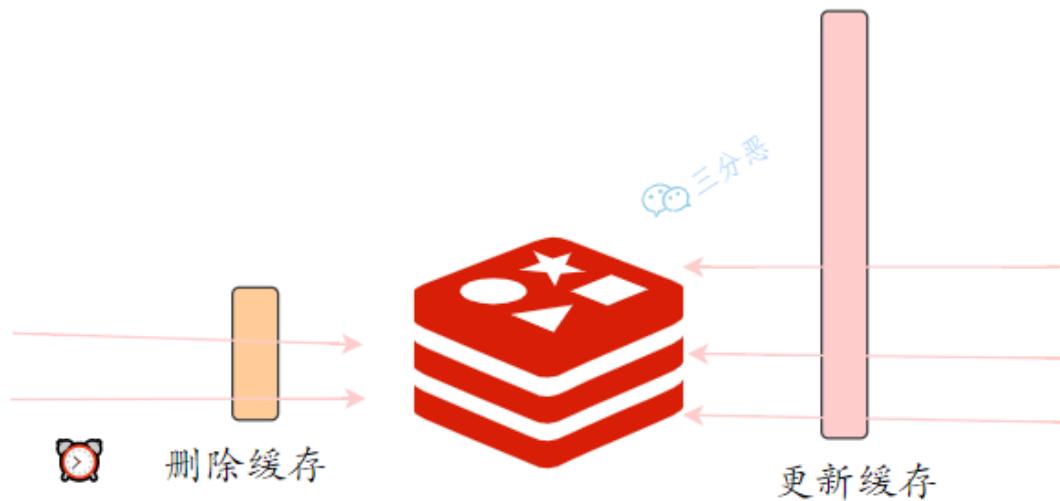
根据CAP理论，在保证可用性和分区容错性的前提下，无法保证一致性，所以缓存和数据库的绝对一致是不可能实现的，只能尽可能保存缓存和数据库的最终一致性。

选择合适的缓存更新策略

1. 删除缓存而不是更新缓存

当一个线程对缓存的key进行写操作的时候，如果其它线程进来读数据库的时候，读到的就是脏数据，产生了数据不一致问题。

相比较而言，删除缓存的速度比更新缓存的速度快很多，所用时间相对也少很多，读脏数据的概率也小很多。

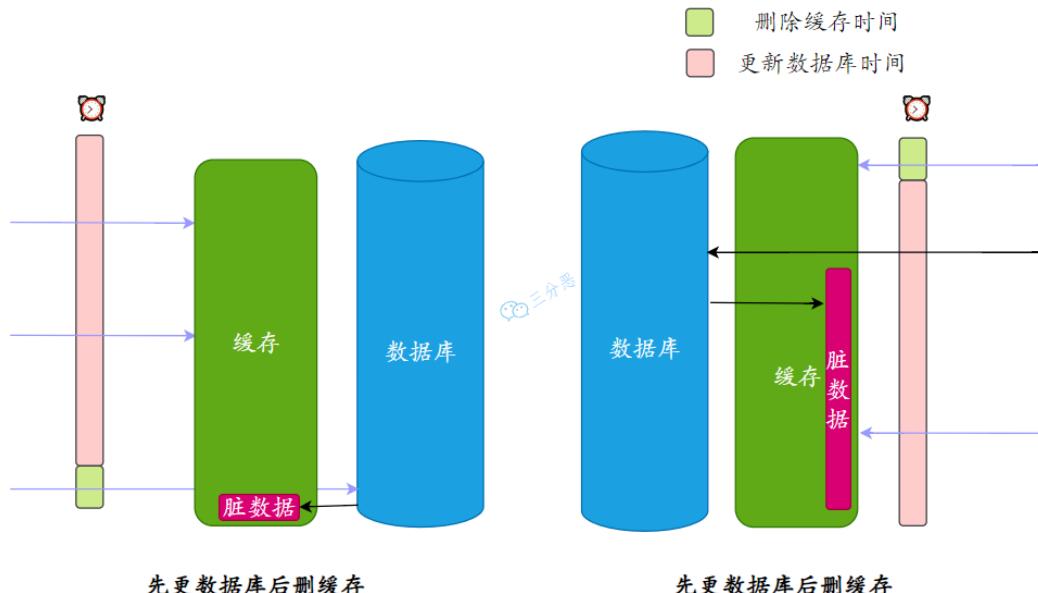


2. 先更数据，后删缓存

先更数据库还是先删缓存？这是一个问题。

更新数据，耗时可能在删除缓存的百倍以上。在缓存中不存在对应的key，数据库又没有完成更新的时候，如果有线程进来读取数据，并写入到缓存，那么在更新成功之后，这个key就是一个脏数据。

毫无疑问，先删缓存，再更数据库，缓存中key不存在的时间的时间更长，有更大的概率会产生脏数据。



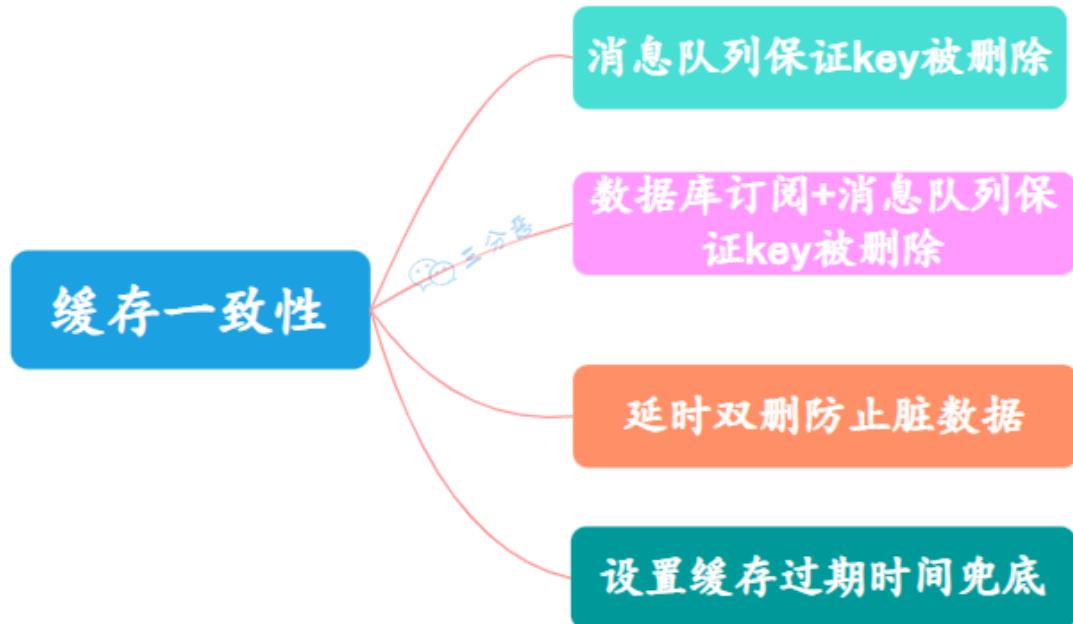
目前最流行的缓存读写策略cache-aside-pattern就是采用先更数据库，再删缓存的方式。

如果不是并发特别高，对缓存依赖性很强，其实一定程度的不一致是可以接受的。

但是如果对一致性要求比较高，那就得想办法保证缓存和数据库中数据一致。

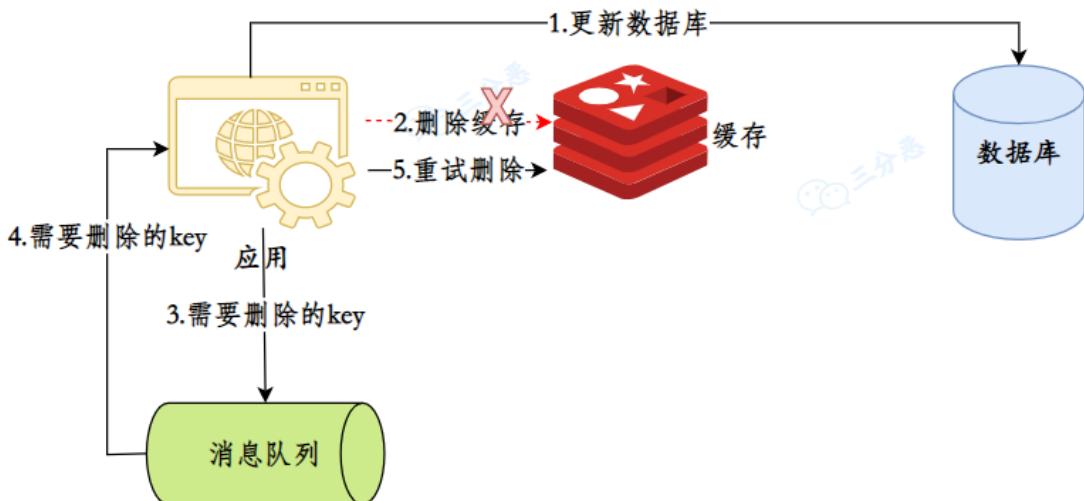
缓存和数据库数据不一致常见的两种原因：

- 缓存key删除失败
- 并发导致写入了脏数据



消息队列保证key被删除

可以引入消息队列，把要删除的key或者删除失败的key丢进消息队列，利用消息队列的重试机制，重试删除对应的key。

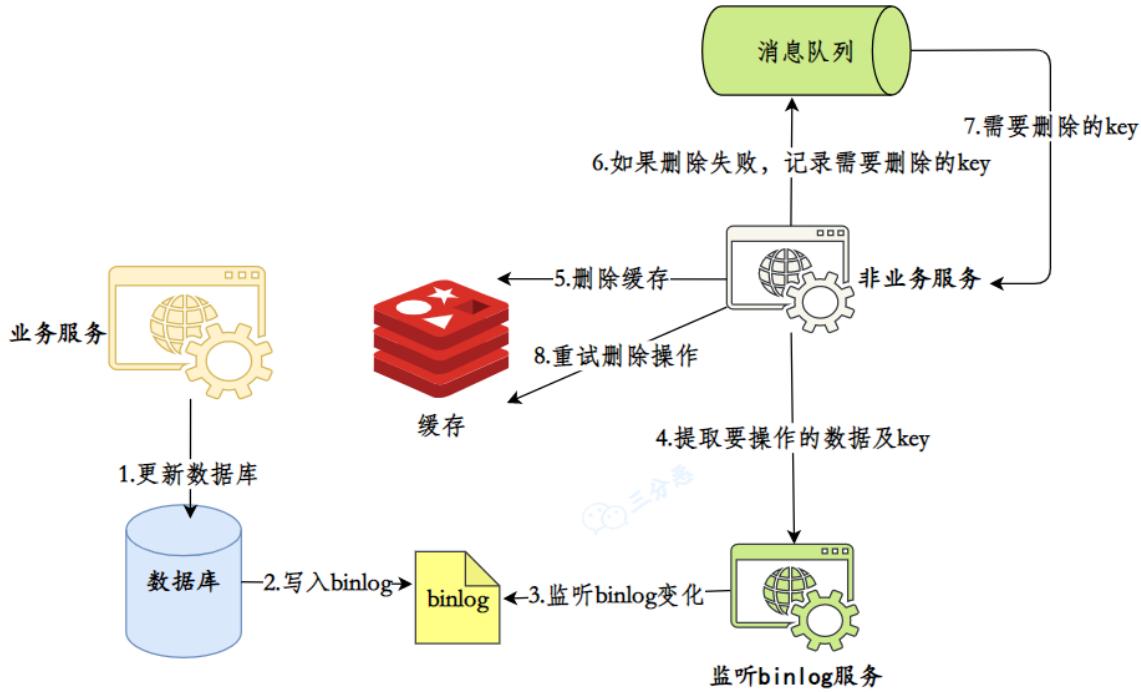


这种方案看起来不错，缺点是对业务代码有一定的侵入性。

数据库订阅+消息队列保证key被删除

可以用一个服务（比如阿里的 canal）去监听数据库的binlog，获取需要操作的数据。

然后用一个公共的服务获取订阅程序传来的信息，进行缓存删除操作。

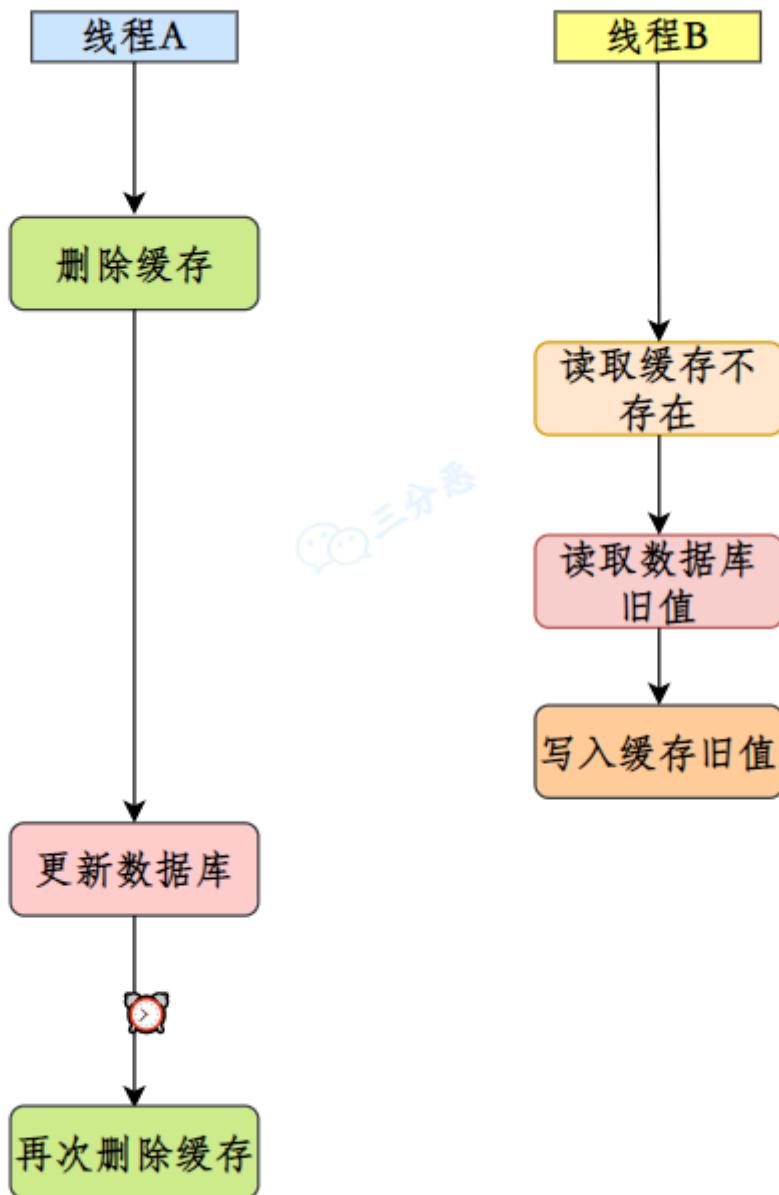


这种方式降低了对业务的侵入，但其实整个系统的复杂度是提升的，适合基建完善的大厂。

延时双删防止脏数据

还有一种情况，是在缓存不存在的时候，写入了脏数据，这种情况在先删缓存，再更数据库的缓存更新策略下发生的比较多，解决方案是延时双删。

简单说，就是在第一次删除缓存之后，过了一段时间之后，再次删除缓存。



这种方式的延时时间设置需要仔细考量和测试。

设置缓存过期时间兜底

这是一个朴素但是有用的办法，给缓存设置一个合理的过期时间，即使发生了缓存数据不一致的问题，它也不会永远不一致下去，缓存过期的时候，自然又会恢复一致。

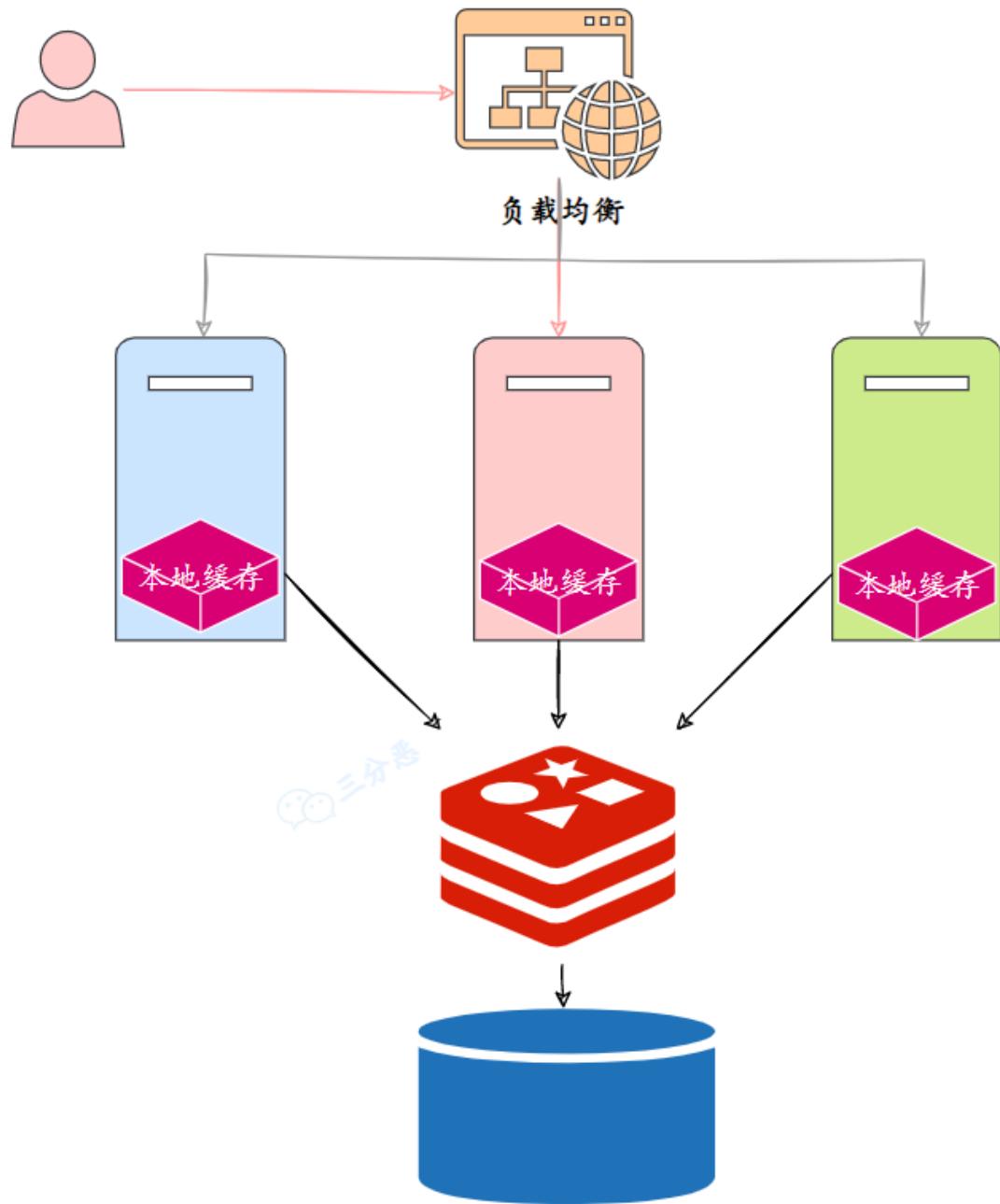
29.如何保证本地缓存和分布式缓存的一致？

PS:这道题面试很少问，但实际工作中很常见。

在日常的开发中，我们常常采用两级缓存：本地缓存+分布式缓存。

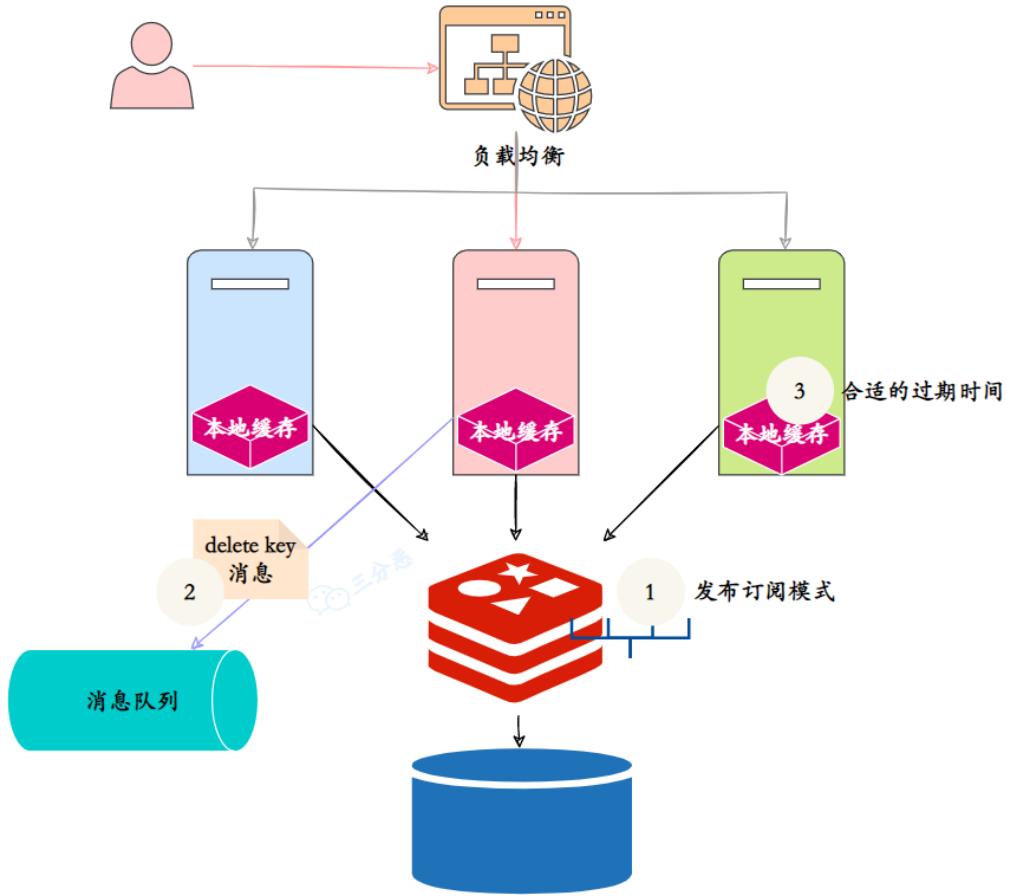
所谓本地缓存，就是对应服务器的内存缓存，比如Caffeine，分布式缓存基本就是采用Redis。

那么问题来了，本地缓存和分布式缓存怎么保持数据一致？



Redis缓存，数据库发生更新，直接删除缓存的key即可，因为对于应用系统而言，它是一种中心化的缓存。

但是本地缓存，它是非中心化的，散落在分布式服务的各个节点上，没法通过客户端的请求删除本地缓存的key，所以得想办法通知集群所有节点，删除对应的本地缓存key。



可以采用消息队列的方式：

1. 采用Redis本身的Pub/Sub机制，分布式集群的所有节点订阅删除本地缓存频道，删除Redis缓存的节点，同事发布删除本地缓存消息，订阅者们订阅到消息后，删除对应的本地key。
但是Redis的发布订阅不是可靠的，不能保证一定删除成功。
2. 引入专业的消息队列，比如RocketMQ，保证消息的可靠性，但是增加了系统的复杂度。
3. 设置适当的过期时间兜底，本地缓存可以设置相对短一些的过期时间。

30. 怎么处理热key？

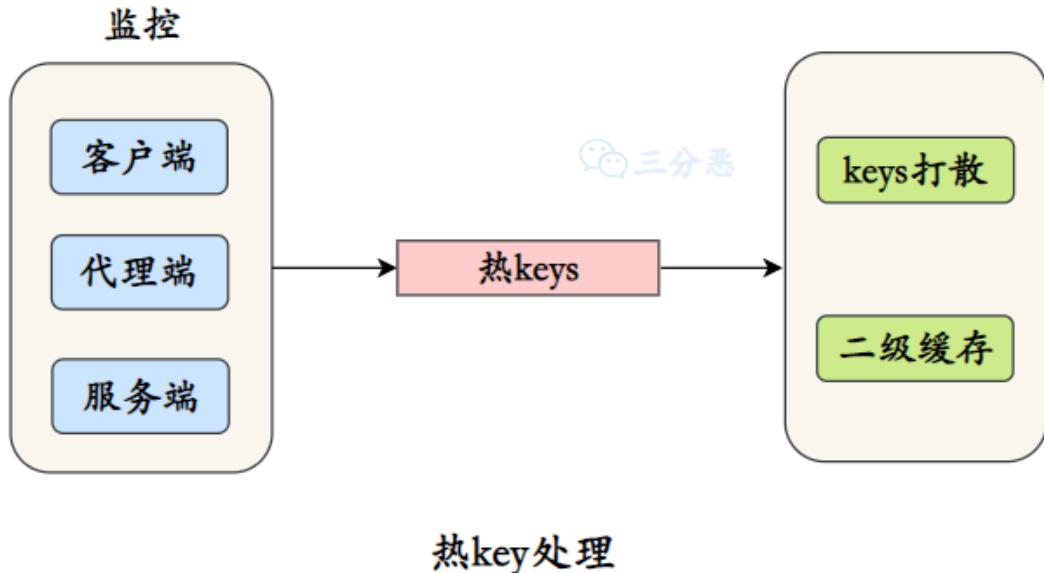
什么是热Key？

所谓的热key，就是访问频率比较高的key。

比如，热门新闻事件或商品，这类key通常有大流量的访问，对存储这类信息的Redis来说，是不小的压力。

假如Redis集群部署，热key可能会造成整体流量的不均衡，个别节点出现OPS过大的情况，极端情况下热点key甚至会超过 Redis本身能够承受的OPS。

怎么处理热key？



对热key的处理，最关键的是对热点key的监控，可以从这些端来监控热点key：

1. 客户端

客户端其实是距离key“最近”的地方，因为Redis命令就是从客户端发出的，例如在客户端设置全局字典（key和调用次数），每次调用Redis命令时，使用这个字典进行记录。

2. 代理端

像Twemproxy、Codis这些基于代理的Redis分布式架构，所有客户端的请求都是通过代理端完成的，可以在代理端进行收集统计。

3. Redis服务端

使用monitor命令统计热点key是很多开发和运维人员首先想到，monitor命令可以监控到Redis执行的所有命令。

只要监控到了热key，对热key的处理就简单了：

1. 把热key打散到不同的服务器，降低压力
2. 加入二级缓存，提前加载热key数据到内存中，如果redis宕机，走内存查询

31.缓存预热怎么做呢？

所谓缓存预热，就是提前把数据库里的数据刷到缓存里，通常有这些方法：

- 1、直接写个缓存刷新页面或者接口，上线时手动操作
- 2、数据量不大，可以在项目启动的时候自动进行加载
- 3、定时任务刷新缓存.

32. 热点key重建？问题？解决？

开发的时候一般使用“缓存+过期时间”的策略，既可以加速数据读写，又保证数据的定期更新，这种模式基本能够满足绝大部分需求。

但是有两个问题如果同时出现，可能就会出现比较大的问题：

- 当前key是一个热点key（例如一个热门的娱乐新闻），并发量非常大。
- 重建缓存不能在短时间完成，可能是一个复杂计算，例如复杂的SQL、多次IO、多个依赖等。在缓存失效的瞬间，有大量线程来重建缓存，造成后端负载加大，甚至可能会让应用崩溃。

怎么处理呢？

要解决这个问题也不是很复杂，解决问题的要点在于：

- 减少重建缓存的次数。
- 数据尽可能一致。
- 较少的潜在危险。

所以一般采用如下方式：

1. 互斥锁（mutex key）

这种方法只允许一个线程重建缓存，其他线程等待重建缓存的线程执行完，重新从缓存获取数据即可。

2. 永远不过期

“永远不过期”包含两层意思：

- 从缓存层面来看，确实没有设置过期时间，所以不会出现热点key过期后产生的问题，也就是“物理”不过期。
- 从功能层面来看，为每个value设置一个逻辑过期时间，当发现超过逻辑过期时间后，会使用单独的线程去构建缓存。

33. 无底洞问题吗？如何解决？

什么是无底洞问题？

2010年，Facebook的Memcache节点已经达到了3000个，承载着TB级别的缓存数据。但开发和运维人员发现了一个问题，为了满足业务要求添加了大量新Memcache节点，但是发现性能不但没有好转反而下降了，当时将这种现象称为缓存的“**无底洞**”现象。

那么为什么会产生这种现象呢？

通常来说添加节点使得Memcache集群性能应该更强了，但事实并非如此。键值数据库由于通常采用哈希函数将key映射到各个节点上，造成key的分布与业务无关，但是由于数据量和访问量的持续增长，造成需要添加大量节点做水平扩容，导致键值分布到更多的节点上，所以无论是Memcache还是Redis的分布式，批量操作通常需要从不同节点上获取，相比于单机批量操作只涉及一次网络操作，分布式批量操作会涉及多次网络时间。

无底洞问题如何优化呢？

先分析一下无底洞问题：

- 客户端一次批量操作会涉及多次网络操作，也就意味着批量操作会随着节点的增多，耗时会不断增大。
- 网络连接数变多，对节点的性能也有一定影响。

常见的优化思路如下：

- 命令本身的优化，例如优化操作语句等。
- 减少网络通信次数。
- 降低接入成本，例如客户端使用长连/连接池、NIO等。

Redis运维

34.Redis报内存不足怎么处理？

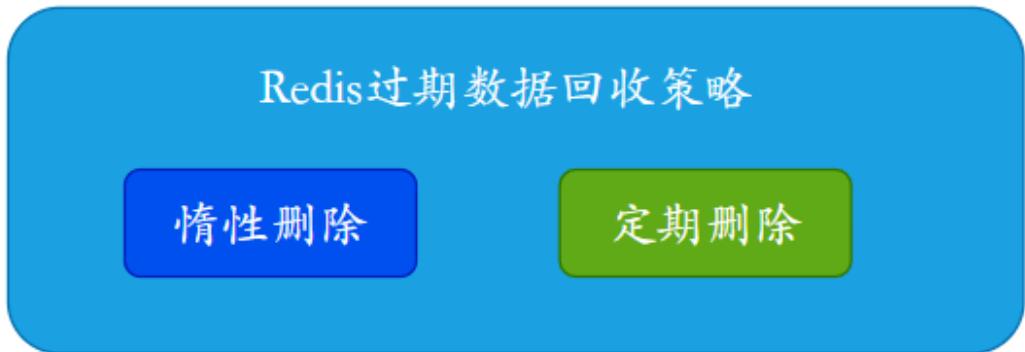
Redis 内存不足有这么几种处理方式：

- 修改配置文件 redis.conf 的 maxmemory 参数，增加 Redis 可用内存
- 也可以通过命令 set maxmemory 动态设置内存上限
- 修改内存淘汰策略，及时释放内存空间

- 使用 Redis 集群模式，进行横向扩容。

35.Redis的过期数据回收策略有哪些？

Redis主要有2种过期数据回收策略：



惰性删除

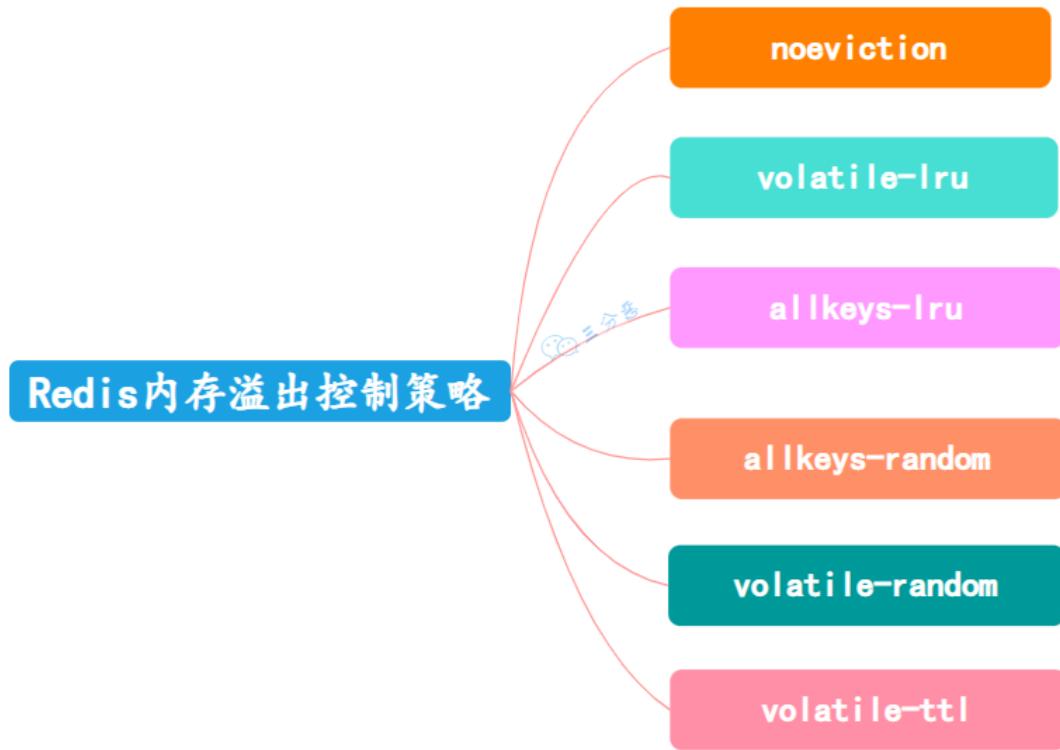
惰性删除指的是当我们查询key的时候才对key进行检测，如果已经达到过期时间，则删除。显然，他有一个缺点就是如果这些过期的key没有被访问，那么他就一直无法被删除，而且一直占用内存。

定期删除

定期删除指的是Redis每隔一段时间对数据库做一次检查，删除里面的过期key。由于不可能对所有key去做轮询来删除，所以Redis会每次随机取一些key去做检查和删除。

36.Redis有哪些内存溢出控制/内存淘汰策略？

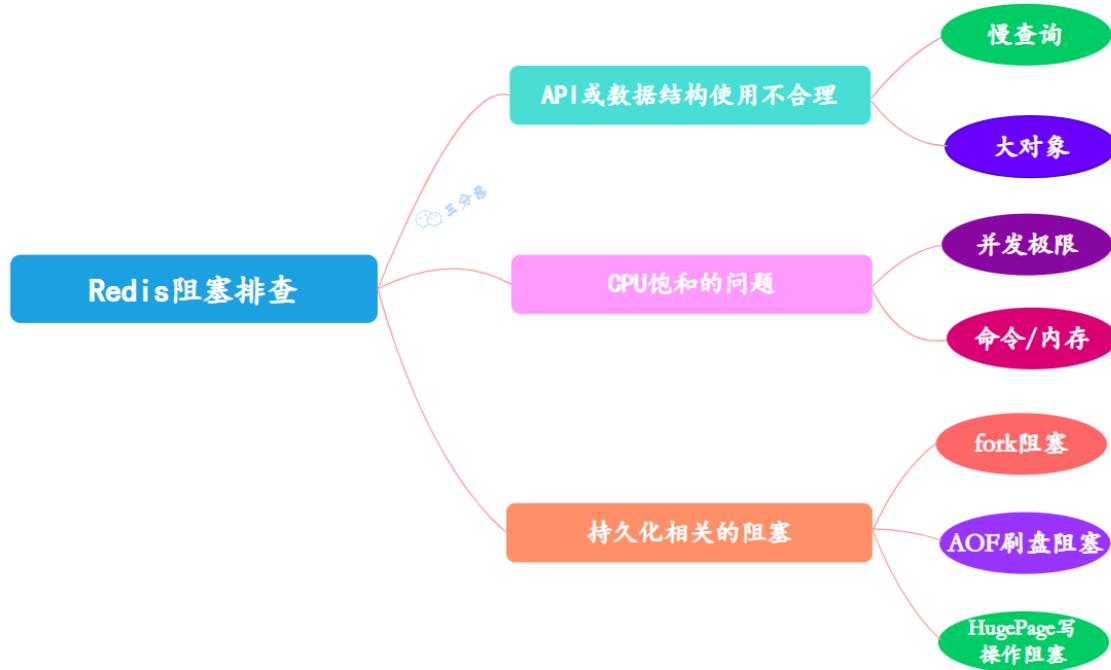
Redis所用内存达到maxmemory上限时会触发相应的溢出控制策略，Redis支持六种策略：



1. noeviction: 默认策略，不会删除任何数据，拒绝所有写入操作并返回客户端错误信息，此时Redis只响应读操作。
2. volatile-lru: 根据LRU算法删除设置了超时属性（expire）的键，直到腾出足够空间为止。如果没有可删除的键对象，回退到noeviction策略。
3. allkeys-lru: 根据LRU算法删除键，不管数据有没有设置超时属性，直到腾出足够空间为止。
4. allkeys-random: 随机删除所有键，直到腾出足够空间为止。
5. volatile-random: 随机删除过期键，直到腾出足够空间为止。
6. volatile-ttl: 根据键值对象的ttl属性，删除最近将要过期数据。如果没有，回退到noeviction策略。

37.Redis阻塞？怎么解决？

Redis发生阻塞，可以从以下几个方面排查：



• API或数据结构使用不合理

通常Redis执行命令速度非常快，但是不合理地使用命令，可能会导致执行速度很慢，导致阻塞，对于高并发的场景，应该尽量避免在大对象上执行算法复杂度超过 $O(n)$ 的命令。

对慢查询的处理分为两步：

1. 发现慢查询：slowlog get{n}命令可以获取最近的n条慢查询命令；
2. 发现慢查询后，可以从两个方向去优化慢查询：
 - 1) 修改为低算法复杂度的命令，如hgetall改为hmget等，禁用keys、sort等命令
 - 2) 调整大对象：缩减大对象数据或把大对象拆分为多个小对象，防止一次命令操作过多的数据。

• CPU饱和的问题

单线程的Redis处理命令时只能使用一个CPU。而CPU饱和是指Redis单核CPU使用率跑到接近100%。

针对这种情况，处理步骤一般如下：

1. 判断当前Redis并发量是否已经达到极限，可以使用统计命令redis-cli-h{ip}-p{port}--stat获取当前Redis使用情况
2. 如果Redis的请求几万+，那么大概就是Redis的OPS已经到了极限，应该做集群化扩展来分摊OPS压力
3. 如果只有几百几千，那么就得排查命令和内存的使用

- 持久化相关的阻塞

对于开启了持久化功能的Redis节点，需要排查是否是持久化导致的阻塞。

1. fork阻塞

fork操作发生在RDB和AOF重写时，Redis主线程调用fork操作产生共享内存的子进程，由子进程完成持久化文件重写工作。如果fork操作本身耗时过长，必然会导致主线程的阻塞。

2. AOF刷盘阻塞

当我们开启AOF持久化功能时，文件刷盘的方式一般采用每秒一次，后台线程每秒对AOF文件做fsync操作。当硬盘压力过大时，fsync操作需要等待，直到写入完成。如果主线程发现距离上一次的fsync成功超过2秒，为了数据安全性它会阻塞直到后台线程执行fsync操作完成。

3. HugePage写操作阻塞

对于开启Transparent HugePages的操作系统，每次写命令引起的复制内存页单位由4K变为2MB，放大了512倍，会拖慢写操作的执行时间，导致大量写操作慢查询。

38. 大key问题了解吗？

Redis使用过程中，有时候会出现大key的情况，比如：

- 单个简单的key存储的value很大，size超过10KB
- hash, set, zset, list 中存储过多的元素（以万为单位）

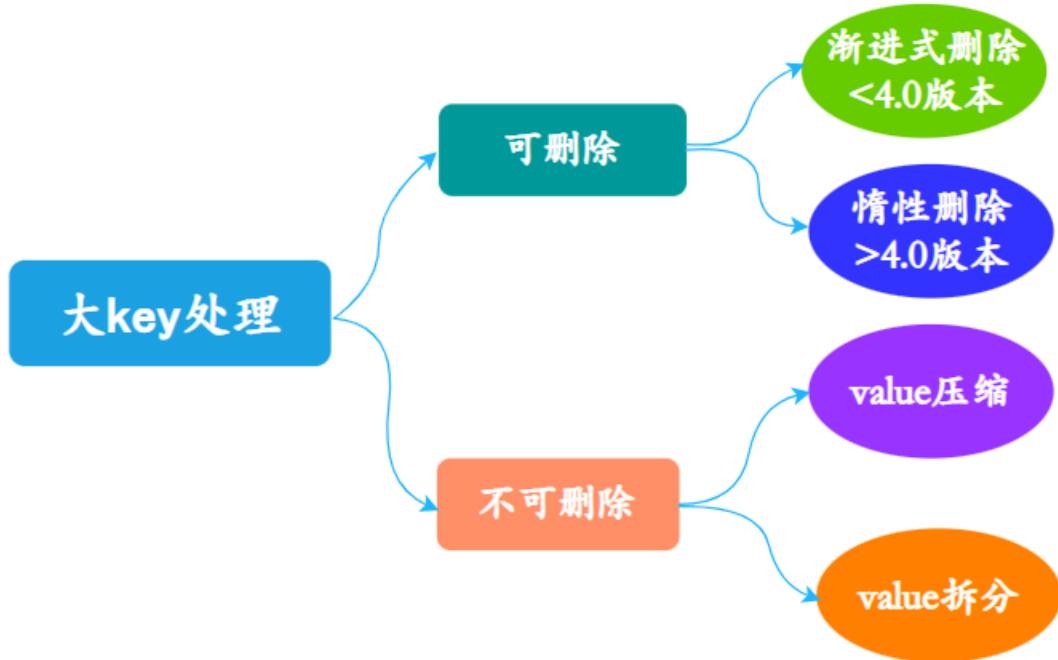
大key会造成什么问题呢？

- 客户端耗时增加，甚至超时
- 对大key进行IO操作时，会严重占用带宽和CPU
- 造成Redis集群中数据倾斜
- 主动删除、被动删等，可能会导致阻塞

如何找到大key？

- bigkeys命令：使用bigkeys命令以遍历的方式分析Redis实例中的所有Key，并返回整体统计信息与每个数据类型中Top1的大Key
- redis-rdb-tools：redis-rdb-tools是由Python写的用来分析Redis的rdb快照文件用的工具，它可以把rdb快照文件生成json文件或者生成报表用来分析Redis的使用详情。

如何处理大key?



• 删除大key

- 当Redis版本大于4.0时，可使用UNLINK命令安全地删除大Key，该命令能够以非阻塞的方式，逐步地清理传入的Key。
- 当Redis版本小于4.0时，避免使用阻塞式命令KEYS，而是建议通过SCAN命令执行增量迭代扫描key，然后判断进行删除。

• 压缩和拆分key

- 当value是string时，比较难拆分，则使用序列化、压缩算法将key的大小控制在合理范围内，但是序列化和反序列化都会带来更多时间上的消耗。
- 当value是string，压缩之后仍然是大key，则需要进行拆分，一个大key分为不同的部分，记录每个部分的key，使用multiget等操作实现事务读取。
- 当value是list/set等集合类型时，根据预估的数据规模来进行分片，不同的元素计算后分到不同的片。

39.Redis常见性能问题和解决方案？

1. Master 最好不要做任何持久化工作，包括内存快照和 AOF 日志文件，特别是不要启用内存快照做持久化。
2. 如果数据比较关键，某个 Slave 开启 AOF 备份数据，策略为每秒同步一次。
3. 为了主从复制的速度和连接的稳定性，Slave 和 Master 最好在同一个局域网内。
4. 尽量避免在压力较大的主库上增加从库。

- Master 调用 BGREWRITEAOF 重写 AOF 文件，AOF 在重写的时候会占大量的 CPU 和内存资源，导致服务 load 过高，出现短暂服务暂停现象。
- 为了 Master 的稳定性，主从复制不要用图状结构，用单向链表结构更稳定，即主从关系为：Master<--Slave1<--Slave2<--Slave3...，这样的结构也方便解决单点故障问题，实现 Slave 对 Master 的替换，也即，如果 Master 挂了，可以立马启用 Slave1 做 Master，其他不变。

Redis应用

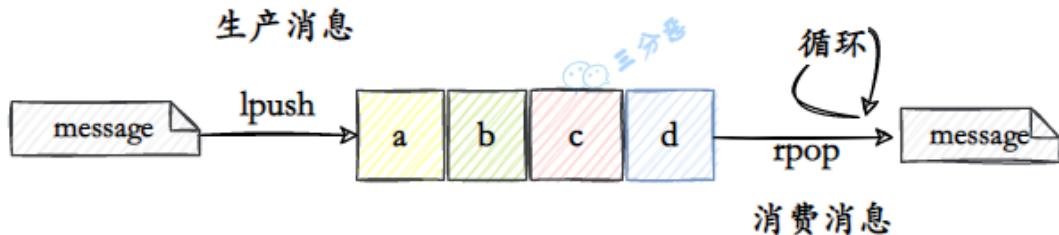
40. 使用Redis 如何实现异步队列？

我们知道redis支持很多种结构的数据，那么如何使用redis作为异步队列使用呢？

一般有以下几种方式：

- 使用list作为队列，lpush生产消息，rpop消费消息

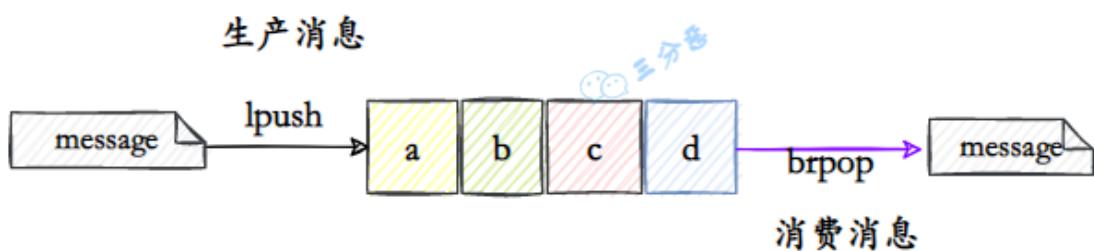
这种方式，消费者死循环rpop从队列中消费消息。但是这样，即使队列里没有消息，也会进行rpop，会导致Redis CPU的消耗。



可以通过让消费者休眠的方式来处理，但是这样又会带来消息的延迟问题。

- 使用list作为队列，lpush生产消息，brpop消费消息

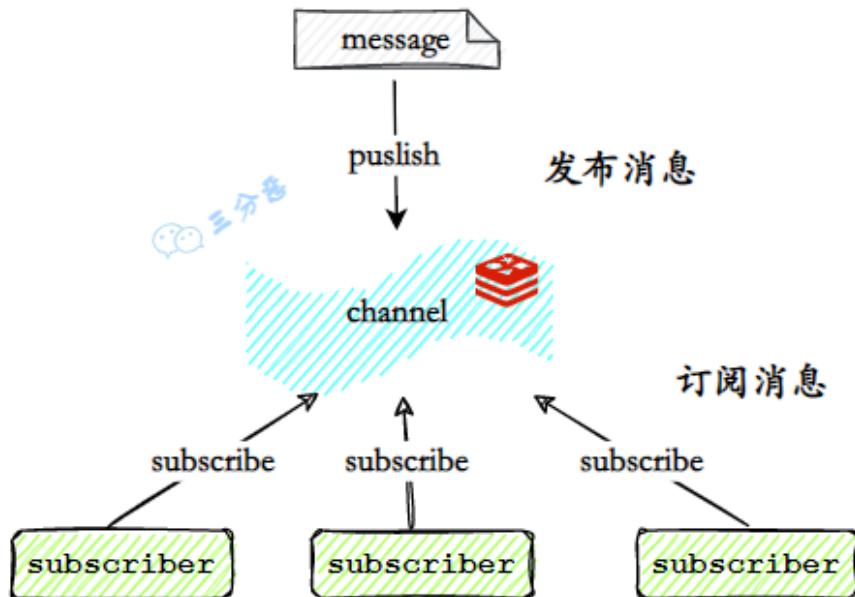
brpop是rpop的阻塞版本，list为空的时候，它会一直阻塞，直到list中有值或者超时。



这种方式只能实现一对一的消息队列。

- 使用Redis的pub/sub来进行消息的发布/订阅

发布/订阅模式可以1: N的消息发布/订阅。发布者将消息发布到指定的频道频道(channel)，订阅相应频道的客户端都能收到消息。



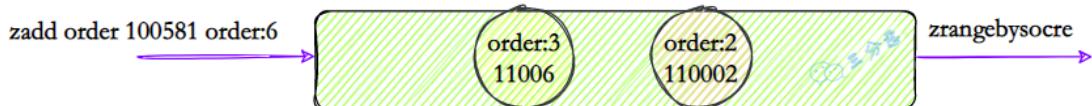
但是这种方式不是可靠的，它不保证订阅者一定能收到消息，也不进行消息的存储。

所以，一般的异步队列的实现还是交给专业的消息队列。

41.Redis 如何实现延时队列？

- 使用zset，利用排序实现

可以使用 zset这个结构，用设置好的时间戳作为score进行排序，使用 zadd score1 value1命令就可以一直往内存中生产消息。再利用 zrangebyscore 查询符合条件的所有待处理的任务，通过循环执行队列任务即可。



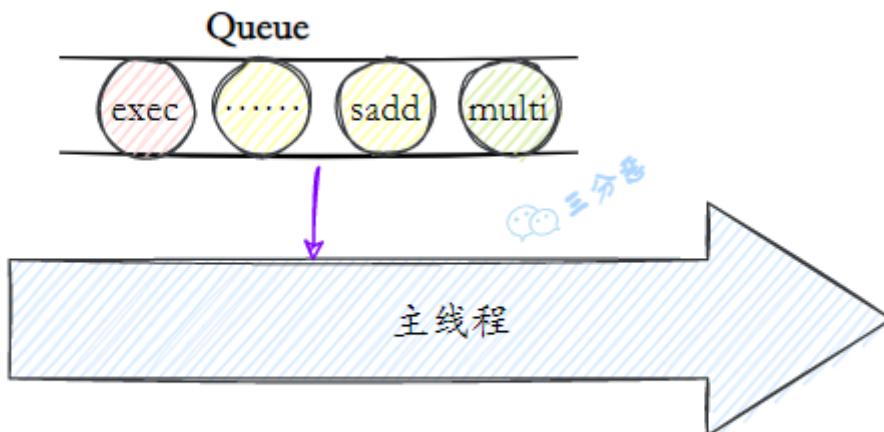
42.Redis 支持事务吗？

Redis提供了简单的事务，但它对事务ACID的支持并不完备。

multi命令代表事务开始，exec命令代表事务结束，它们之间的命令是原子顺序执行的：

```
1 127.0.0.1:6379> multi
2 OK
3 127.0.0.1:6379> sadd user:a:follow user:b
4 QUEUED
5 127.0.0.1:6379> sadd user:b:fans user:a
6 QUEUED
7 127.0.0.1:6379> sismember user:a:follow user:b
8 (integer) 0
9 127.0.0.1:6379> exec 1) (integer) 1
10 2) (integer) 1
```

Redis事务的原理，是所有的指令在 exec 之前不执行，而是缓存在服务器的一个事务队列中，服务器一旦收到 exec 指令，才开始执行整个事务队列，执行完毕后一次性返回所有指令的运行结果。



因为Redis执行命令是单线程的，所以这组命令顺序执行，而且不会被其它线程打断。

Redis事务的注意点有哪些？

需要注意的点有：

- Redis 事务是不支持回滚的，不像 MySQL 的事务一样，要么都执行要么都不执行；
- Redis 服务端在执行事务的过程中，不会被其他客户端发送来的命令请求打断。直到事务命令全部执行完毕才会执行其他客户端的命令。

Redis 事务为什么不支持回滚？

Redis 的事务不支持回滚。

如果执行的命令有语法错误，Redis 会执行失败，这些问题可以从程序层面捕获并解决。但是如果出现其他问题，则依然会继续执行余下的命令。

这样做的原因是因为回滚需要增加很多工作，而不支持回滚则可以**保持简单、快速的特性**。

43.Redis和Lua脚本的使用了解吗？

Redis的事务功能比较简单，平时的开发中，可以利用Lua脚本来增强Redis的命令。

Lua脚本能给开发人员带来这些好处：

- Lua脚本在Redis中是原子执行的，执行过程中间不会插入其他命令。
- Lua脚本可以帮助开发和运维人员创造出自己定制的命令，并可以将这些命令常驻在Redis内存中，实现复用的效果。
- Lua脚本可以将多条命令一次性打包，有效地减少网络开销。

比如这一段很（烂）经（大）典（街）的秒杀系统利用lua扣减Redis库存的脚本：

```
1  -- 库存未预热
2  if (redis.call('exists', KEYS[2]) == 1) then
3      return -9;
4  end;
5  -- 秒杀商品库存存在
6  if (redis.call('exists', KEYS[1]) == 1) then
7      local stock = tonumber(redis.call('get', KEYS[1]));
8      local num = tonumber(ARGV[1]);
9      -- 剩余库存少于请求数量
10     if (stock < num) then
11         return -3
12     end;
13     -- 扣减库存
14     if (stock >= num) then
15         redis.call('incrby', KEYS[1], 0 - num);
16         -- 扣减成功
17         return 1
18     end;
19     return -2;
```

```
20 | end;  
21 | -- 秒杀商品库存不存在  
22 | return -1;
```

44.Redis的管道了解吗？

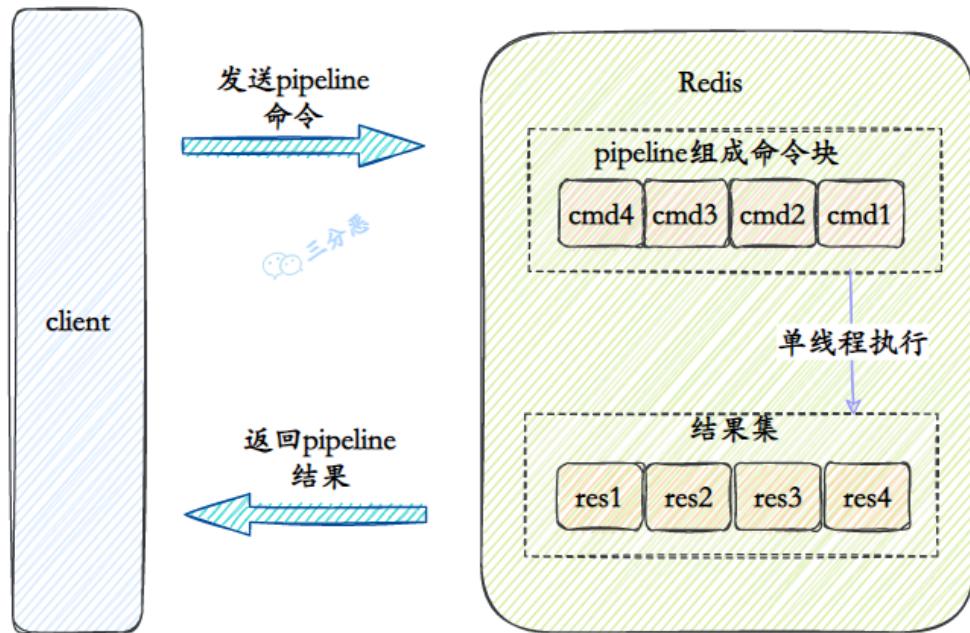
Redis 提供三种将客户端多条命令打包发送给服务端执行的方式：

Pipelining(管道)、 Transactions(事务) 和 Lua Scripts(Lua 脚本)。

Pipelining (管道)

Redis 管道是三者之中最简单的，当客户端需要执行多条 redis 命令时，可以通过管道一次性将要执行的多条命令发送给服务端，其作用是为了降低 RTT(Round Trip Time) 对性能的影响，比如我们使用 nc 命令将两条指令发送给 redis 服务端。

Redis 服务端接收到管道发送过来的多条命令后，会一直执命令，并将命令的执行结果进行缓存，直到最后一条命令执行完成，再所有命令的执行结果一次性返回给客户端。



Pipelining的优势

在性能方面，Pipelining 有下面两个优势：

- 节省了RTT：将多条命令打包一次性发送给服务端，减少了客户端与服务端之间的网络调用次数

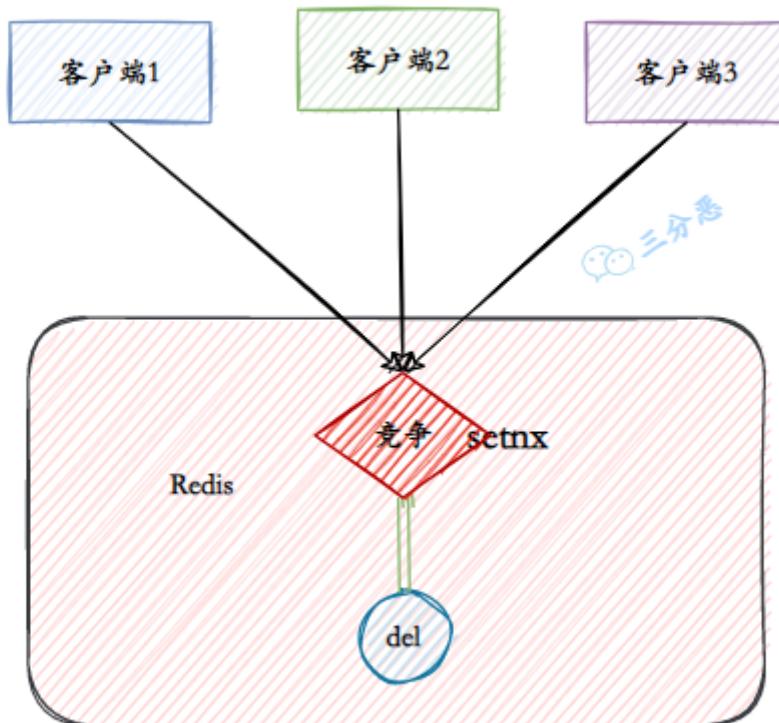
- 减少了上下文切换：当客户端/服务端需要从网络中读写数据时，都会产生一次系统调用，系统调用是非常耗时的操作，其中涉及到程序由用户态切换到内核态，再从内核态切换回用户态的过程。当我们执行 10 条 redis 命令的时候，就会发生 10 次用户态到内核态的上下文切换，但如果我们使用 Pipelining 将多条命令打包成一条一次性发送给服务端，就只会产生一次上下文切换。

45.Redis实现分布式锁了解吗？

Redis 是分布式锁本质上要实现的目标就是在 Redis 里面占一个“茅坑”，当别的进程也要来占时，发现已经有人蹲在那里了，就只好放弃或者稍后再试。

- V1: setnx 命令

占坑一般是使用 setnx(set if not exists) 指令，只允许被一个客户端占坑。先来先占，用完了，再调用 del 指令释放茅坑。



```

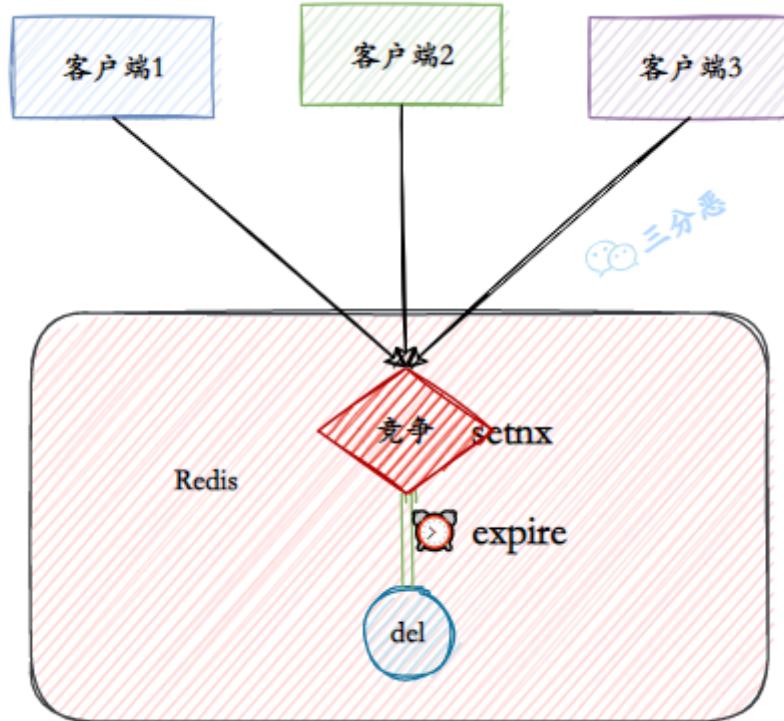
1 > setnx lock:fighter true
2 OK
3 ... do something critical ...
4 > del lock:fighter
5 (integer) 1

```

但是有个问题，如果逻辑执行到中间出现异常了，可能会导致 del 指令没有被调用，这样就会陷入死锁，锁永远得不到释放。

- V2: 锁超时释放

所以在拿到锁之后，再给锁加上一个过期时间，比如 5s，这样即使中间出现异常也可以保证 5 秒之后锁会自动释放。



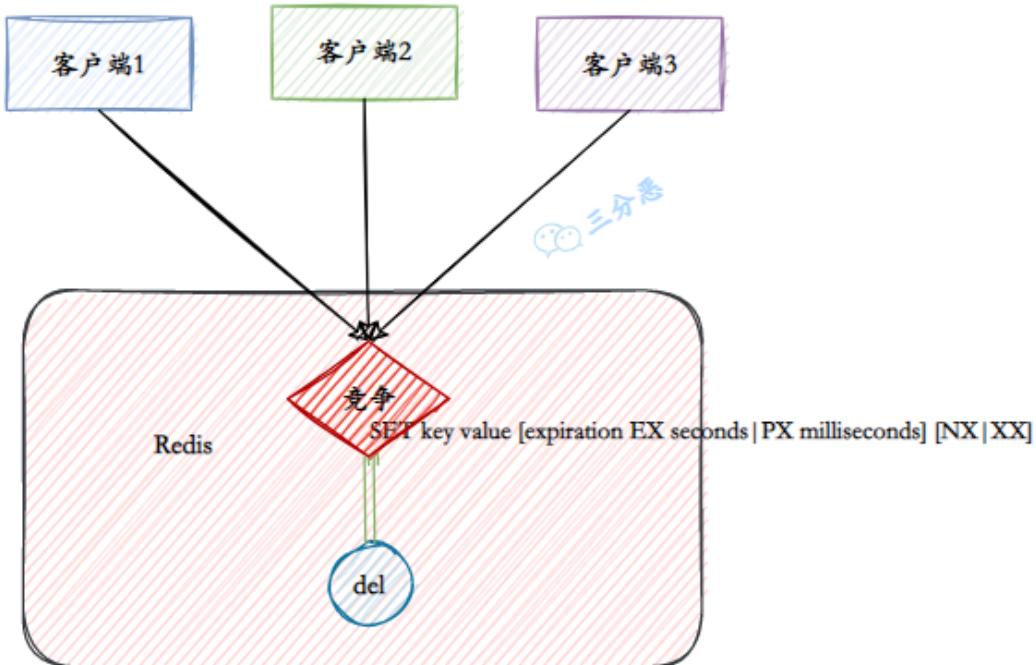
```
1 > setnx lock:fighter true
2 OK
3 > expire lock:fighter 5
4 ... do something critical ...
5 > del lock:fighter
6 (integer) 1
```

但是以上逻辑还有问题。如果在 setnx 和 expire 之间服务器进程突然挂掉了，可能是因为机器掉电或者是被人为杀掉的，就会导致 expire 得不到执行，也会造成死锁。

这种问题的根源就在于 setnx 和 expire 是两条指令而不是原子指令。如果这两条指令可以一起执行就不会出现问题。

- V3:set指令

这个问题在Redis 2.8 版本中得到了解决，这个版本加入了 set 指令的扩展参数，使得 setnx 和 expire 指令可以一起执行。



```
1 | set lock:fighter3 true ex 5 nx OK ... do something critical  
... > del lock:codehole
```

上面这个指令就是 setnx 和 expire 组合在一起的原子指令，这个就算是比较完善的分布式锁了。

当然实际的开发，没人会去自己写分布式锁的命令，因为有专业的轮子——**Redisson**。

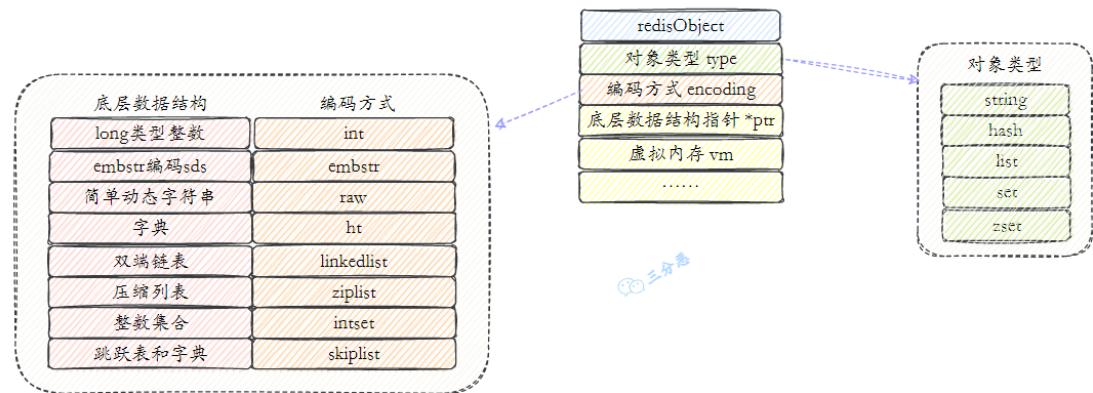
底层结构

这一部分就比较深了，如果不是简历上写了精通Redis，应该不会怎么问。

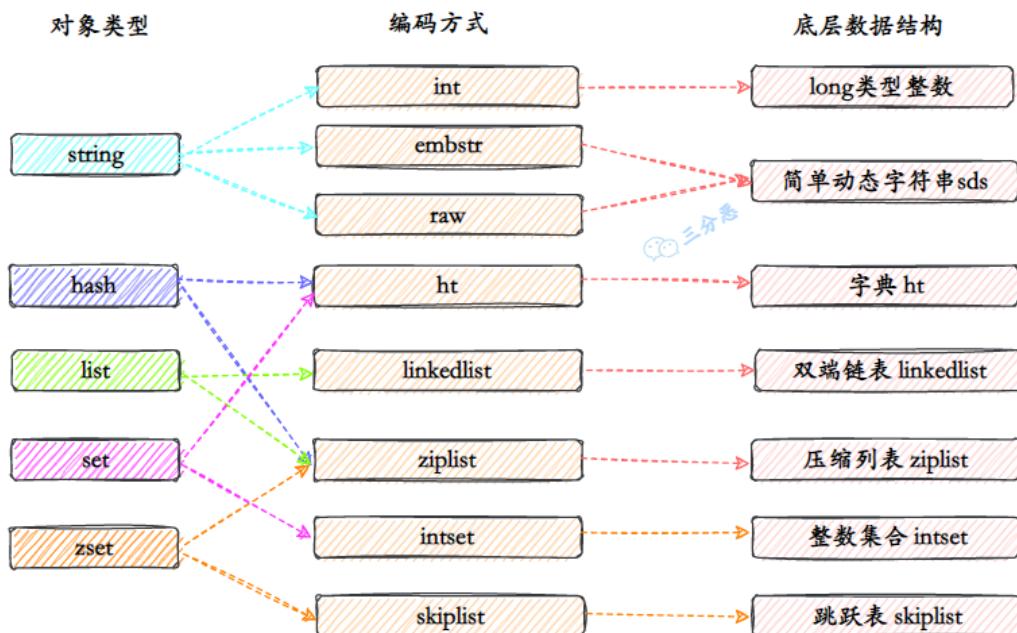
46. 说说Redis底层数据结构？

Redis有**动态字符串(sds)**、**链表(list)**、**字典(ht)**、**跳跃表(skiplist)**、**整数集合(intset)**、**压缩列表(ziplist)** 等底层数据结构。

Redis并没有使用这些数据结构来直接实现键值对数据库，而是基于这些数据结构创建了一个对象系统，来表示所有的key-value。



我们常用的数据类型和编码对应的映射关系:



简单看一下底层数据结构，如果对数据结构掌握不错的话，理解这些结构应该不是特别难：

1. **字符串**: redis没有直接使用C语言传统的字符串表示，而是自己实现的叫做简单动态字符串SDS的抽象类型。

C语言的字符串不记录自身的长度信息，而SDS则保存了长度信息，这样将获取字符串长度的时间由O(N)降低到了O(1)，同时可以避免缓冲区溢出和减少修改字符串长度时所需的内存重分配次数。

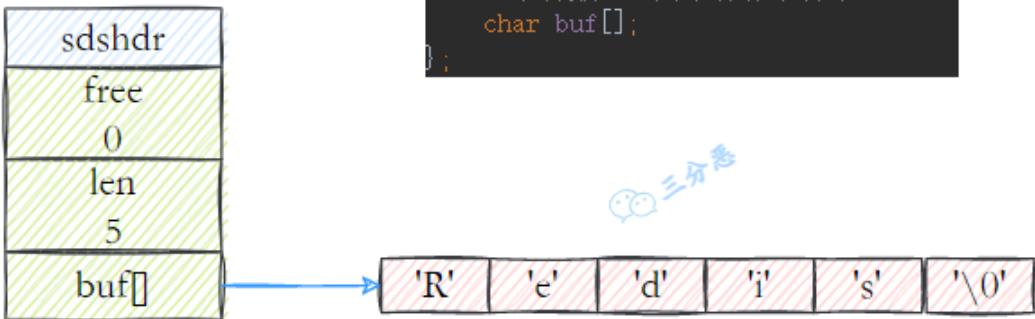
```

struct sdshdr {
    //记录buf数组中已经使用字节的数量
    //等于SDS所保存字符串的长度
    unsigned int len;

    //记录buf数组中未使用字节的数量
    unsigned int free;

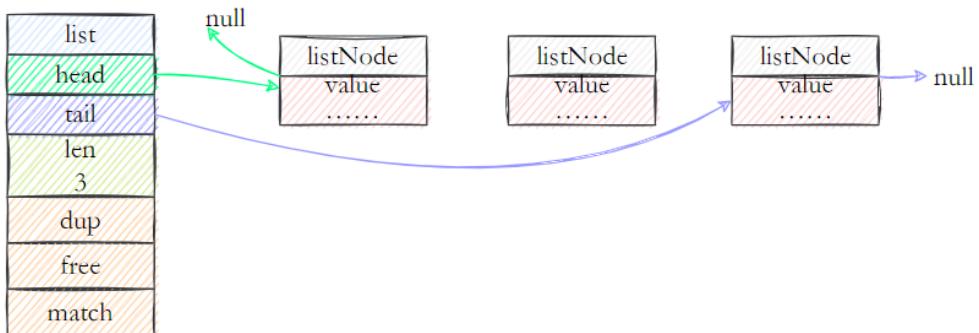
    //字符数组，用于保存字符串
    char buf[];
};

```



2. 链表linkedlist：redis链表是一个双向无环链表结构，很多发布订阅、慢查询、监视器功能都是使用到了链表来实现，每个链表的节点由一个listNode结构来表示，每个节点都有指向前置节点和后置节点的指针，同时表头节点的前置和后置节点都指向NULL。

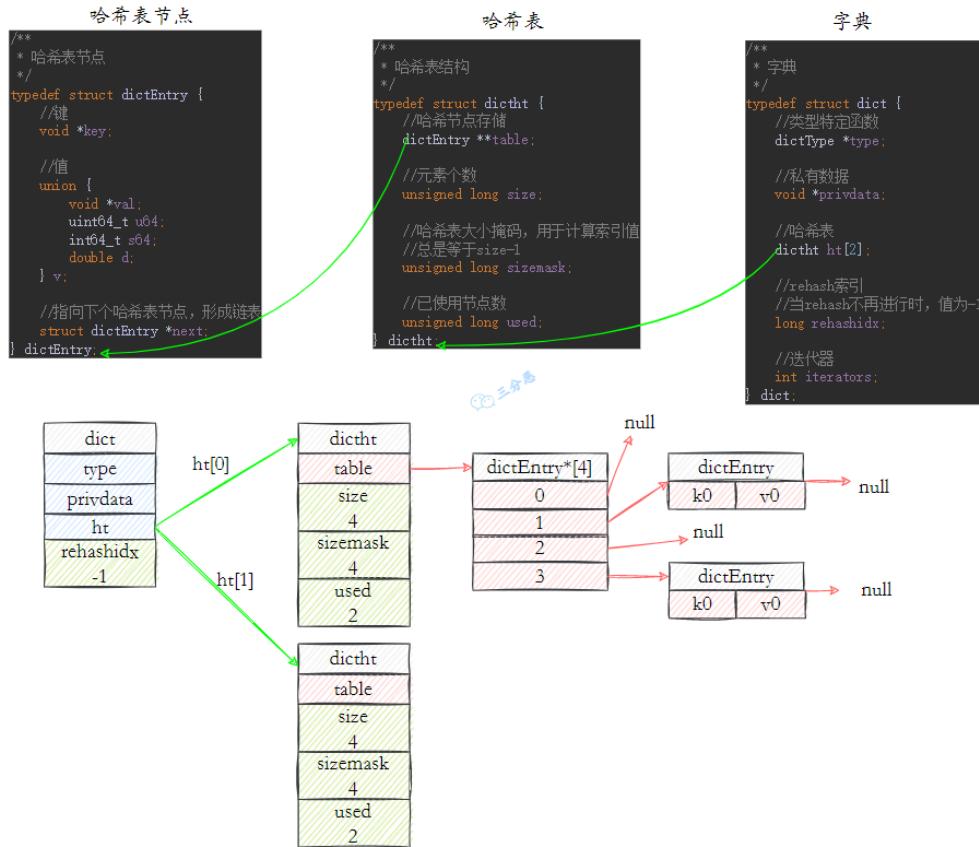
链表节点	链表类型
<pre> typedef struct listNode { //前驱节点 struct listNode *prev; //后继节点 struct listNode *next; //节点的值 void *value; } listNode; </pre>	<pre> typedef struct list { //链表头节点 listNode *head; //链表尾节点 listNode *tail; //节点值复制函数 void *(*dup)(void *ptr); //节点值释放函数 void (*free)(void *ptr); //节点值比对函数 int (*match)(void *ptr, void *key); //链表包含的节点数量 unsigned long len; } list; </pre>



3. 字典dict：用于保存键值对的抽象数据结构。Redis使用hash表作为底层实现，一个哈希表里可以有多个哈希表节点，而每个哈希表节点就保存了字典里中的一个

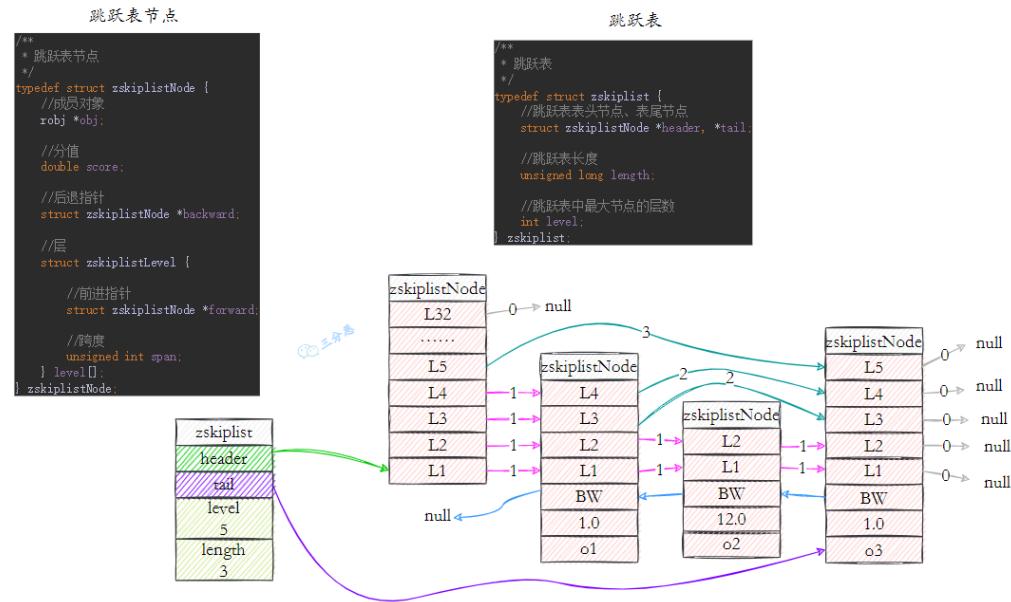
键值对。

每个字典带有两个hash表，供平时使用和rehash时使用，hash表使用链地址法来解决键冲突，被分配到同一个索引位置的多个键值对会形成一个单向链表，在对hash表进行扩容或者缩容的时候，为了服务的可用性，rehash的过程不是一次性完成的，而是渐进式的。

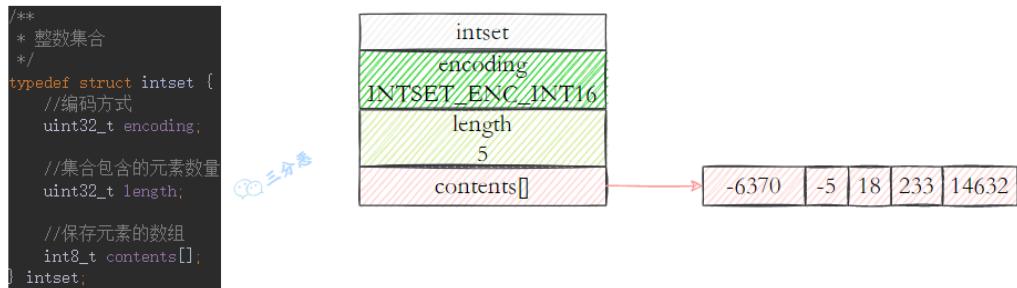


4. 跳跃表skiplist：跳跃表是有序集合的底层实现之一，Redis中在实现有序集合键和集群节点的内部结构中都是用到了跳跃表。Redis跳跃表由zskiplist和zskiplistNode组成，zskiplist用于保存跳跃表信息（表头、表尾节点、长度等），zskiplistNode用于表示表跳跃节点，每个跳跃表节点的层高都是1-32的随机数，在同一个跳跃表中，多个节点可以包含相同的分值，但是每个节点的成员对象必须是唯一的，节点按照分值大小排序，如果分值相同，则按照成员对象的大小排

序。



5. 整数集合intset：用于保存整数值的集合抽象数据结构，不会出现重复元素，底层实现为数组。



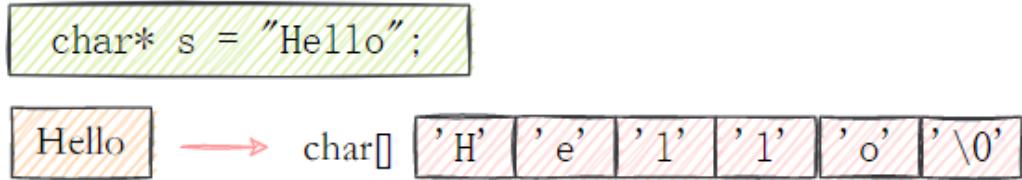
6. 压缩列表ziplist：压缩列表是为节约内存而开发的顺序性数据结构，它可以包含任意多个节点，每个节点可以保存一个字节数组或者整数值。

zlbytes	zltail	zllen	entry1	entry2	entryN	zlen
---------	--------	-------	--------	--------	-------	--------	------

47.Redis 的 SDS 和 C 中字符串相比有什么优势？

C 语言使用了一个长度为 **N+1** 的字符数组来表示长度为 **N** 的字符串，并且字符数组最后一个元素总是 **\0**，这种简单的字符串表示方式不符合 Redis 对字符串在安全性、效率以及功能方面的要求。

C语言字符串

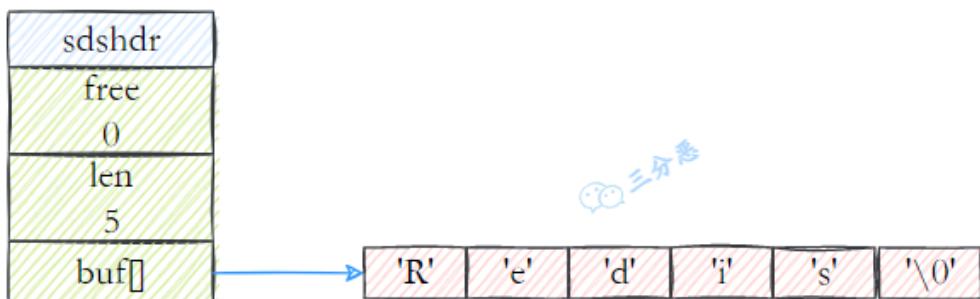


C语言的字符串可能有什么问题？

这样简单的数据结构可能会造成以下一些问题：

- 获取字符串长度复杂度高：因为 C 不保存数组的长度，每次都需要遍历一遍整个数组，时间复杂度为 $O(n)$ ；
- 不能杜绝 缓冲区溢出/内存泄漏 的问题：C字符串不记录自身长度带来的另外一个问题就是容易造成缓存区溢出（buffer overflow），例如在字符串拼接的时候，新的
- C 字符串 只能保存文本数据 → 因为 C 语言中的字符串必须符合某种编码（比如 ASCII），例如中间出现的 '\0' 可能会被判定为提前结束的字符串而识别不了；

Redis如何解决？优势？



简单来说一下 Redis 如何解决的：

1. 多增加 len 表示当前字符串的长度：这样就可以直接获取长度了，复杂度 $O(1)$ ；
2. 自动扩展空间：当 SDS 需要对字符串进行修改时，首先借助于 len 和 alloc 检查空间是否满足修改所需的要求，如果空间不够的话，SDS 会自动扩展空间，避免了像 C 字符串操作中的溢出情况；

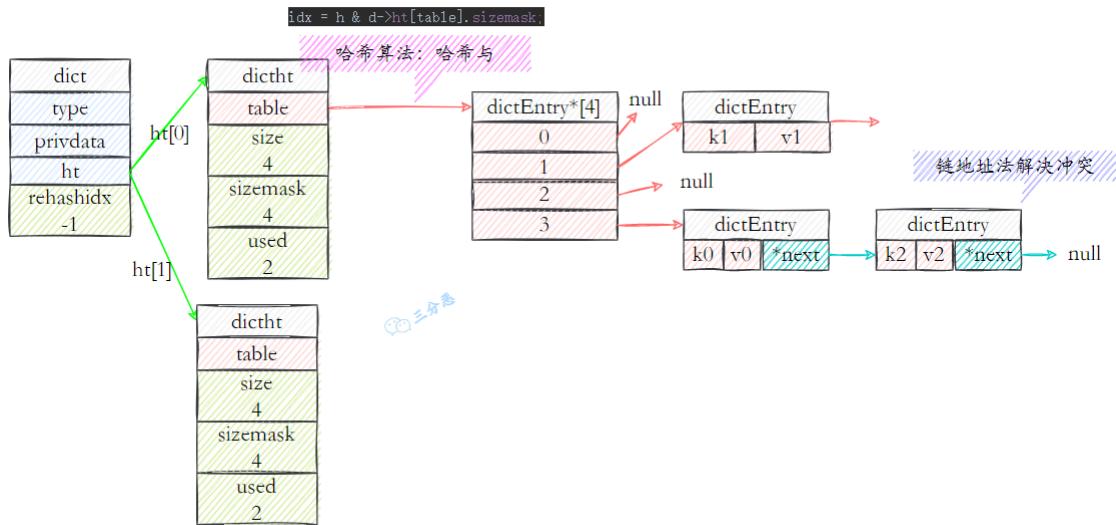
3. 有效降低内存分配次数：C 字符串在涉及增加或者清除操作时会改变底层数组的大小造成重新分配，SDS 使用了 空间预分配 和 惰性空间释放 机制，简单理解就是每次在扩展时是成倍的多分配的，在缩容时也是先留着并不正式归还给 OS；
4. 二进制安全：C 语言字符串只能保存 `ascii` 码，对于图片、音频等信息无法保存，SDS 是二进制安全的，写入什么读取就是什么，不做任何过滤和限制；

48. 字典是如何实现的？Rehash 了解吗？

字典是 Redis 服务器中出现最为频繁的复合型数据结构。除了 `hash` 结构的数据会用到字典外，整个 Redis 数据库的所有 `key` 和 `value` 也组成了一个 **全局字典**，还有带过期时间的 `key` 也是一个字典。**(存储在 RedisDb 数据结构中)**

字典结构是什么样的呢？

Redis 中的字典相当于 Java 中的 `HashMap`，内部实现也差不多类似，采用哈希与运算计算下标位置；通过 “**数组 + 链表**” 的 **链地址法** 来解决哈希冲突，同时这样的结构也吸收了两种不同数据结构的优点。



字典是怎么扩容的？

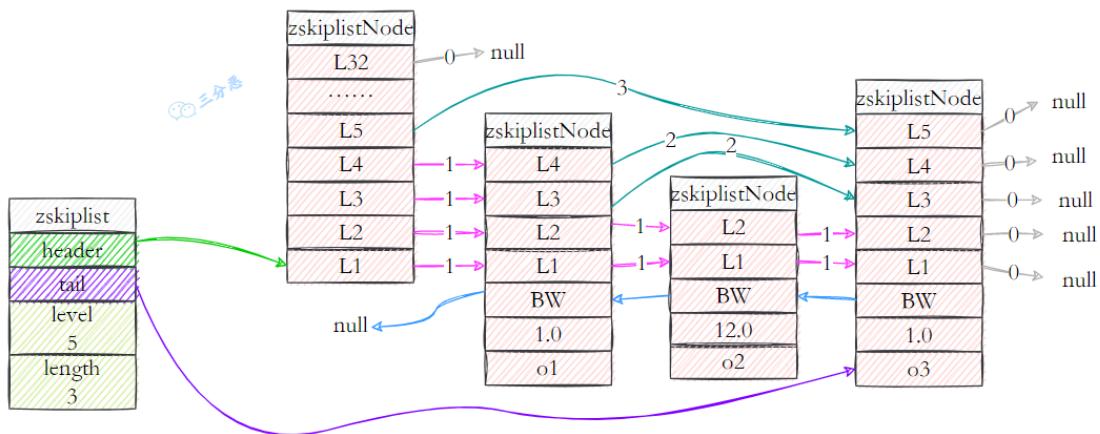
字典结构内部包含 **两个 hashtable**，通常情况下只有一个哈希表 `ht[0]` 有值，在扩容的时候，把 `ht[0]` 里的值 `rehash` 到 `ht[1]`，然后进行 **渐进式 rehash** —— 所谓渐进式 `rehash`，指的是这个 `rehash` 的动作并不是一次性、集中式地完成的，而是分多次、渐进式地完成的。

待搬迁结束后，`ht[1]` 就取代 `ht[0]` 存储字典的元素。

49. 跳跃表是如何实现的？原理？

PS:跳跃表是比较常问的一种结构。

跳跃表（skiplist）是一种有序数据结构，它通过在每个节点中维持多个指向其它节点的指针，从而达到快速访问节点的目的。



为什么使用跳跃表？

首先，因为 zset 要支持随机的插入和删除，所以它 **不宜使用数组来实现**，关于排序问题，我们也很容易就想到 **红黑树/平衡树** 这样的树形结构，为什么 Redis 不使用这样一些结构呢？

1. 性能考虑： 在高并发的情况下，树形结构需要执行一些类似于 rebalance 这样的可能涉及整棵树的操作，相对来说跳跃表的变化只涉及局部；
2. 实现考虑： 在复杂度与红黑树相同的情况下，跳跃表实现起来更简单，看起来也更加直观；

基于以上的一些考虑，Redis 基于 **William Pugh** 的论文做出一些改进后采用了 **跳跃表** 这样的结构。

本质是解决查找问题。

跳跃表是怎么实现的？

跳跃表的节点里有这些元素：

- 层

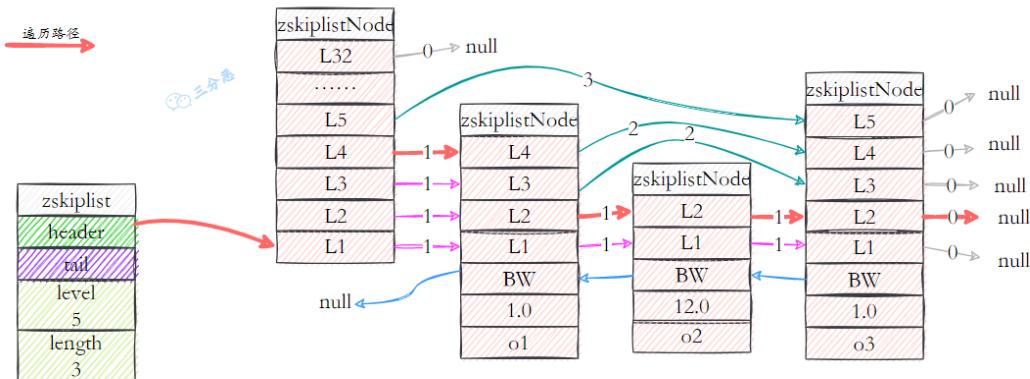
跳跃表节点的level数组可以包含多个元素，每个元素都包含一个指向其它节点的指针，程序可以通过这些层来加快访问其它节点的速度，一般来说，层的数量越多，访问其它节点的速度就越快。

每次创建一个新的跳跃表节点的时候，程序都根据幂次定律，随机生成一个介于1和32之间的值作为level数组的大小，这个大小就是层的“高度”

- **前进指针**

每个层都有一个指向表尾的前进指针（level[i].forward属性），用于从表头向表尾方向访问节点。

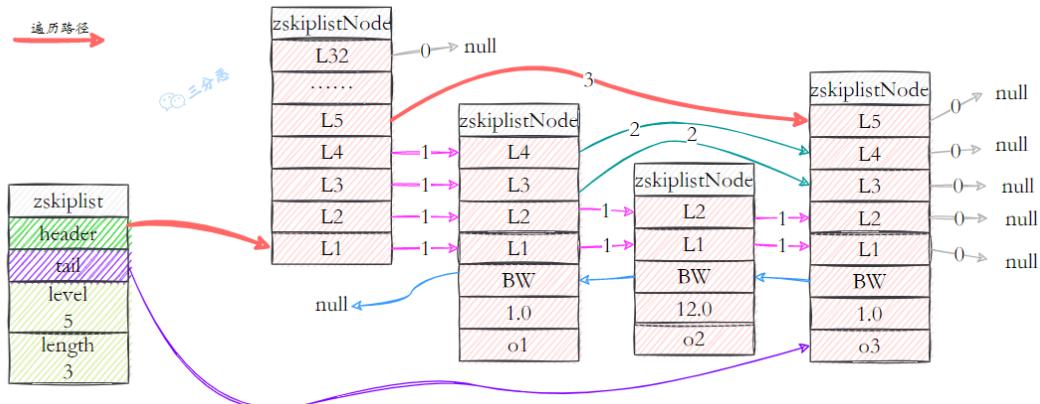
我们看一下跳跃表从表头到表尾，遍历所有节点的路径：



- **跨度**

层的跨度用于记录两个节点之间的距离。跨度是用来计算排位（rank）的：在查找某个节点的过程中，将沿途访问过的所有层的跨度累计起来，得到的结果就是目标节点在跳跃表中的排位。

例如查找，分值为3.0、成员对象为o3的节点时，沿途经历的层：查找的过程只经过了一个层，并且层的跨度为3，所以目标节点在跳跃表中的排位为3。



- **分值和成员**

节点的分值（score属性）是一个double类型的浮点数，跳跃表中所有的节点都按分值从小到大来排序。

节点的成员对象（obj属性）是一个指针，它指向一个字符串对象，而字符串对象则保存这一个SDS值。

50. 压缩列表了解吗？

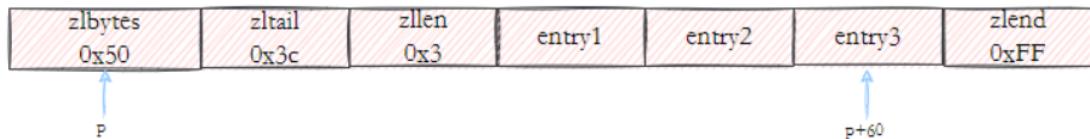
压缩列表是 Redis 为了节约内存 而使用的一种数据结构，是由一系列特殊编码的连续内存快组成的顺序型数据结构。

一个压缩列表可以包含任意多个节点（entry），每个节点可以保存一个字节数组或者一个整数值。



压缩列表由这么几部分组成：

- zbytes :记录整个压缩列表占用的内存字节数
- ztail :记录压缩列表表尾节点距离压缩列表的起始地址有多少字节
- zllen :记录压缩列表包含的节点数量
- entryX :列表节点
- zlend :用于标记压缩列表的末端



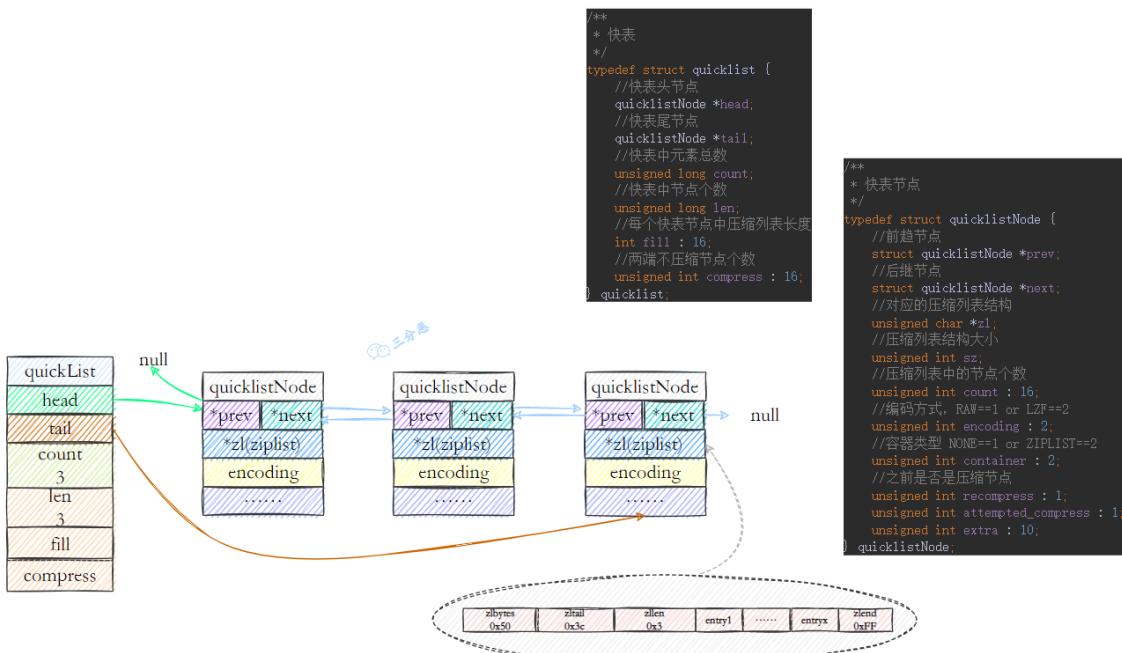
51. 快速列表 quicklist 了解吗？

Redis 早期版本存储 list 列表数据结构使用的是压缩列表 ziplist 和普通的双向链表 linkedlist，也就是说当元素少时使用 ziplist，当元素多时用 linkedlist。

但考虑到链表的附加空间相对较高，`prev` 和 `next` 指针就要占去 16 个字节（64 位操作系统占用 8 个字节），另外每个节点的内存都是单独分配，会家具内存的碎片化，影响内存管理效率。

后来 Redis 新版本（3.2）对列表数据结构进行了改造，使用 `quicklist` 代替了 `ziplist` 和 `linkedlist`，`quicklist` 是综合考虑了时间效率与空间效率引入的新型数据结构。

quicklist由list和ziplist结合而成，它是一个由ziplist充当节点的双向链表。



其他问题

52.假如Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如何将它们全部找出来？

使用 `keys` 指令可以扫出指定模式的 key 列表。但是要注意 `keys` 指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用 `scan` 指令，`scan` 指令可以无阻塞的提取出指定模式的 `key` 列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用 `keys` 指令长。

参考：

- [1]. 《Redis开发与实战》
- [2]. 《Redis设计与实现》
- [3]. 《Redis深度历险》
- [4]. 艾小仙《我要进大厂》
- [5]. 田维常《后端面试小笔记》
- [6]. [美团二面：Redis与MySQL双写一致性如何保证？](#)
- [7]. [妈妈再也不担心我面试被Redis问得脸都绿了](#)

- [8]. [面试官：缓存一致性问题怎么解决？](#)
 - [9]. [高并发场景下，到底先更新缓存还是先更新数据库？](#)
 - [10]. [【Redis破障之路】三：Redis单线程架构](#)
 - [11]. Redis官网
 - [12]. [解决了Redis大key问题，同事们都夸他牛皮](#)
 - [13]. [Redis 分布式锁原理看这篇就够了，循循渐进](#)
 - [14]. [《Redis5设计与源码分析》](#)
-

关注公众号：三分恶

手册更新动态
即刻送达



添加个人微信：ThirdFighter

技术交流
加大佬云集微信群



第五部分：中间件

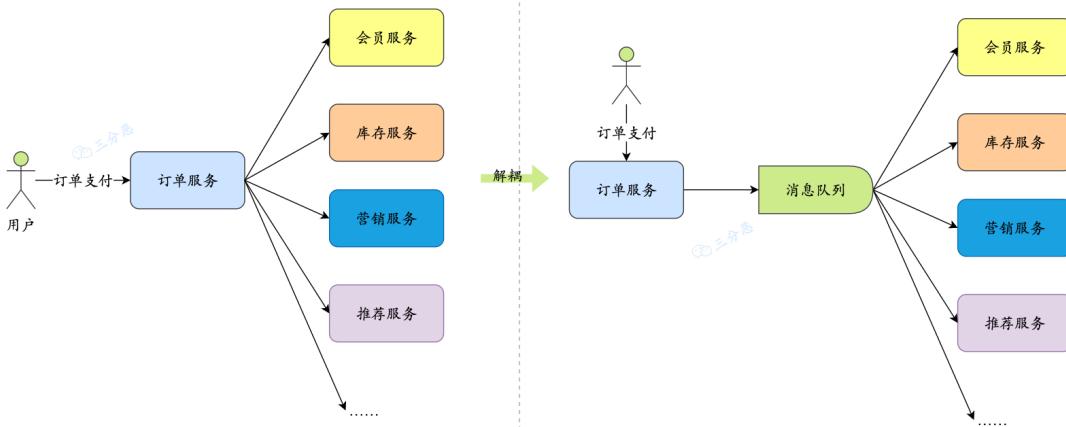
一、RocketMQ

| 基础

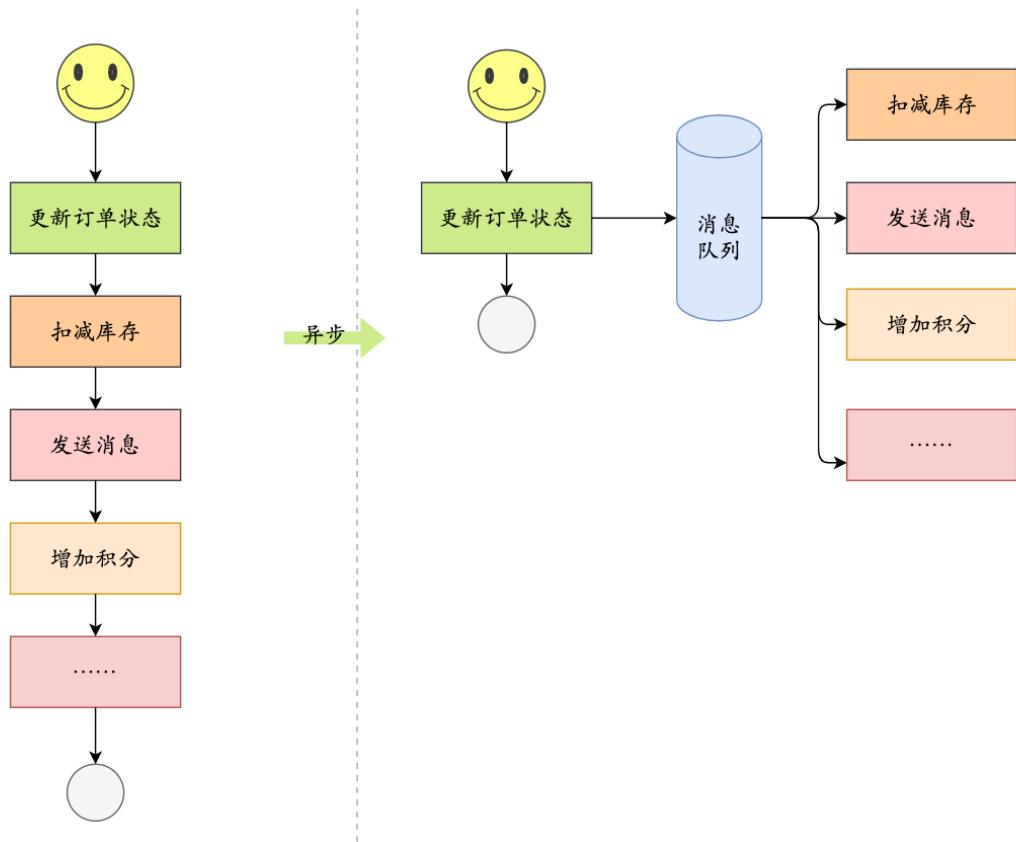
1.为什么要使用消息队列呢？

消息队列主要有三大用途，我们拿一个电商系统的下单举例：

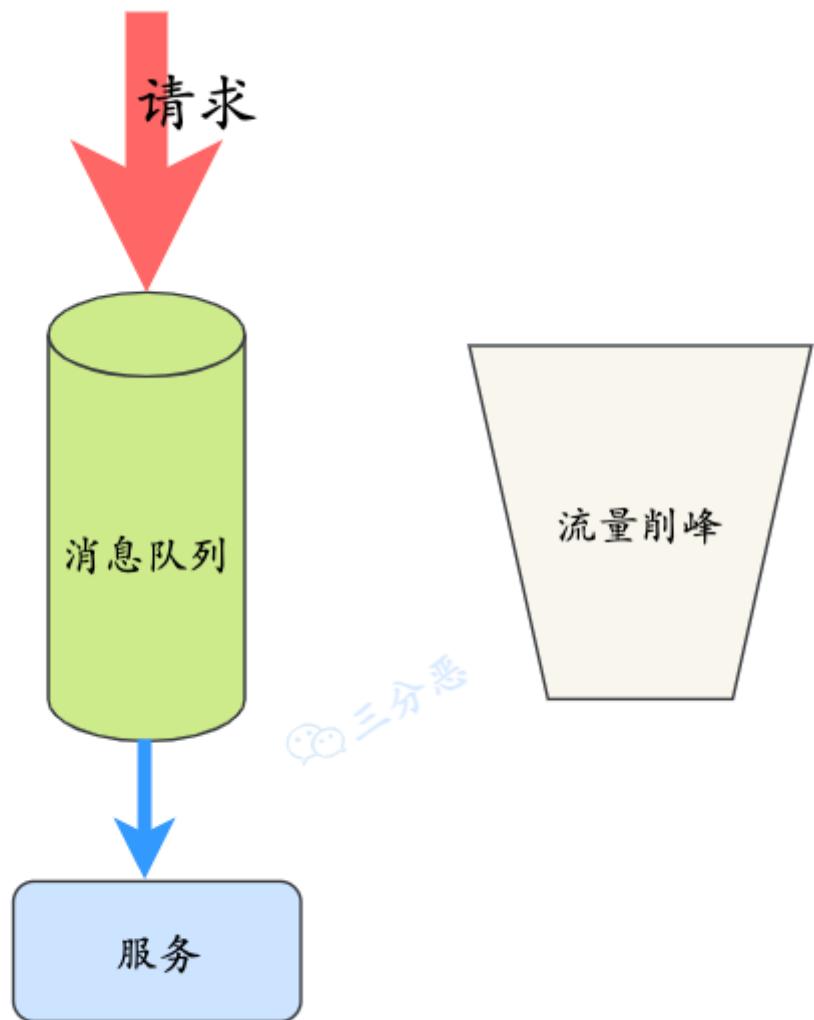
- **解耦**: 引入消息队列之前，下单完成之后，需要订单服务去调用库存服务减库存，调用营销服务加营销数据……引入消息队列之后，可以把订单完成的消息丢进队列里，下游服务自己去调用就行了，这样就完成了订单服务和其它服务的解耦合。



- **异步**: 订单支付之后，我们要扣减库存、增加积分、发送消息等等，这样一来这个链路就长了，链路一长，响应时间就变长了。引入消息队列，除了 **更新订单状态**，其它的都可以**异步**去做，这样一来就来，就能降低响应时间。



- **削峰**: 消息队列合一用来削峰，例如秒杀系统，平时流量很低，但是要做秒杀活动，秒杀的时候流量疯狂怼进来，我们的服务器，Redis，MySQL各自的承受能力都不一样，直接全部流量照单全收肯定有问题啊，严重点可能直接打挂了。我们可以把请求扔到队列里面，只放出我们服务能处理的流量，这样就能抗住短时间的大流量了。



解耦、异步、削峰，是消息队列最主要的三大作用。

2.为什么要选择RocketMQ?

市场上几大消息队列对比如下：

【三分恶】原创	RabbitMQ	ActiveMQ	RocketMQ	Kafka
公司/社区	Rabbit	Apache	阿里	Apache
语言	Erlang	Java	Java	Scala&Java
协议支持	AMQP	OpenWire、 STOMP、 REST、XMPP、 AMQP	自定义	自定义协议，社区封装了http协议支持
客户端支持语言	官方支持Erlang、 Java、Ruby等，社区查出多种API，几乎支持所有语言	Java、C、 C++、 Python、 PHP、 Perl、.net 等	Java C++ (不成熟)	官方支持Java，社区产出多种API，如PHP，Python等
单机吞吐量	万级 ③	万级 ④	十万级 ①	十万级 ②
消息延迟	微秒级	毫秒级	毫秒级	毫秒以内
可用性	高，基于主从架构实现可用性	高，基于主从架构实现可用性	非常高，分布式架构	非常高，分布式架构，一个数据多副本
消息可靠性	-	有较低的概率丢失数据	经过参数优化配置，可以做到零丢失	经过参数配置，消息可以做到零丢失
功能支持	基于erlang开发，所以并发性能极强，性能极好，延时低	MQ领域的功能极其完备	MQ功能较为完备，分布式扩展性好	功能较为简单，主要支持加单MQ功能
优势	erlang语言开发，性能极好、延时很低，吞吐量万级、MQ功能完备，管理界面非常好，社区活跃；互联网公司使用较多	非常成熟，功能强大，在业内大量公司和项目中都有应用	接口简单易用，阿里出品有保障，吞吐量大，分布式扩展方便、社区比较活跃，支持大规模的Topic、支持复杂的业务场景，可以基于源码进行定制开发	超高吞吐量，ms级的时延，极高的可用性和可靠性，分布式扩展方便
劣势	吞吐量较低，erlang语音开发不容易进行定制开发，集群动态扩展麻烦	偶尔有较低概率丢失消息，社区活跃度不高	接口不是按照标准JMS规范走的，有的系统迁移要修改大量的代码，技术有被抛弃的风险	有可能进行消息的重复消费

应用	都有使用	主要用于解耦和异步，较少用在大规模吞吐的场景中	用于大规模吞吐、复杂业务中	在大数据的实时计算和日志采集中被大规模使用，是业界的标准
----	------	-------------------------	---------------	------------------------------

总结一下：

选择中间件的可以从这些维度来考虑：可靠性，性能，功能，可运维性，可拓展性，社区活跃度。目前常用的几个中间件，ActiveMQ作为“老古董”，市面上用的已经不多，其它几种：

- RabbitMQ:
 - 优点：轻量，迅捷，容易部署和使用，拥有灵活的路由配置
 - 缺点：性能和吞吐量不太理想，不易进行二次开发
- RocketMQ:
 - 优点：性能好，高吞吐量，稳定可靠，有活跃的中文社区
 - 缺点：兼容性上不是太好
- Kafka:
 - 优点：拥有强大的性能及吞吐量，兼容性很好
 - 缺点：由于“攒一波再处理”导致延迟比较高

我们的系统是面向用户的C端系统，具有一定的并发量，对性能也有比较高的要求，所以选择了低延迟、吞吐量比较高，可用性比较好的RocketMQ。

3.RocketMQ有什么优缺点？

RocketMQ优点：

- 单机吞吐量：十万级
- 可用性：非常高，分布式架构
- 消息可靠性：经过参数优化配置，消息可以做到0丢失
- 功能支持：MQ功能较为完善，还是分布式的，扩展性好
- 支持10亿级别的消息堆积，不会因为堆积导致性能下降
- 源码是Java，方便结合公司自己的业务二次开发
- 天生为金融互联网领域而生，对于可靠性要求很高的场景，尤其是电商里面的订单扣款，以及业务削峰，在大量交易涌入时，后端可能无法及时处理的情况

- RocketMQ 在稳定性上可能更值得信赖，这些业务场景在阿里双11已经经历了多次考验，如果你的业务有上述并发场景，建议可以选择 RocketMQ

RocketMQ缺点：

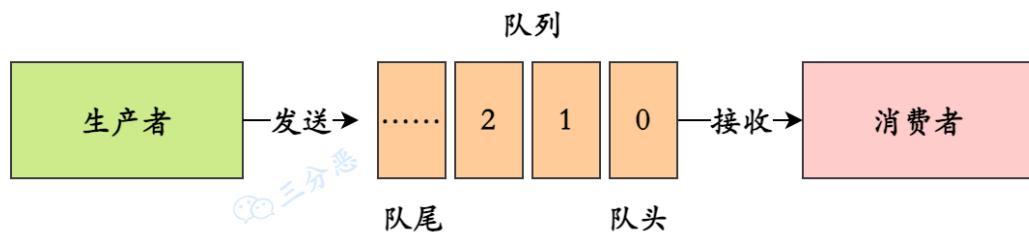
- 支持的客户端语言不多，目前是Java及c++，其中c++不成熟
- 没有在 MQ核心中去实现 JMS 等接口，有些系统要迁移需要修改大量代码

4.消息队列有哪些消息模型？

消息队列有两种模型： **队列模型** 和 **发布/订阅模型**。

- **队列模型**

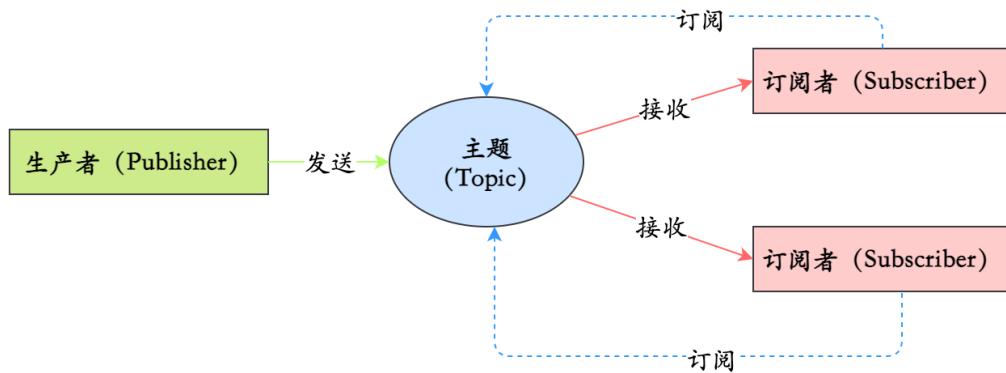
这是最初的一种消息队列模型，对应着消息队列“发-存-收”的模型。生产者往某个队列里面发送消息，一个队列可以存储多个生产者的消息，一个队列也可以有多个消费者，但是消费者之间是竞争关系，也就是说每条消息只能被一个消费者消费。



- **发布/订阅模型**

如果需要将一份消息数据分发给多个消费者，并且每个消费者都要求收到全量的消息。很显然，队列模型无法满足这个需求。解决的方式就是发布/订阅模型。

在发布 - 订阅模型中，消息的发送方称为发布者（Publisher），消息的接收方称为订阅者（Subscriber），服务端存放消息的容器称为主题（Topic）。发布者将消息发送到主题中，订阅者在接收消息之前需要先“订阅主题”。“订阅”在这里既是一个动作，同时还可以认为是主题在消费时的一个逻辑副本，每份订阅中，订阅者都可以接收到主题的所有消息。

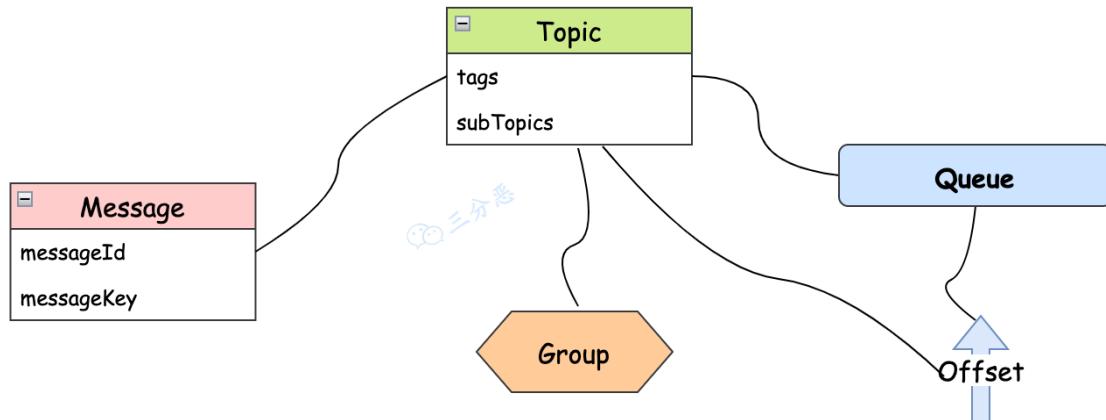


它和“队列模式”的异同：生产者就是发布者，队列就是主题，消费者就是订阅者，无本质区别。唯一的不同点在于：一份消息数据是否可以被多次消费。

5. 那RocketMQ的消息模型呢？

RocketMQ使用的消息模型是标准的发布-订阅模型，在RocketMQ的术语表中，生产者、消费者和主题，与发布-订阅模型中的概念是完全一样的。

RocketMQ本身的消息是由下面几部分组成：



- Message

Message（消息）就是要传输的信息。

一条消息必须有一个主题（Topic），主题可以看做是你的信件要邮寄的地址。

一条消息也可以拥有一个可选的标签（Tag）和额外的键值对，它们可以用于设置一个业务 Key 并在 Broker 上查找此消息以便在开发期间查找问题。

- Topic

Topic（主题）可以看做消息的归类，它是消息的第一级类型。比如一个电商系统可以分为：交易消息、物流消息等，一条消息必须有一个 Topic。

Topic 与生产者和消费者的关系非常松散，一个 Topic 可以有0个、1个、多个生产者向其发送消息，一个生产者也可以同时向不同的 Topic 发送消息。

一个 Topic 也可以被 0个、1个、多个消费者订阅。

- Tag

Tag（标签）可以看作子主题，它是消息的第二级类型，用于为用户提供额外的灵活性。使用标签，同一业务模块不同目的的消息就可以用相同 Topic 而不同的 **Tag** 来标识。比如交易消息又可以分为：交易创建消息、交易完成消息等，一条消息可以没有 **Tag**。

标签有助于保持你的代码干净和连贯，并且还可以为 **RocketMQ** 提供的查询系统提供帮助。

- Group

RocketMQ中，订阅者的概念是通过消费组（Consumer Group）来体现的。每个消费组都消费主题中一份完整的消息，不同消费组之间消费进度彼此不受影响，也就是说，一条消息被Consumer Group1消费过，也会再给Consumer Group2消费。

消费组中包含多个消费者，同一个组内的消费者是竞争消费的关系，每个消费者负责消费组内的一部分消息。默认情况，如果一条消息被消费者Consumer1消费了，那同组的其他消费者就不会再收到这条消息。

- Message Queue

Message Queue（消息队列），一个 Topic 下可以设置多个消息队列，Topic 包括多个 Message Queue，如果一个 Consumer 需要获取 Topic下所有的消息，就要遍历所有的 Message Queue。

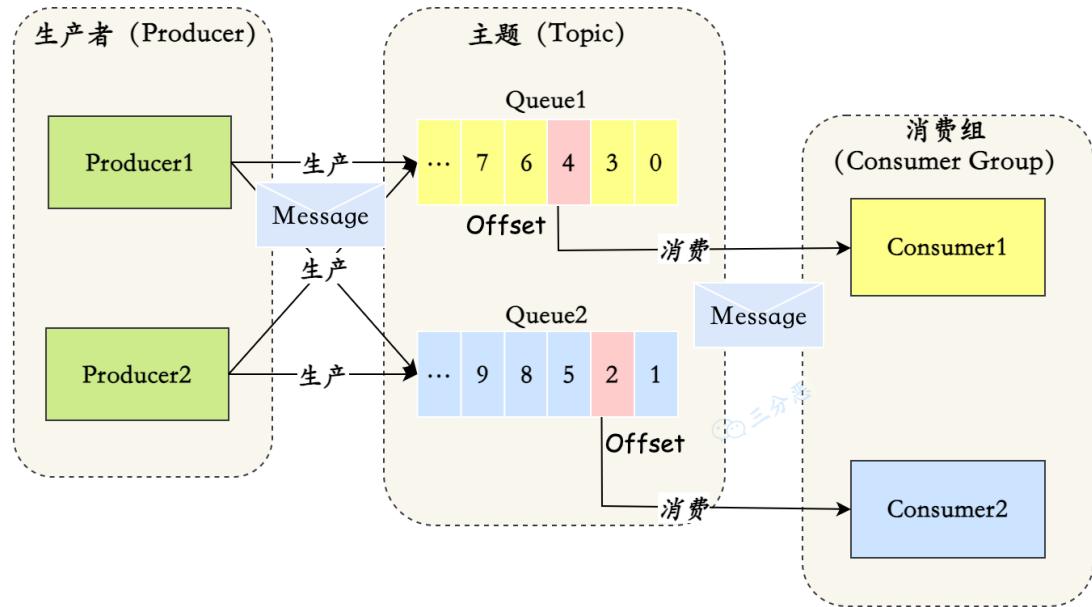
RocketMQ还有一些其它的Queue——例如ConsumerQueue。

- Offset

在Topic的消费过程中，由于消息需要被不同的组进行多次消费，所以消费完的消息并不会立即被删除，这就需要RocketMQ为每个消费组在每个队列上维护一个消费位置（Consumer Offset），这个位置之前的消息都被消费过，之后的消息都没有被消费过，每成功消费一条消息，消费位置就加一。

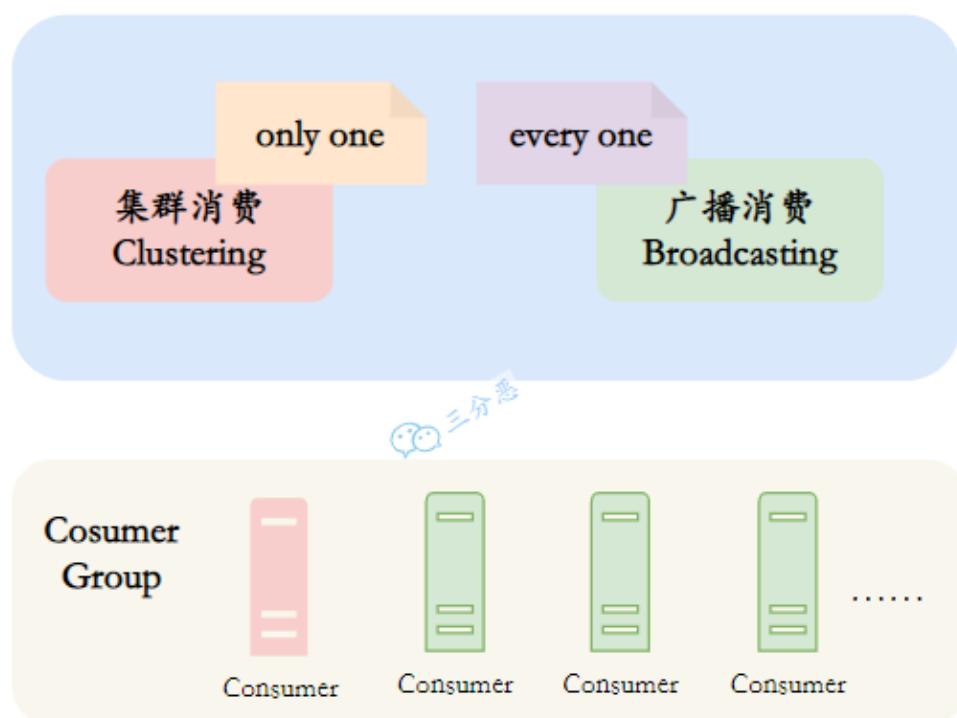
也可以这么说，**Queue** 是一个长度无限的数组，**Offset** 就是下标。

RocketMQ的消息模型中，这些就是比较关键的概念了。画张图总结一下：



6.消息的消费模式了解吗？

消息消费模式有两种：**Clustering**（集群消费）和**Broadcasting**（广播消费）。

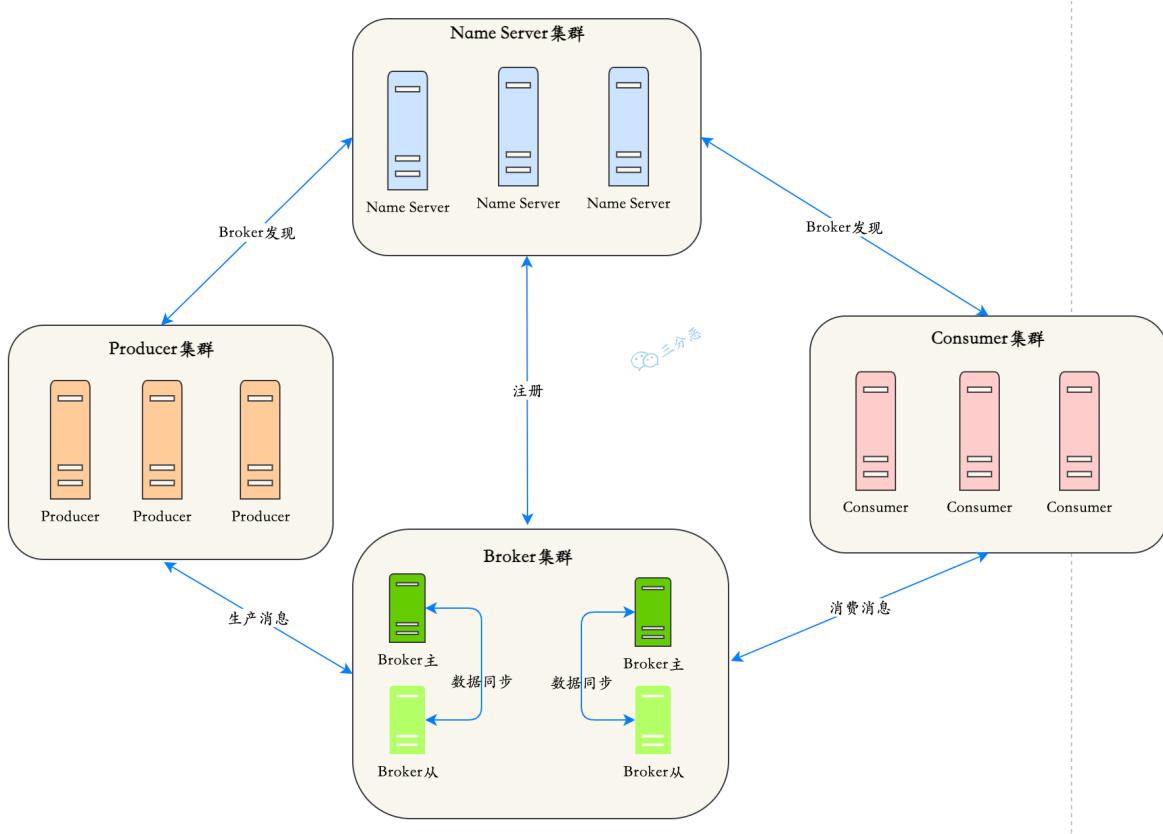


默认情况下就是集群消费，这种模式下 **一个消费者组共同消费一个主题的多个队列，一个队列只会被一个消费者消费**，如果某个消费者挂掉，分组内其它消费者会接替挂掉的消费者继续消费。

而广播消费消息会发给消费者组中的每一个消费者进行消费。

7.RocketMQ基本架构了解吗？

先看图，RocketMQ的基本架构：

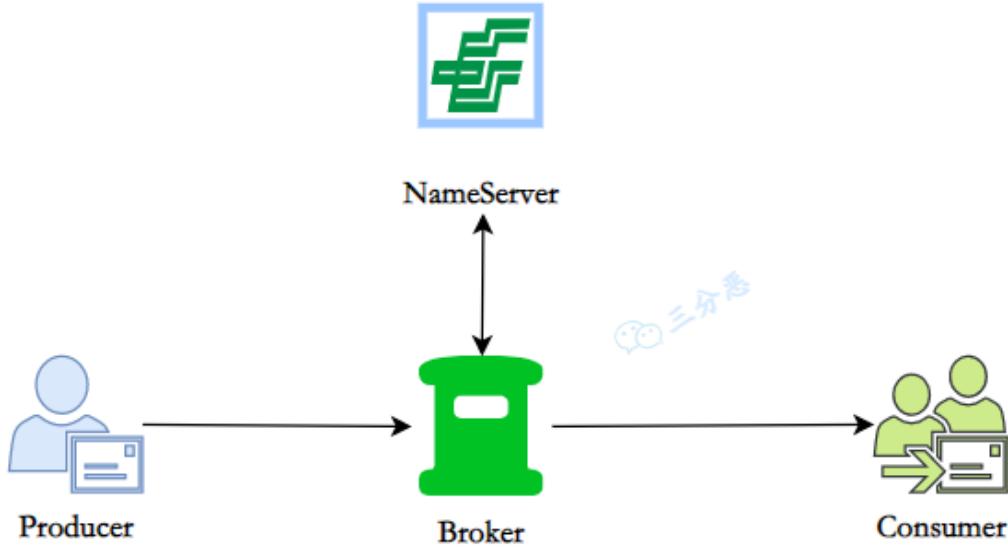


RocketMQ 一共有四个部分组成：NameServer，Broker，Producer 生产者，Consumer 消费者，它们对应了：发现、发、存、收，为了保证高可用，一般每一部分都是集群部署的。

8.那能介绍一下这四部分吗？

类比一下我们生活的邮政系统——

邮政系统要正常运行，离不开下面这四个角色，一是发信者，二是收信者，三是负责暂存传输的邮局，四是负责协调各个地方邮局的管理机构。对应到 RocketMQ 中，这四个角色就是 Producer、Consumer、Broker 、NameServer。



H6 NameServer

NameServer 是一个无状态的服务器，角色类似于 Kafka 使用的 Zookeeper，但比 Zookeeper 更轻量。特点：

- 每个 NameServer 结点之间是相互独立，彼此没有任何信息交互。
- Nameserver 被设计成几乎是无状态的，通过部署多个结点来标识自己是一个伪集群，Producer 在发送消息前从 NameServer 中获取 Topic 的路由信息也就是发往哪个 Broker，Consumer 也会定时从 NameServer 获取 Topic 的路由信息，Broker 在启动时会向 NameServer 注册，并定时进行心跳连接，且定时同步维护的 Topic 到 NameServer。

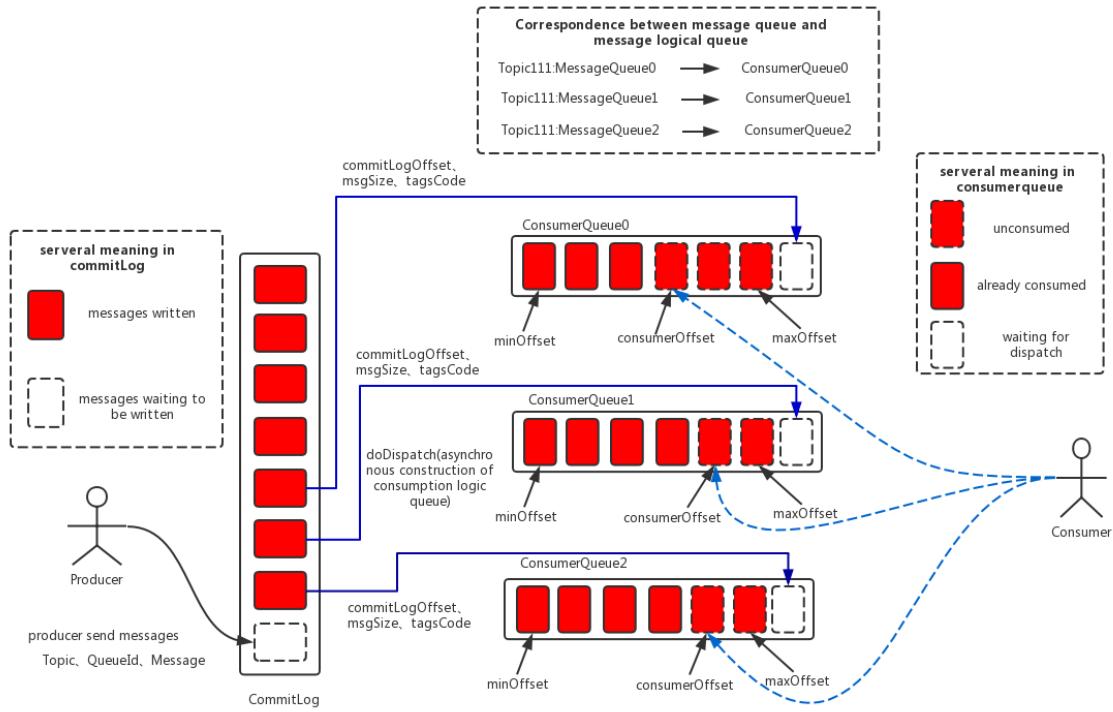
功能主要有两个：

- 1、和 Broker 结点保持长连接。
- 2、维护 Topic 的路由信息。

H6 Broker

消息存储和中转角色，负责存储和转发消息。

- Broker 内部维护着一个个 Consumer Queue，用来存储消息的索引，真正存储消息的地方是 CommitLog（日志文件）。



- 单个 Broker 与所有的 Nameserver 保持着长连接和心跳，并会定时将 Topic 信息同步到 NameServer，和 NameServer 的通信底层是通过 Netty 实现的。

H6 Producer

消息生产者，业务端负责发送消息，由用户自行实现和分布式部署。

- Producer**由用户进行分布式部署，消息由**Producer**通过多种负载均衡模式发送到**Broker**集群，发送低延时，支持快速失败。
- RocketMQ** 提供了三种方式发送消息：同步、异步和单向
 - 同步发送**：同步发送指消息发送方发出数据后会在收到接收方发回响应之后才发下一个数据包。一般用于重要通知消息，例如重要通知邮件、营销短信。
 - 异步发送**：异步发送指发送方发出数据后，不等接收方发回响应，接着发送下个数据包，一般用于可能链路耗时较长而对响应时间敏感的业务场景，例如用户视频上传后通知启动转码服务。
 - 单向发送**：单向发送是指只负责发送消息而不等待服务器回应且没有回调函数触发，适用于某些耗时非常短但对可靠性要求并不高的场景，例如日志收集。

H6 Consumer

消息消费者，负责消费消息，一般是后台系统负责异步消费。

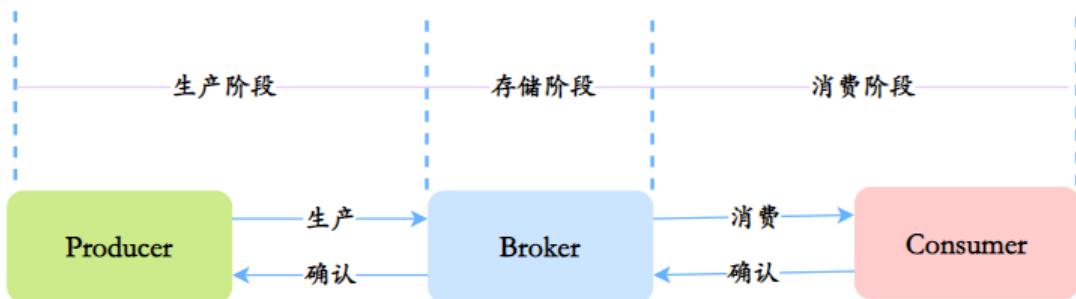
- Consumer 也由用户部署，支持PUSH和PULL两种消费模式，支持 集群消费 和 广播消费，提供 实时的消息订阅机制。
- Pull：拉取型消费者（Pull Consumer）主动从消息服务器拉取信息，只要批量拉 取到消息，用户应用就会启动消费过程，所以 Pull 称为主动消费型。
- Push：推送型消费者（Push Consumer）封装了消息的拉取、消费进度和其他的 内部维护工作，将消息到达时执行的回调接口留给用户应用程序来实现。所以 Push 称为被动消费类型，但其实从实现上看还是从消息服务器中拉取消息，不 同于 Pull 的是 Push 首先要注册消费监听器，当监听器处触发后才开始消费消 息。

进阶

9.如何保证消息的可用性/可靠性/不丢失呢？

消息可能在哪些阶段丢失呢？可能会在这三个阶段发生丢失：生产阶段、存储阶段、消费阶段。

所以要从这三个阶段考虑：



H6 生产

在生产阶段，主要通过请求确认机制，来保证消息的可靠传递。

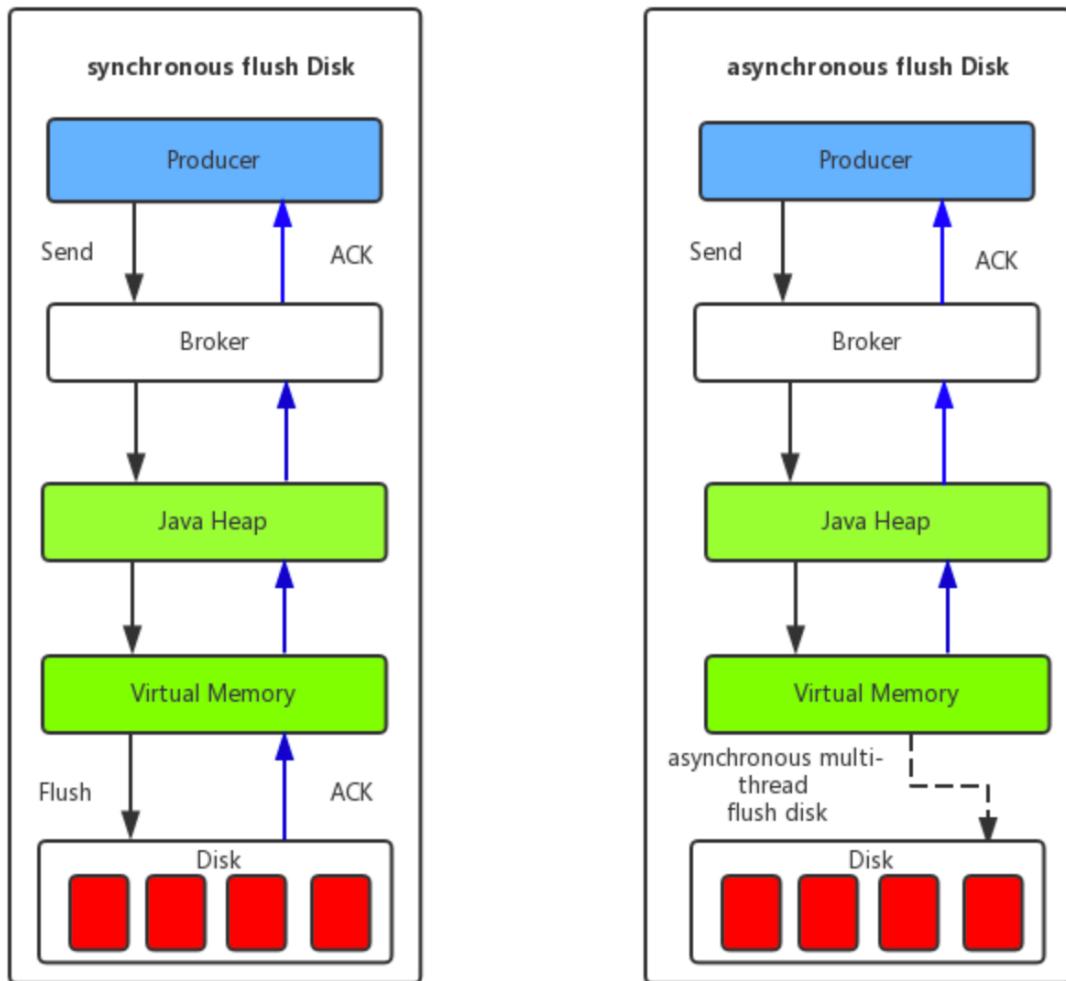
- 1、同步发送的时候，要注意处理响应结果和异常。如果返回响应OK，表示消息 成功发送到了Broker，如果响应失败，或者发生其它异常，都应该重试。
- 2、异步发送的时候，应该在回调方法里检查，如果发送失败或者异常，都应 该进行重试。

- 3、如果发生超时的情况，也可以通过查询日志的API，来检查是否在Broker存储成功。

H6 存储

存储阶段，可以通过**配置可靠性优先的 Broker 参数**来避免因为宕机丢消息，简单说就是可靠性优先的场景都应该使用同步。

- 1、消息只要持久化到CommitLog（日志文件）中，即使Broker宕机，未消费的消息也能重新恢复再消费。
- 2、Broker的刷盘机制：同步刷盘和异步刷盘，不管哪种刷盘都可以保证消息一定存储在pagecache中（内存中），但是同步刷盘更可靠，它是Producer发送消息后等数据持久化到磁盘之后再返回响应给Producer。



- 3、Broker通过主从模式来保证高可用，Broker支持Master和Slave同步复制、Master和Slave异步复制模式，生产者的消息都是发送给Master，但是消费既可以从Master消费，也可以从Slave消费。同步复制模式可以保证即使Master宕机，消息肯定在Slave中有备份，保证了消息不会丢失。

H6 消费

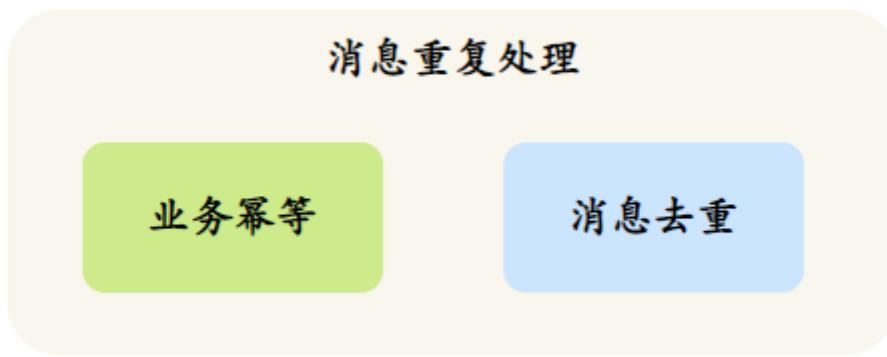
从Consumer角度分析，如何保证消息被成功消费？

- Consumer保证消息成功消费的关键在于确认的时机，不要在收到消息后就立即发送消费确认，而是应该在执行完所有消费业务逻辑之后，再发送消费确认。因为消息队列维护了消费的位置，逻辑执行失败了，没有确认，再去队列拉取消息，就还是之前的一条。

10.如何处理消息重复的问题呢？

对分布式消息队列来说，同时做到确保一定投递和不重复投递是很难的，就是所谓的“有且仅有一次”。RocketMQ择了确保一定投递，保证消息不丢失，但有可能造成消息重复。

处理消息重复问题，主要有业务端自己保证，主要的方式有两种：**业务幂等**和**消息去重**。



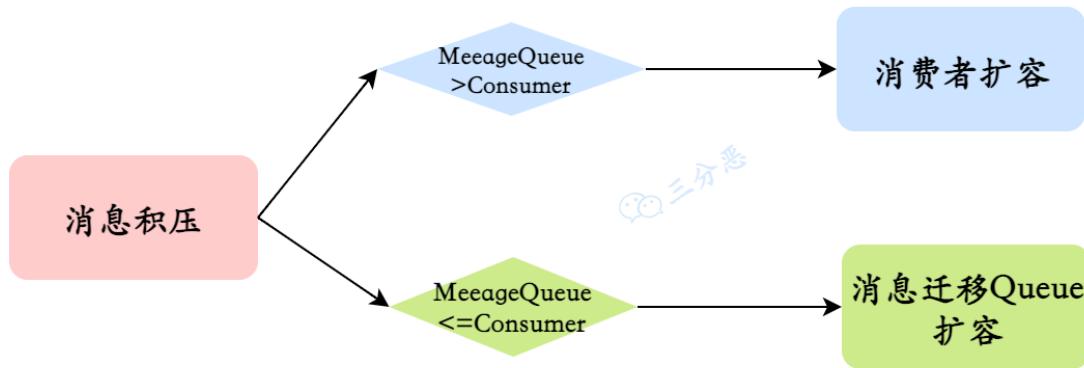
业务幂等：第一种是保证消费逻辑的幂等性，也就是多次调用和一次调用的效果是一样的。这样一来，不管消息消费多少次，对业务都没有影响。

消息去重：第二种是业务端，对重复的消息就不再消费了。这种方法，需要保证每条消息都有一个唯一的编号，通常是业务相关的，比如订单号，消费的记录需要落库，而且需要保证和消息确认这一步的原子性。

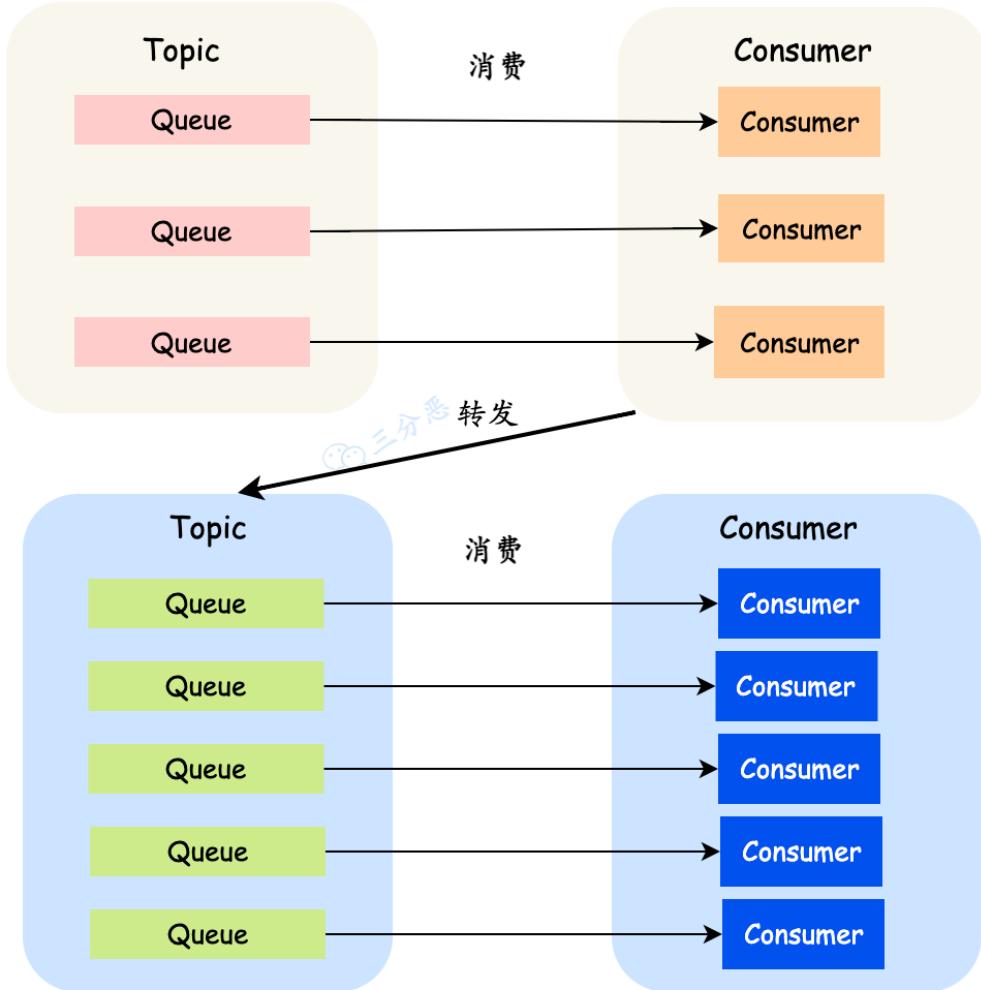
具体做法是可以建立一个消费记录表，拿到这个消息做数据库的insert操作。给这个消息做一个唯一主键（primary key）或者唯一约束，那么就算出现重复消费的情况，就会导致主键冲突，那么就不再处理这条消息。

11.怎么处理消息积压？

发生了消息积压，这时候就得想办法赶紧把积压的消息消费完，就得考虑提高消费能力，一般有两种办法：



- 消费者扩容：如果当前Topic的Message Queue的数量大于消费者数量，就可以对消费者进行扩容，增加消费者，来提高消费能力，尽快把积压的消息消费玩。
- 消息迁移Queue扩容：如果当前Topic的Message Queue的数量小于或者等于消费者数量，这种情况，再扩容消费者就没什么用，就得考虑扩容Message Queue。可以新建一个临时的Topic，临时的Topic多设置一些Message Queue，然后先用一些消费者把消费的数据丢到临时的Topic，因为不用业务处理，只是转发一下消息，还是很快的。接下来用扩容的消费者去消费新的Topic里的数据，消费完了之后，恢复原状。



12.顺序消息如何实现？

顺序消息是指消息的消费顺序和产生顺序相同，在有些业务逻辑下，必须保证顺序，比如订单的生成、付款、发货，这个消息必须按顺序处理才行。

顺序消息

全局顺序消息

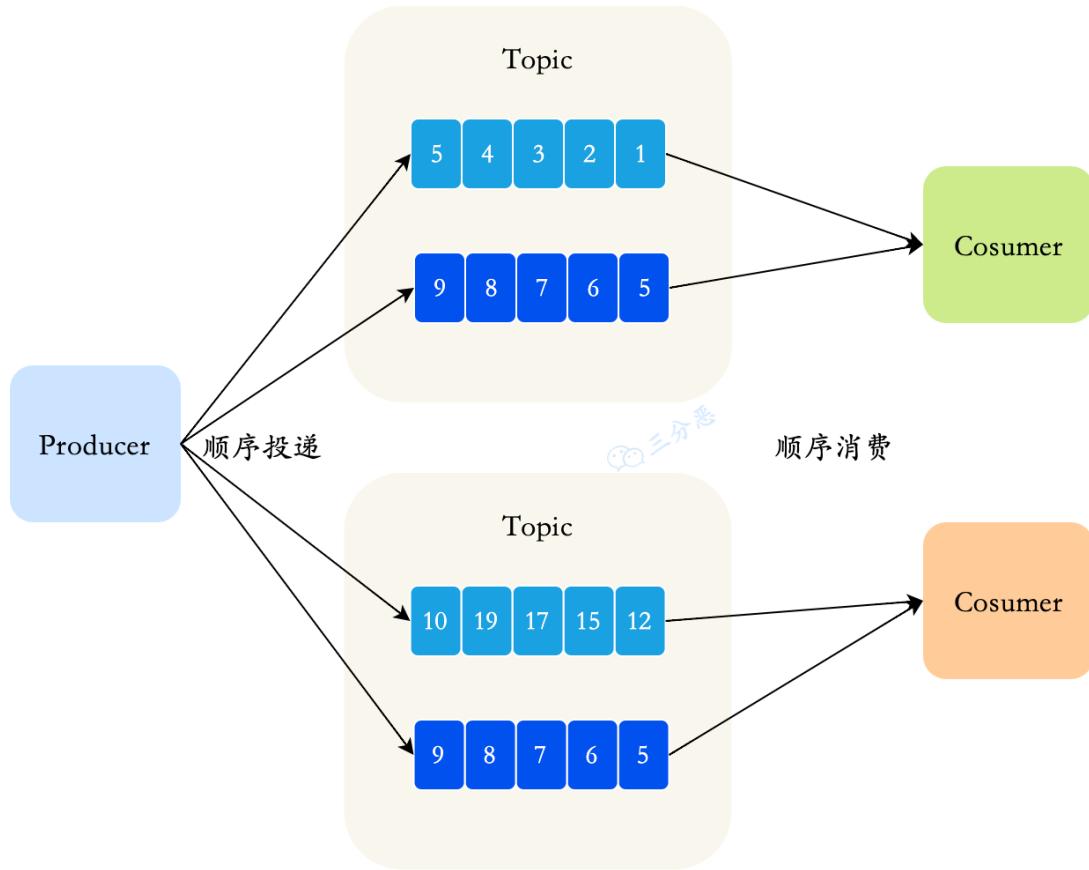
局部顺序消息

顺序消息分为全局顺序消息和部分顺序消息，全局顺序消息指某个 Topic 下的所有消息都要保证顺序；

部分顺序消息只要保证每一组消息被顺序消费即可，比如订单消息，只要保证同一个订单 ID 个消息能按顺序消费即可。

H6 部分顺序消息

部分顺序消息相对比较好实现，生产端需要做到把同 ID 的消息发送到同一个 Message Queue；在消费过程中，要做到从同一个Message Queue读取的消息顺序处理——消费端不能并发处理顺序消息，这样才能达到部分有序。



发送端使用 `MessageQueueSelector` 类来控制 把消息发往哪个 `Message Queue`。

```
● ● ●

package org.apache.rocketmq.example.order2;

import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.client.producer.MessageQueueSelector;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.common.message.MessageQueue;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

/**
 * Producer, 发送顺序消息
 */
public class Producer {

    public static void main(String[] args) throws Exception {
        DefaultMQProducer producer = new DefaultMQProducer("please_rename_unique_group_name");

        producer.setNamesrvAddr("127.0.0.1:9876");

        producer.start();

        String[] tags = new String[]{"TagA", "TagC", "TagD"};

        // 订单列表
        List<OrderStep> orderList = new Producer().buildOrders();

        Date date = new Date();
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String dateStr = sdf.format(date);
        for (int i = 0; i < 10; i++) {
            // 加个时间前缀
            String body = dateStr + " Hello RocketMQ " + orderList.get(i);
            Message msg = new Message("TopicTest", tags[i % tags.length], "KEY" + i,
                body.getBytes());
            SendResult sendResult = producer.send(msg, new MessageQueueSelector() {
                @Override
                public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg) {
                    Long id = (Long) arg; //根据订单id选择发送queue
                    long index = id % mqs.size();
                    return mqs.get((int) index);
                }
            }, orderList.get(i).getOrderId()); //订单id
            System.out.println(String.format("SendResult status:%s, queueId:%d, body:%s",
                sendResult.getSendStatus(),
                sendResult.getMessageQueue().getQueueId(),
                body));
        }

        producer.shutdown();
    }

    /**
     * 订单的步骤
     */
    private static class OrderStep {
        private long orderId;
        private String desc;

        public long getOrderId() {
            return orderId;
        }

        public void setOrderId(long orderId) {
            this.orderId = orderId;
        }

        public String getDesc() {
            return desc;
        }

        public void setDesc(String desc) {
            this.desc = desc;
        }

        @Override
        public String toString() {
            return "OrderStep{" +
                "orderId=" + orderId +
                ", desc='" + desc + '\'' +
                '}';
        }
    }

    /**
     * 生成模拟订单数据
     */
}
```

```
/*
private List<OrderStep> buildOrders() {
    List<OrderStep> orderList = new ArrayList<OrderStep>();

    OrderStep orderDemo = new OrderStep();
    orderDemo.setOrderId(15103111039L);
    orderDemo.setDesc("创建");
    orderList.add(orderDemo);

    orderDemo = new OrderStep();
    orderDemo.setOrderId(15103111065L);
    orderDemo.setDesc("创建");
    orderList.add(orderDemo);

    orderDemo = new OrderStep();
    orderDemo.setOrderId(15103111039L);
    orderDemo.setDesc("付款");
    orderList.add(orderDemo);

    orderDemo = new OrderStep();
    orderDemo.setOrderId(151031117235L);
    orderDemo.setDesc("创建");
    orderList.add(orderDemo);

    orderDemo = new OrderStep();
    orderDemo.setOrderId(15103111065L);
    orderDemo.setDesc("付款");
    orderList.add(orderDemo);

    orderDemo = new OrderStep();
    orderDemo.setOrderId(151031117235L);
    orderDemo.setDesc("付款");
    orderList.add(orderDemo);

    orderDemo = new OrderStep();
    orderDemo.setOrderId(15103111065L);
    orderDemo.setDesc("完成");
    orderList.add(orderDemo);

    orderDemo = new OrderStep();
    orderDemo.setOrderId(15103111039L);
    orderDemo.setDesc("推送");
    orderList.add(orderDemo);

    orderDemo = new OrderStep();
    orderDemo.setOrderId(151031117235L);
    orderDemo.setDesc("完成");
    orderList.add(orderDemo);

    orderDemo = new OrderStep();
    orderDemo.setOrderId(15103111039L);
    orderDemo.setDesc("完成");
    orderList.add(orderDemo);

    return orderList;
}
}
```

消费端通过使用 MessageListenerOrderly 来解决单 Message Queue 的消息被并发处理的问题。



```
package org.apache.rocketmq.example.order2;

import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.listener.ConsumeOrderlyContext;
import org.apache.rocketmq.client.consumer.listener.ConsumeOrderlyStatus;
import org.apache.rocketmq.client.consumer.listener.MessageListenerOrderly;
import org.apache.rocketmq.common.consumer.ConsumeFromWhere;
import org.apache.rocketmq.common.message.MessageExt;

import java.util.List;
import java.util.Random;
import java.util.concurrent.TimeUnit;

/**
 * 顺序消息消费，带事务方式（应用可控制Offset什么时候提交）
 */
public class ConsumerInOrder {

    public static void main(String[] args) throws Exception {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please_rename_unique_group_name_3");
        consumer.setNamesrvAddr("127.0.0.1:9876");
        /**
         * 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>
         * 如果非第一次启动，那么按照上次消费的位置继续消费
         */
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);

        consumer.subscribe("TopicTest", "TagA || TagC || TagD");

        consumer.registerMessageListener(new MessageListenerOrderly() {

            Random random = new Random();

            @Override
            public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs, ConsumeOrderlyContext context) {
                context.setAutoCommit(true);
                for (MessageExt msg : msgs) {
                    // 可以看到每个queue有唯一的consume线程来消费，订单对每个queue(分区)有序
                    System.out.println("consumeThread=" + Thread.currentThread().getName() + " queueId=" +
+ msg.getQueueId() + ", content:" + new String(msg.getBody()));
                }

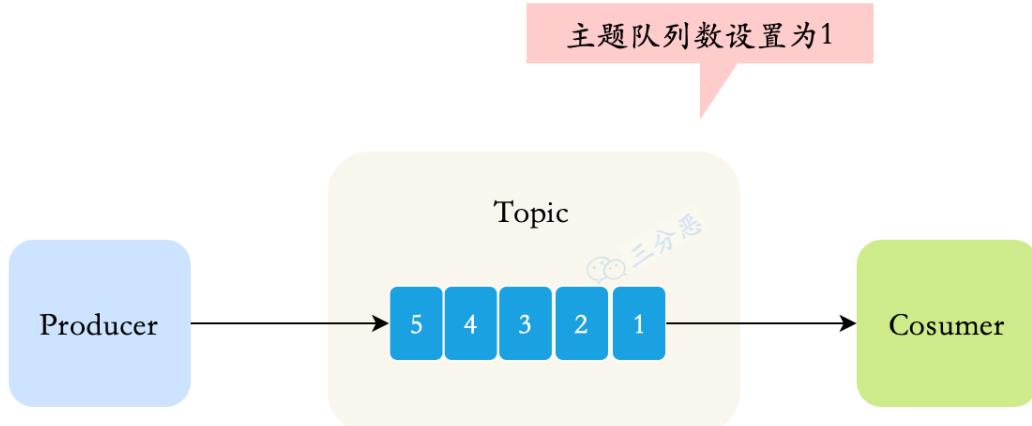
                try {
                    //模拟业务逻辑处理中...
                    TimeUnit.SECONDS.sleep(random.nextInt(10));
                } catch (Exception e) {
                    e.printStackTrace();
                }
                return ConsumeOrderlyStatus.SUCCESS;
            }
        });
        consumer.start();

        System.out.println("Consumer Started.");
    }
}
```

H6 全局顺序消息

RocketMQ 默认情况下不保证顺序，比如创建一个 Topic，默认八个写队列，八个读队列，这时候一条消息可能被写入任意一个队列里；在数据的读取过程中，可能有多个 Consumer，每个 Consumer 也可能启动多个线程并行处理，所以消息被哪个 Consumer 消费，被消费的顺序和写入的顺序是否一致是不确定的。

要保证全局顺序消息，需要先把 Topic 的读写队列数设置为一，然后 Producer Consumer 的并发设置，也要是一。简单来说，为了保证整个 Topic 全局消息有序，只能消除所有的并发处理，各部分都设置成单线程处理，这时候就完全牺牲 RocketMQ 的高并发、高吞吐的特性了。



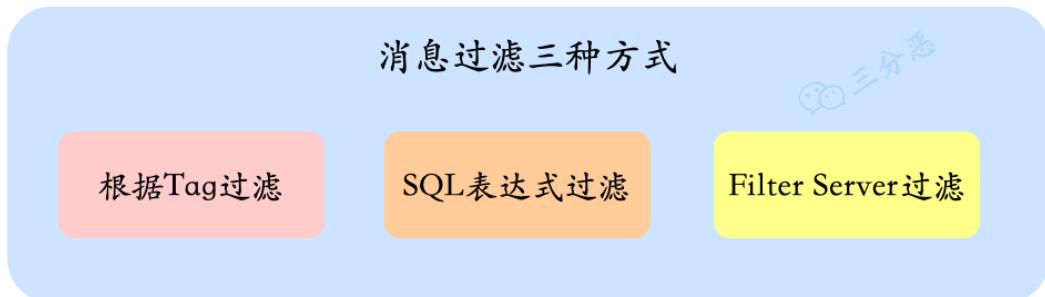
13. 如何实现消息过滤？

有两种方案：

- 一种是在 Broker 端按照 Consumer 的去重逻辑进行过滤，这样做的好处是避免了无用的消息传输到 Consumer 端，缺点是加重了 Broker 的负担，实现起来相对复杂。
- 另一种是在 Consumer 端过滤，比如按照消息设置的 tag 去重，这样的好处是实现起来简单，缺点是有大量无用的消息到达了 Consumer 端只能丢弃不处理。

一般采用 Consumer 端过滤，如果希望提高吞吐量，可以采用 Broker 过滤。

对消息的过滤有三种方式：



- 根据 Tag 过滤：这是最常见的一种，用起来高效简单

```
1 | DefaultMQPushConsumer consumer = new
2 | DefaultMQPushConsumer("CID_EXAMPLE");
3 | consumer.subscribe("TOPIC", "TAGA || TAGB || TAGC");
```

- SQL 表达式过滤: SQL表达式过滤更加灵活

```
1 | DefaultMQPushConsumer consumer = new
2 | DefaultMQPushConsumer("please_rename_unique_group_name_4")
3 | ;
4 | // 只有订阅的消息有这个属性a, a >=0 and a <= 3
5 | consumer.subscribe("TopicTest", MessageSelector.bySql("a
6 | between 0 and 3"));
7 | consumer.registerMessageListener(new
8 | MessageListenerConcurrently() {
9 |     @Override
10 |     public ConsumeConcurrentlyStatus
11 |     consumeMessage(List<MessageExt> msgs,
12 |     ConsumeConcurrentlyContext context) {
13 |         return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
14 |     }
15 | });
16 | consumer.start();
17 |
```

- Filter Server 方式: 最灵活, 也是最复杂的一种方式, 允许用户自定义函数进行过滤

14. 延时消息了解吗?

电商的订单超时自动取消, 就是一个典型的利用延时消息的例子, 用户提交了一个订单, 就可以发送一个延时消息, 1h后去检查这个订单的状态, 如果还是未付款就取消订单释放库存。

RocketMQ是支持延时消息的, 只需要在生产消息的时候设置消息的延时级别:

```
1 // 实例化一个生产者来产生延时消息
2 DefaultMQProducer producer = new
DefaultMQProducer("ExampleProducerGroup");
3 // 启动生产者
4 producer.start();
5 int totalMessagesToSend = 100;
6 for (int i = 0; i < totalMessagesToSend; i++) {
7     Message message = new Message("TestTopic", ("Hello
scheduled message " + i).getBytes());
8     // 设置延时等级3, 这个消息将在10s之后发送(现在只支持固定的几
个时间, 详看delayTimeLevel)
9     message.setDelayTimeLevel(3);
10    // 发送消息
11    producer.send(message);
12 }
```

但是目前RocketMQ支持的延时级别是有限的:

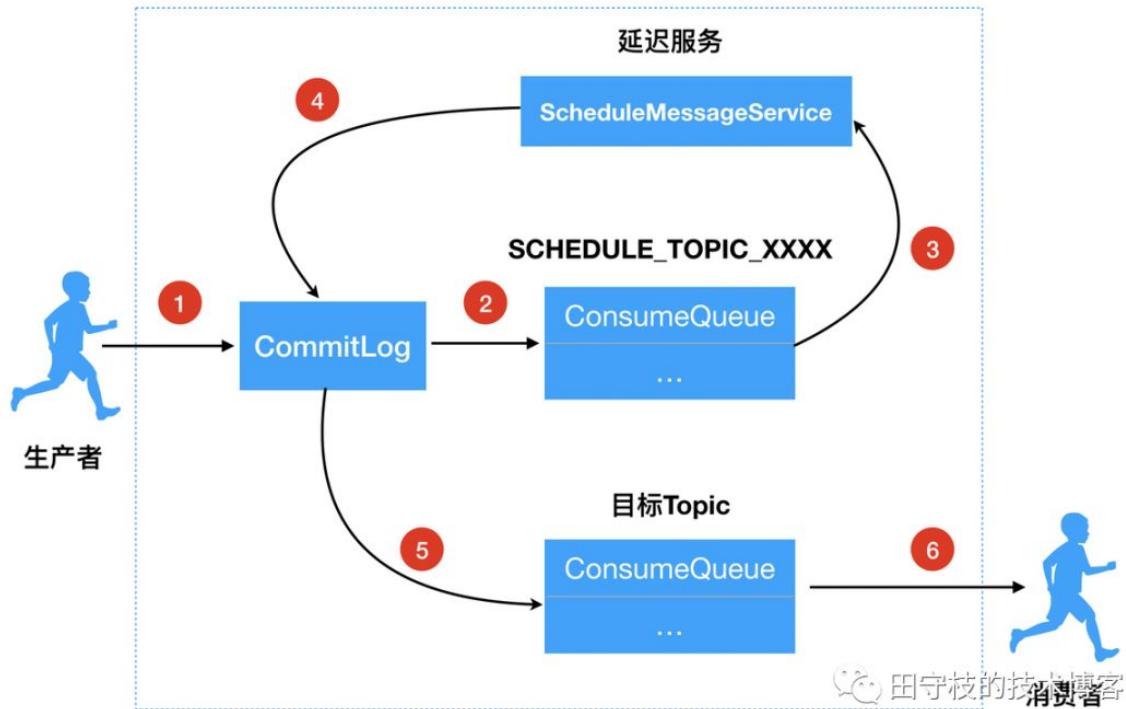
```
1 private String messageDelayLevel = "1s 5s 10s 30s 1m 2m 3m 4m
5m 6m 7m 8m 9m 10m 20m 30m 1h 2h";
```

H5 RocketMQ怎么实现延时消息的?

简单, 八个字: **临时存储 + 定时任务**。

Broker收到延时消息了, 会先发送到主题(SCHEDULE_TOPIC_XXXX)的相应时间段的Message Queue中, 然后通过一个定时任务轮询这些队列, 到期后, 把消息投递到目标Topic的队列中, 然后消费者就可以正常消费这些消息。

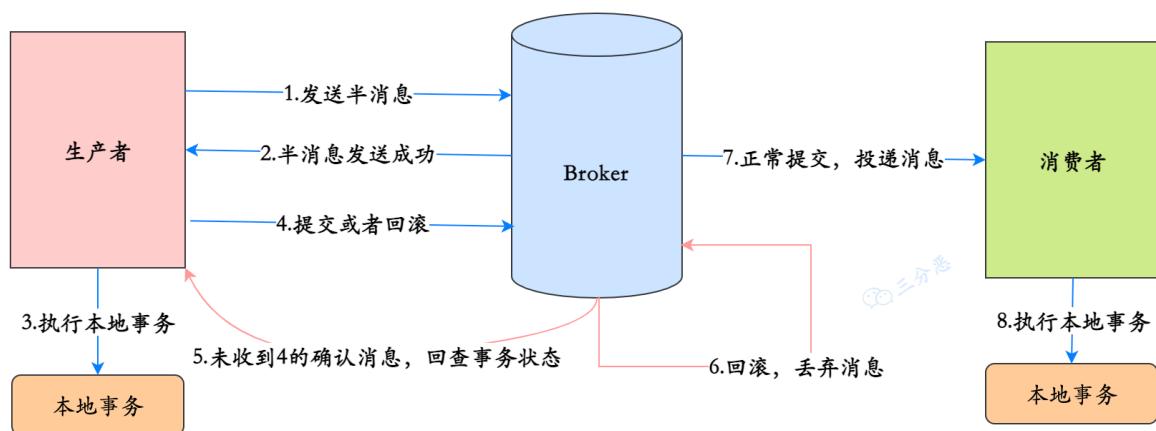
RocketMQ延迟消息Broker内部流转示意图



15. 怎么实现分布式消息事务的？半消息？

半消息：是指暂时还不能被 Consumer 消费的消息，Producer 成功发送到 Broker 端的消息，但是此消息被标记为“暂不可投递”状态，只有等 Producer 端执行完本地事务后经过二次确认了之后，Consumer 才能消费此条消息。

依赖半消息，可以实现分布式消息事务，其中的关键在于二次确认以及消息回查：



- 1、Producer 向 broker 发送半消息
- 2、Producer 端收到响应，消息发送成功，此时消息是半消息，标记为“不可投递”状态，Consumer 消费不了。
- 3、Producer 端执行本地事务。
- 4、正常情况本地事务执行完成，Producer 向 Broker 发送 Commit/Rollback，如果是 Commit，Broker 端将半消息标记为正常消息，Consumer 可以消费，如果是 Rollback，Broker 端将半消息标记为失败消息，Consumer 不可以消费。

Rollback，Broker 丢弃此消息。

- 5、异常情况，Broker 端迟迟等不到二次确认。在一定时间后，会查询所有的半消息，然后到 Producer 端查询半消息的执行情况。
- 6、Producer 端查询本地事务的状态
- 7、根据事务的状态提交 commit/rollback 到 broker 端。（5, 6, 7 是消息回查）
- 8、消费者段消费到消息之后，执行本地事务，执行本地事务。

16.死信队列知道吗？

死信队列用于处理无法被正常消费的消息，即死信消息。

当一条消息初次消费失败，**消息队列 RocketMQ 会自动进行消息重试**；达到最大重试次数后，若消费依然失败，则表明消费者在正常情况下无法正确地消费该消息，此时，消息队列 RocketMQ 不会立刻将消息丢弃，而是将其发送到该**消费者对应的特殊队列中**，该特殊队列称为**死信队列**。

死信消息的特点：

- 不会再被消费者正常消费。
- 有效期与正常消息相同，均为 3 天，3 天后会被自动删除。因此，需要在死信消息产生后的 3 天内及时处理。

死信队列的特点：

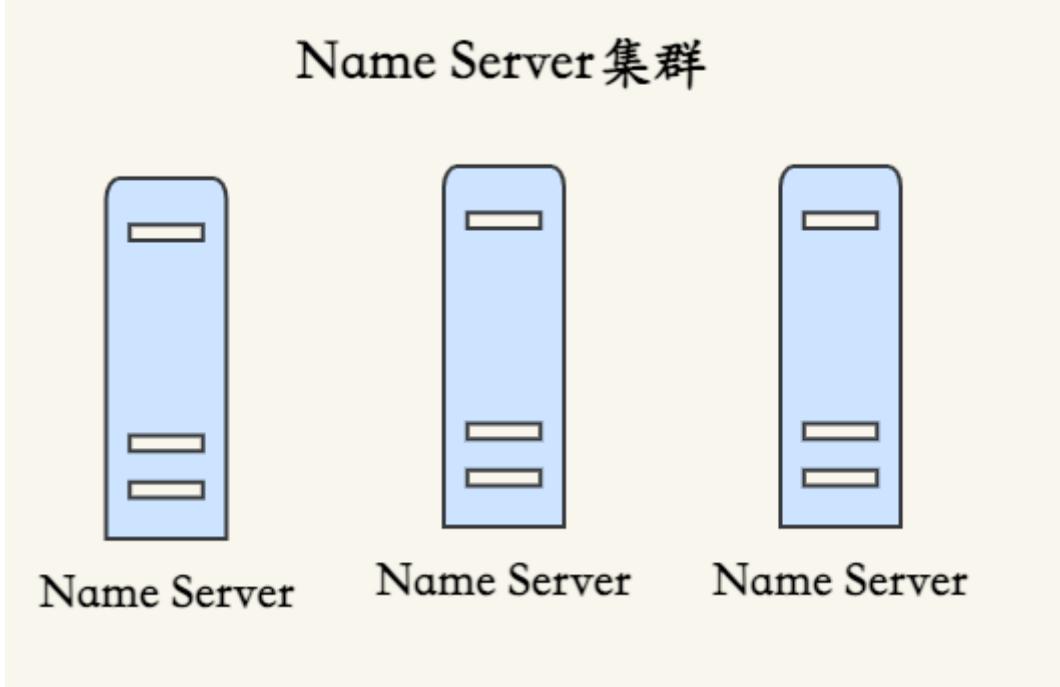
- 一个死信队列对应一个 Group ID，而不是对应单个消费者实例。
- 如果一个 Group ID 未产生死信消息，消息队列 RocketMQ 不会为其创建相应的死信队列。
- 一个死信队列包含了对应 Group ID 产生的所有死信消息，不论该消息属于哪个 Topic。

RocketMQ 控制台提供对死信消息的查询、导出和重发的功能。

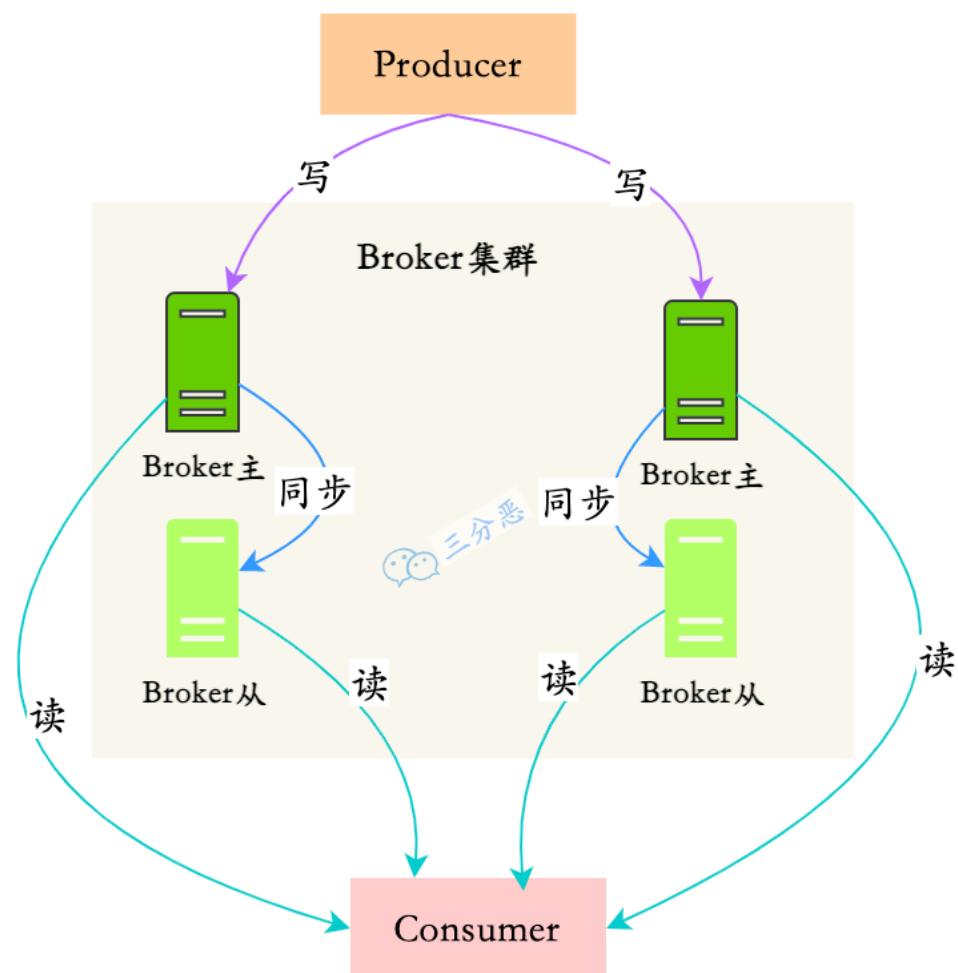
17.如何保证RocketMQ的高可用？

NameServer因为是无状态，且不相互通信的，所以只要集群部署就可以保证高可用。

Name Server 集群



RocketMQ的高可用主要是在体现在Broker的读和写的高可用，Broker的高可用是通过**集群**和**主从**实现的。



Broker可以配置两种角色：Master和Slave，Master角色的Broker支持读和写，Slave角色的Broker只支持读，Master会向Slave同步消息。

也就是说Producer只能向Master角色的Broker写入消息，Consumer可以从Master和Slave角色的Broker读取消息。

Consumer 的配置文件中，并不需要设置是从 Master 读还是从 Slave 读，当 Master 不可用或者繁忙的时候，Consumer 的读请求会被自动切换到从 Slave。有了自动切换 Consumer 这种机制，当一个 Master 角色的机器出现故障后，Consumer 仍然可以从 Slave 读取消息，不影响 Consumer 读取消息，这就实现了读的高可用。

如何达到发送端写的高可用性呢？在创建 Topic 的时候，把 Topic 的多个 Message Queue 创建在多个 Broker 组上（相同 Broker 名称，不同 brokerId 机器组成 Broker 组），这样当 Broker 组的 Master 不可用后，其他组 Master 仍然可用，Producer 仍然可以发送消息 RocketMQ 目前还不支持把 Slave 自动转成 Master，如果机器资源不足，需要把 Slave 转成 Master，则要手动停止 Slave 色的 Broker，更改配置文件，用新的配置文件启动 Broker。

原理

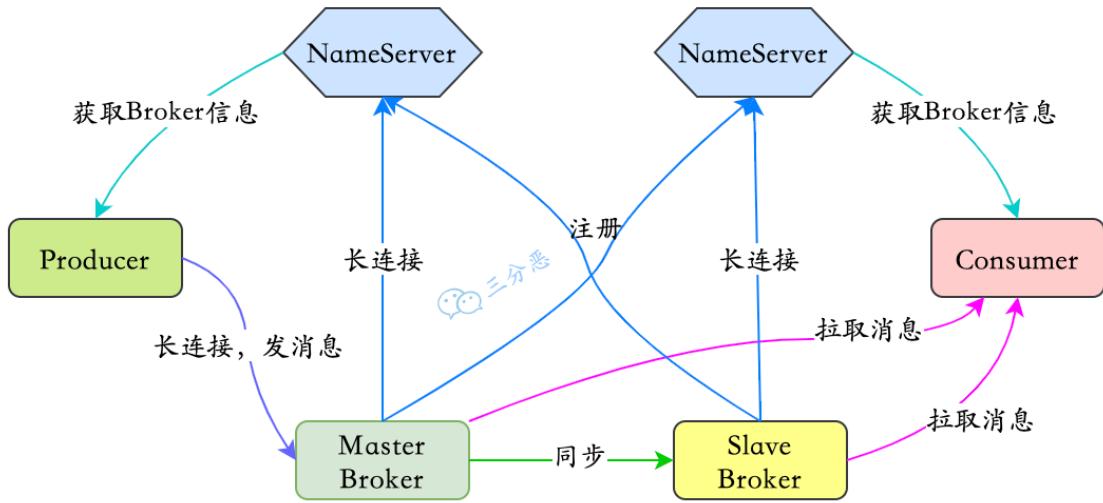
18. 说一下RocketMQ的整体工作流程？

简单来说，RocketMQ是一个分布式消息队列，也就是 消息队列 + 分布式系统。

作为消息队列，它是 发 - 存 - 收 的一个模型，对应的就是 Producer、Broker、Consumer；作为分布式系统，它要有服务端、客户端、注册中心，对应的就是 Broker、Producer/Consumer、NameServer

所以我们看一下它主要的工作流程：RocketMQ由NameServer注册中心集群、Producer生产者集群、Consumer消费者集群和若干Broker（RocketMQ进程）组成：

1. Broker在启动的时候去向所有的NameServer注册，并保持长连接，每30s发送一次心跳
2. Producer在发送消息的时候从NameServer获取Broker服务器地址，根据负载均衡算法选择一台服务器来发送消息
3. Consumer消费消息的时候同样从NameServer获取Broker地址，然后主动拉取消息来消费



19.为什么RocketMQ不使用Zookeeper作为注册中心呢？

Kafka我们都知道采用Zookeeper作为注册中心——当然也开始逐渐去Zookeeper，RocketMQ不使用Zookeeper其实主要可能从这几方面来考虑：

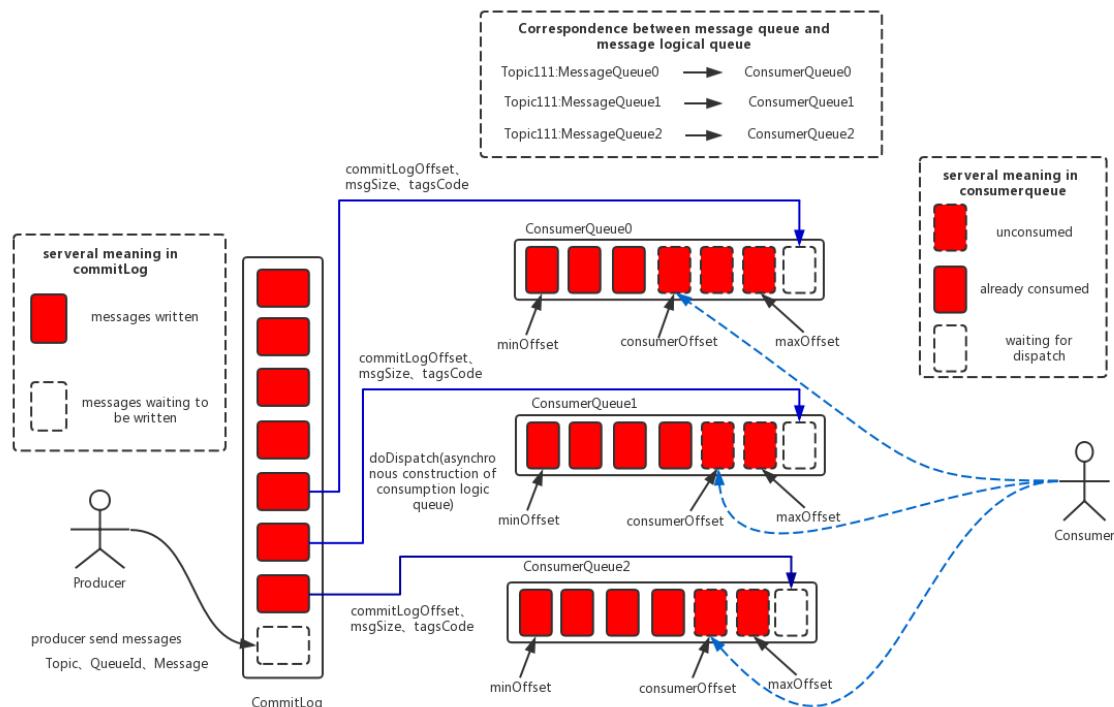
1. 基于可用性的考虑，根据CAP理论，同时最多只能满足两个点，而Zookeeper满足的是CP，也就是说Zookeeper并不能保证服务的可用性，Zookeeper在进行选举的时候，整个选举的时间太长，期间整个集群都处于不可用的状态，而对于一个注册中心来说肯定是不能接受的，作为服务发现来说就应该是为可用性而设计。
2. 基于性能的考虑，NameServer本身的实现非常轻量，而且可以通过增加机器的方式水平扩展，增加集群的抗压能力，而Zookeeper的写是不可扩展的，Zookeeper要解决这个问题只能通过划分领域，划分多个Zookeeper集群来解决，首先操作起来太复杂，其次这样还是又违反了CAP中的A的设计，导致服务之间是不连通的。
3. 持久化的机制带来的问题，ZooKeeper的ZAB协议对每一个写请求，会在每个ZooKeeper节点上保持写一个事务日志，同时再加上定期的将内存数据镜像(Snapshot)到磁盘来保证数据的一致性和持久性，而对于一个简单的服务发现的场景来说，这其实没有太大的必要，这个实现方案太重了。而且本身存储的数据应该是高度定制化的。
4. 消息发送应该弱依赖注册中心，而RocketMQ的设计理念也正是基于此，生产者在第一次发送消息的时候从NameServer获取到Broker地址后缓存到本地，如果NameServer整个集群不可用，短时间内对于生产者和消费者并不会产生太大影响。

20.Broker是怎么保存数据的呢？

RocketMQ主要的存储文件包括CommitLog文件、ConsumeQueue文件、Indexfile文件。

rocketmq > store			
名称	修改日期	类型	大小
commitlog	2018/3/4 12:55	文件夹	
config	2018/2/26 23:56	文件夹	
consumequeue	2018/2/26 21:38	文件夹	
index	2018/2/26 21:37	文件夹	
abort	2018/2/4 17:08	文件	0 KB
checkpoint	2018/2/4 17:08	文件	4 KB

消息存储的整体的设计：



- **CommitLog:** 消息主体以及元数据的存储主体，存储Producer端写入的消息主体内容,消息内容不是定长的。单个文件大小默认1G, 文件名长度为20位, 左边补零, 剩余为起始偏移量, 比如00000000000000000000代表了第一个文件, 起始偏移量为0, 文件大小为1G=1073741824; 当第一个文件写满了, 第二个文件为00000000001073741824, 起始偏移量为1073741824, 以此类推。消息主要是顺序写入日志文件, 当文件满了, 写入下一个文件。

CommitLog文件保存于\${Rocket_Home}/store/commitlog目录中，从图中我们可以明显看出来文件名的偏移量，每个文件默认1G，写满后自动生成一个新的文件。

rocketmq > store > commitlog			
名称	修改日期	类型	大小
00000000000000000000	2018/2/4 17:09	文件	1,048,576 KB
0000000001073741824	2018/2/4 17:09	文件	1,048,576 KB

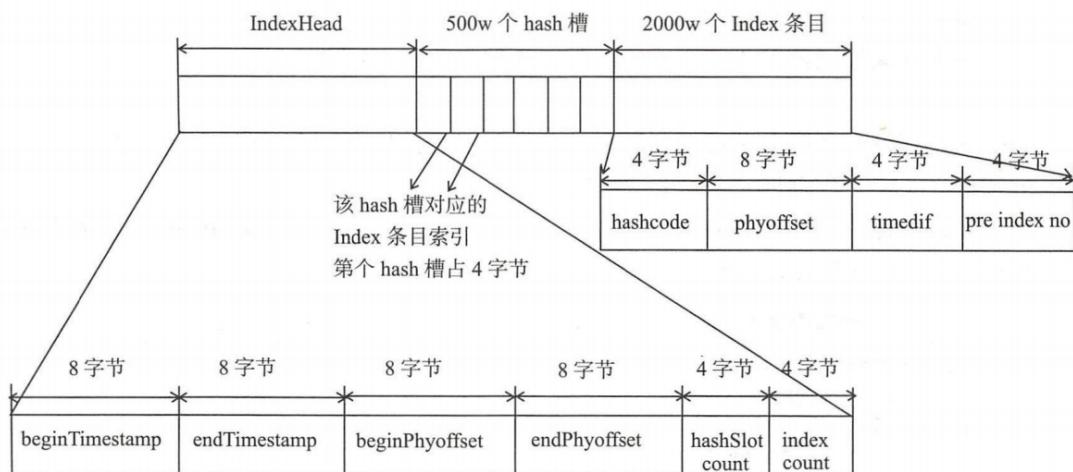
- **ConsumeQueue**: 消息消费队列，引入的目的主要是提高消息消费的性能，由于RocketMQ是基于主题topic的订阅模式，消息消费是针对主题进行的，如果要遍历commitlog文件中根据topic检索消息是非常低效的。

Consumer即可根据ConsumeQueue来查找待消费的消息。其中，ConsumeQueue（逻辑消费队列）作为消费消息的索引，保存了指定Topic下的队列消息在CommitLog中的起始物理偏移量offset，消息大小size和消息Tag的HashCode值。

ConsumeQueue文件可以看成是基于Topic的CommitLog索引文件，故ConsumeQueue文件夹的组织方式如下：topic/queue/file三层组织结构，具体存储路径为：\$HOME/store/consumequeue/{topic}/{queueId}/{fileName}。同样ConsumeQueue文件采取定长设计，每一个条目共20个字节，分别为8字节的CommitLog物理偏移量、4字节的消息长度、8字节tag hashcode，单个文件由30W个条目组成，可以像数组一样随机访问每一个条目，每个ConsumeQueue文件大小约5.72M；

名称	修改日期	类型
00000000000000000000	2018/2/26 23:40	文件

- **IndexFile**: IndexFile（索引文件）提供了一种可以通过key或时间区间来查询消息的方法。Index文件的存储位置是：\$HOME\store\index\$\{fileName}，文件名fileName是以创建时的时间戳命名的，固定的单个IndexFile文件大小约为400M，一个IndexFile可以保存 2000W个索引，IndexFile的底层存储设计为在文件系统中实现HashMap结构，故RocketMQ的索引文件其底层实现为hash索引。



总结一下：RocketMQ采用的是混合型的存储结构，即为Broker单个实例下所有的队列共用一个日志数据文件（即为CommitLog）来存储。

RocketMQ的混合型存储结构(多个Topic的消息实体内容都存储于一个CommitLog中)针对Producer和Consumer分别采用了数据和索引部分相分离的存储结构，Producer发送消息至Broker端，然后Broker端使用同步或者异步的方式对消息刷盘持久化，保存至CommitLog中。

只要消息被刷盘持久化至磁盘文件CommitLog中，那么Producer发送的消息就不会丢失。正因为如此，Consumer也就肯定有机会去消费这条消息。当无法拉取到消息后，可以等下一次消息拉取，同时服务端也支持长轮询模式，如果一个消息拉取请求未拉取到消息，Broker允许等待30s的时间，只要这段时间内有新消息到达，将直接返回给消费端。

这里，RocketMQ的具体做法是，使用Broker端的后台服务线程—ReputMessageService不停地分发请求并异步构建ConsumeQueue（逻辑消费队列）和IndexFile（索引文件）数据。



21. 说说RocketMQ怎么对文件进行读写的？

RocketMQ对文件的读写巧妙地利用了操作系统的一些高效文件读写方式——

PageCache、**顺序读写**、**零拷贝**。

- PageCache、顺序读取

在RocketMQ中，ConsumeQueue逻辑消费队列存储的数据较少，并且是顺序读取，在page cache机制的预读取作用下，Consume Queue文件的读性能几乎接近读内存，即使在有消息堆积情况下也不会影响性能。而对于CommitLog消息存储的日志数据文件来说，读取消息内容时候会产生较多的随机访问读取，严重影响性能。如果选择合适的系统IO调度算法，比如设置调度算法为“Deadline”（此时块存储采用SSD的话），随机读的性能也会有所提升。

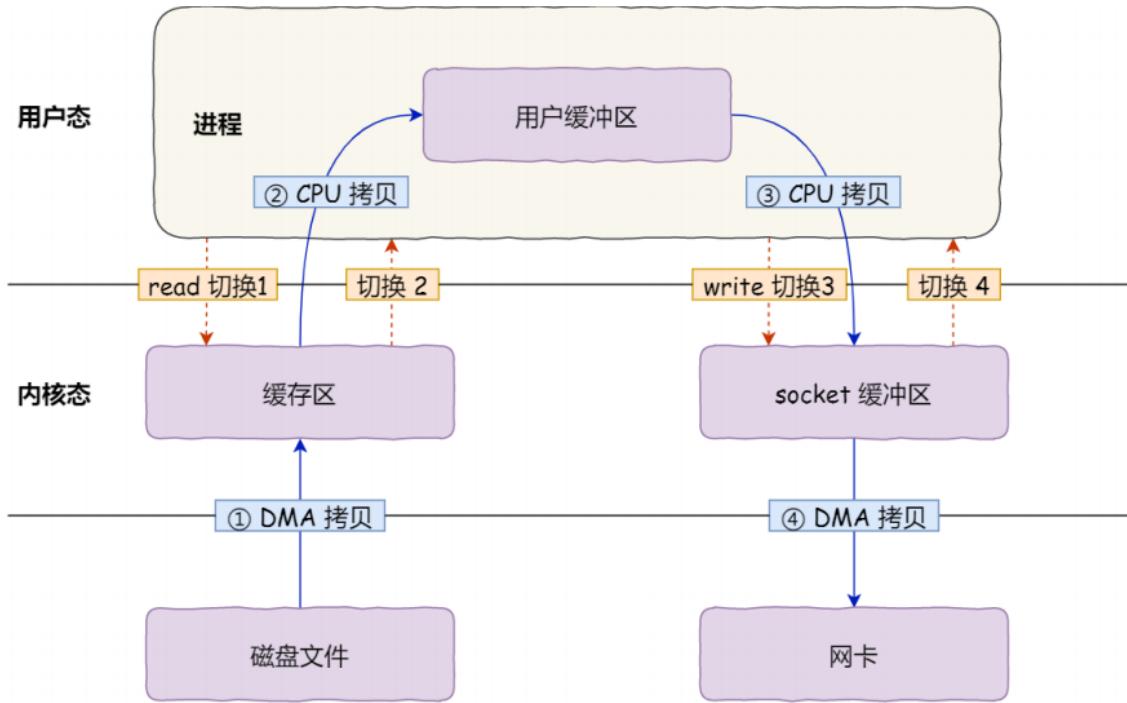
页缓存（PageCache）是OS对文件的缓存，用于加速对文件的读写。一般来说，程序对文件进行顺序读写的速度几乎接近于内存的读写速度，主要原因就是由于OS使用PageCache机制对读写访问操作进行了性能优化，将一部分的内存用作PageCache。对于数据的写入，OS会先写入至Cache内，随后通过异步的方式由pdflush内核线程将Cache内的数据刷盘至物理磁盘上。对于数据的读取，如果一次读取文件时出现未命中PageCache的情况，OS从物理磁盘上访问读取文件的同时，会顺序对其他相邻块的数据文件进行预读取。

- 零拷贝

另外，RocketMQ主要通过MappedByteBuffer对文件进行读写操作。其中，利用了NIO中的FileChannel模型将磁盘上的物理文件直接映射到用户态的内存地址中（这种Mmap的方式减少了传统IO，将磁盘文件数据在操作系统内核地址空间的缓冲区，和用户应用程序地址空间的缓冲区之间来回进行拷贝的性能开销），将对文件的操作转化为直接对内存地址进行操作，从而极大地提高了文件的读写效率（正因为需要使用内存映射机制，故RocketMQ的文件存储都使用定长结构来存储，方便一次将整个文件映射至内存）。

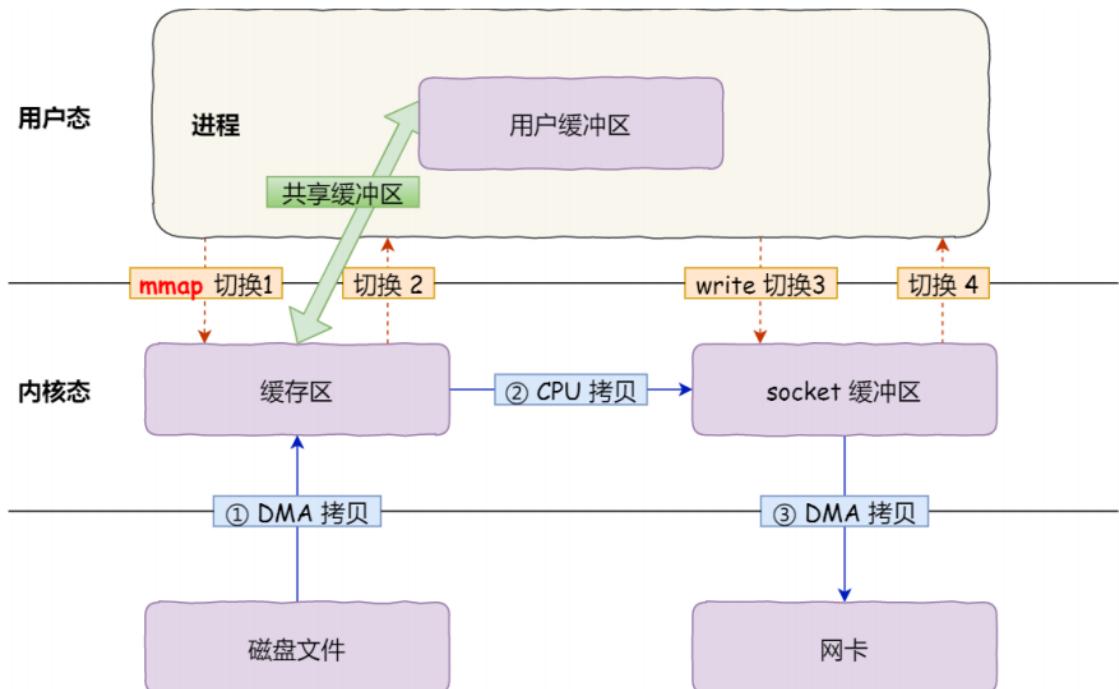
H6 说说什么是零拷贝？

在操作系统中，使用传统的方式，数据需要经历几次拷贝，还要经历用户态/内核态切换。



1. 从磁盘复制数据到内核态内存；
2. 从内核态内存复制到用户态内存；
3. 然后从用户态内存复制到网络驱动的内核态内存；
4. 最后是从网络驱动的内核态内存复制到网卡中进行传输。

所以，可以通过零拷贝的方式，**减少用户态与内核态的上下文切换和内存拷贝的次数**，用来提升I/O的性能。零拷贝比较常见的实现方式是**mmap**，这种机制在Java中是通过**MappedByteBuffer**实现的。



22.消息刷盘怎么实现的呢？

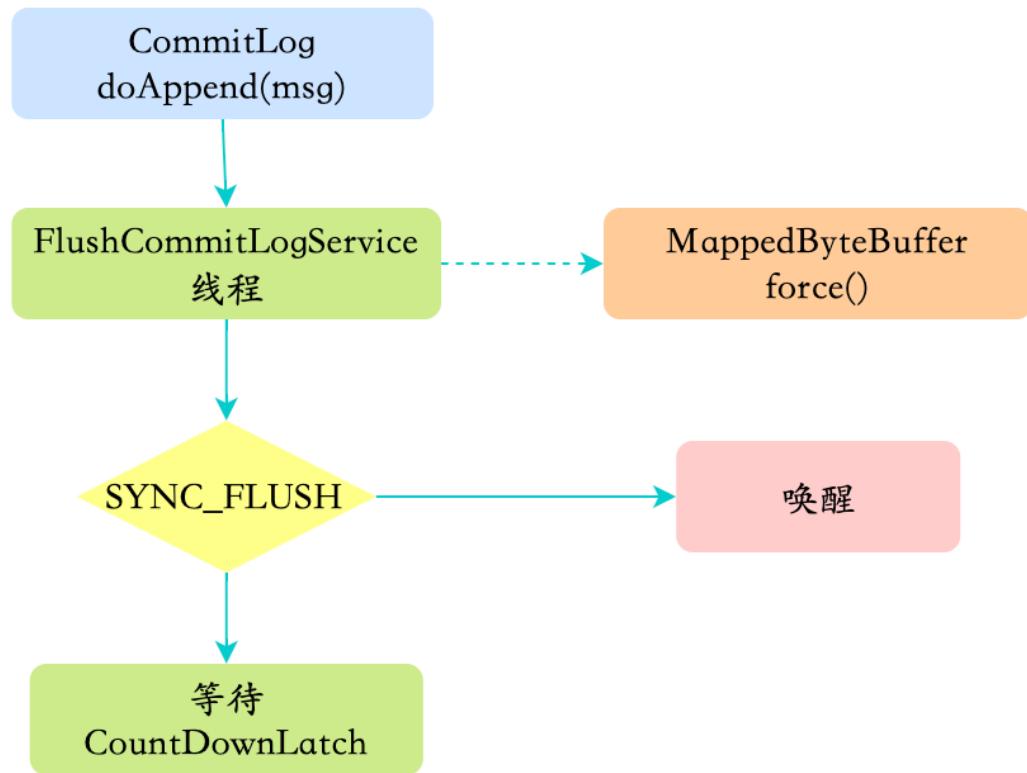
RocketMQ提供了两种刷盘策略：同步刷盘和异步刷盘

- 同步刷盘：在消息达到Broker的内存之后，必须刷到commitLog日志文件中才算成功，然后返回Producer数据已经发送成功。
- 异步刷盘：异步刷盘是指消息达到Broker内存后就返回Producer数据已经发送成功，会唤醒一个线程去将数据持久化到CommitLog日志文件中。

Broker 在消息的存取时直接操作的是内存（内存映射文件），这可以提供系统的吞吐量，但是无法避免机器掉电时数据丢失，所以需要持久化到磁盘中。

刷盘的最终实现都是使用**NIO**中的 `MappedByteBuffer.force()` 将映射区的数据写入到磁盘，如果是同步刷盘的话，在**Broker**把消息写到**CommitLog**映射区后，就会等待写入完成。

异步而言，只是唤醒对应的线程，不保证执行的时机，流程如图所示。



22.能说下 RocketMQ 的负载均衡是如何实现的？

RocketMQ中的负载均衡都在Client端完成，具体来说的话，主要可以分为Producer端发送消息时候的负载均衡和Consumer端订阅消息的负载均衡。

H6 Producer的负载均衡

Producer端在发送消息的时候，会先根据Topic找到指定的TopicPublishInfo，在获取了TopicPublishInfo路由信息后，RocketMQ的客户端在默认方式下selectOneMessageQueue()方法会从TopicPublishInfo中的messageQueueList中选择一个队列（MessageQueue）进行发送消息。具这里有一个sendLatencyFaultEnable开关变量，如果开启，在随机递增取模的基础上，再过滤掉not available的Broker代理。

Producer负载均衡：索引递增随机取模

```
/**  
 * 选择一个MessageQueue  
 */  
public MessageQueue selectOneMessageQueue() {  
    //索引递增  
    int index = this.sendWhichQueue.incrementAndGet();  
    //利用索引取随机数，取余数  
    int pos = Math.abs(index) % this.messageQueueList.size();  
    if (pos < 0)  
        pos = 0;  
    return this.messageQueueList.get(pos);  
}
```

所谓的"latencyFaultTolerance"，是指对之前失败的，按一定的时间做退避。例如，如果上次请求的latency超过550Lms，就退避3000Lms；超过1000L，就退避60000L；如果关闭，采用随机递增取模的方式选择一个队列（MessageQueue）来发送消息，latencyFaultTolerance机制是实现消息发送高可用的核心关键所在。

H6 Consumer的负载均衡

在RocketMQ中，Consumer端的两种消费模式（Push/Pull）都是基于拉模式来获取消息的，而在Push模式只是对pull模式的一种封装，其本质实现为消息拉取线程在从服务器拉取到一批消息后，然后提交到消息消费线程池后，又“马不停蹄”的继续向服务器再次尝试拉取消息。如果未拉取到消息，则延迟一下又继续拉取。在两种基于拉模式的消费方式（Push/Pull）中，均需要Consumer端知道从Broker端的哪一个消息队列中去获取消息。因此，有必要在Consumer端来做负载均衡，即Broker端中多个MessageQueue分配给同一个ConsumerGroup中的哪些Consumer消费。

1. Consumer端的心跳包发送

在Consumer启动后，它就会通过定时任务不断地向RocketMQ集群中的所有Broker实例发送心跳包（其中包含了，消息消费分组名称、订阅关系集合、消息通信模式和客户端id的值等信息）。Broker端在收到Consumer的心跳消息后，会将它维护在ConsumerManager的本地缓存变量—consumerTable，同时并将封装后的客户端网络通

道信息保存在本地缓存变量—channelInfoTable中，为之后做Consumer端的负载均衡提供可以依据的元数据信息。

2. Consumer端实现负载均衡的核心类—RebalanceImpl

在Consumer实例的启动流程中的启动MQClientInstance实例部分，会完成负载均衡服务线程—RebalanceService的启动（每隔20s执行一次）。

通过查看源码可以发现，RebalanceService线程的run()方法最终调用的是RebalanceImpl类的rebalanceByTopic()方法，这个方法是实现Consumer端负载均衡的核心。

rebalanceByTopic()方法会根据消费者通信类型为“广播模式”还是“集群模式”做不同的逻辑处理。这里主要来看下集群模式下的主要处理流程：

```
● ● ●

//集群模式负载均衡
case CLUSTERING: {
    //获取该Topic主题下的消息消费队列集合
    Set<MessageQueue> mqSet = this.topicSubscribeInfoTable.get(topic);
    //找到消费者Id列表
    List<String> cidAll = this.mQClientFactory.findConsumerIdList(topic, consumerGroup);
    if (null == mqSet) {
        if (!topic.startsWith(MixAll.RETRY_GROUP_TOPIC_PREFIX)) {
            this.messageQueueChanged(topic, Collections.<MessageQueue>emptySet(),
Collections.<MessageQueue>emptySet());
            log.warn("doRebalance, {}, but the topic[{}] not exist.", consumerGroup,
topic);
        }
    }

    if (null == cidAll) {
        log.warn("doRebalance, {} {}, get consumer id list failed", consumerGroup, topic);
    }

    if (mqSet != null && cidAll != null) {
        List<MessageQueue> mqAll = new ArrayList<MessageQueue>();
        mqAll.addAll(mqSet);

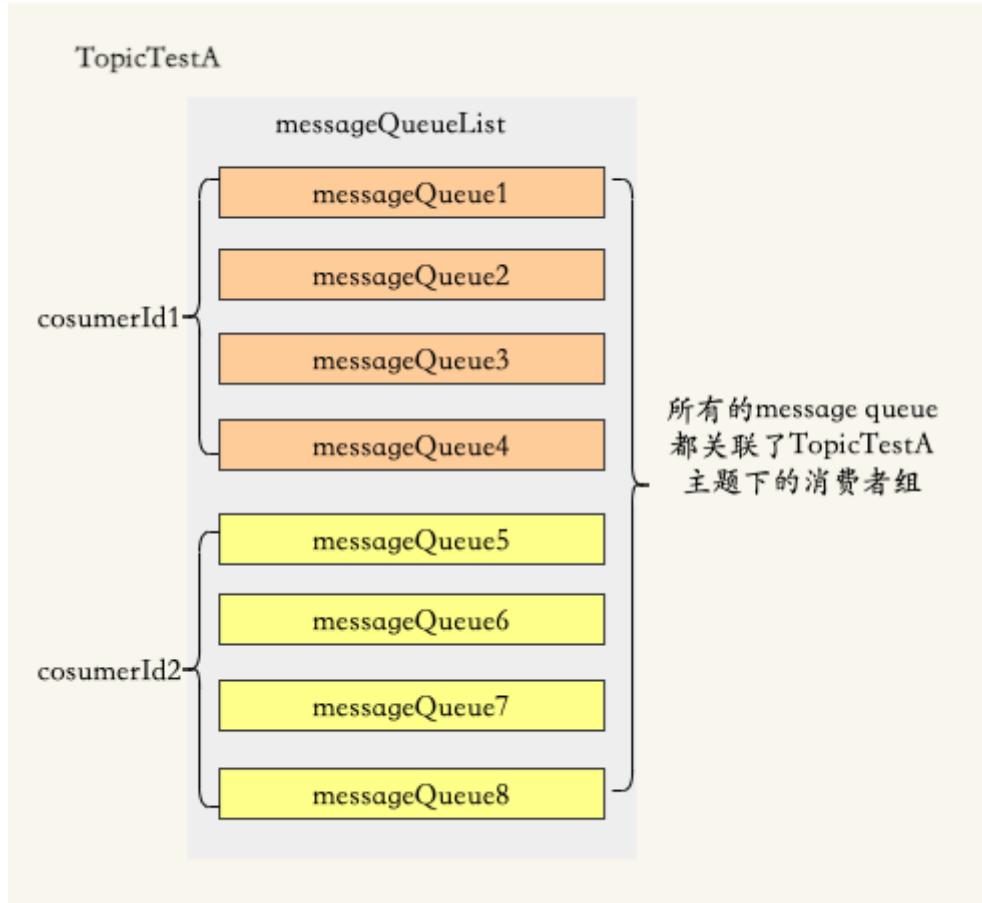
        Collections.sort(mqAll);
        Collections.sort(cidAll);
        //消费者消息分配负载均衡策略
        AllocateMessageQueueStrategy strategy = this.allocateMessageQueueStrategy;
        //分配队列的结果
        List<MessageQueue> allocateResult = null;
        try {
            //相应的策略类获取负载队列结果
            allocateResult = strategy.allocate(
                this.consumerGroup,
                this.mQClientFactory.getClientId(),
                mqAll,
                cidAll);
        } catch (Throwable e) {
            log.error("allocate message queue exception. strategy name: {}, ex: {}", strategy.getName(), e);
            return false;
        }
        //消息队列结果去重
        Set<MessageQueue> allocateResultSet = new HashSet<MessageQueue>();
        if (allocateResult != null) {
            allocateResultSet.addAll(allocateResult);
        }

        boolean changed = this.updateProcessQueueTableInRebalance(topic, allocateResultSet,
isOrder);
        if (changed) {
            log.info(
                "client rebalanced result changed. allocateMessageQueueStrategyName={},",
                group={}, topic={}, clientId={}, mqAllSize={}, cidAllSize={}, rebalanceResultSetSize={},
                rebalanceResultSet={}",
                strategy.getName(), consumerGroup, topic,
this.mQClientFactory.getClientId(), mqSet.size(), cidAll.size(),
                allocateResultSet.size(), allocateResultSet);
            this.messageQueueChanged(topic, mqSet, allocateResultSet);
        }

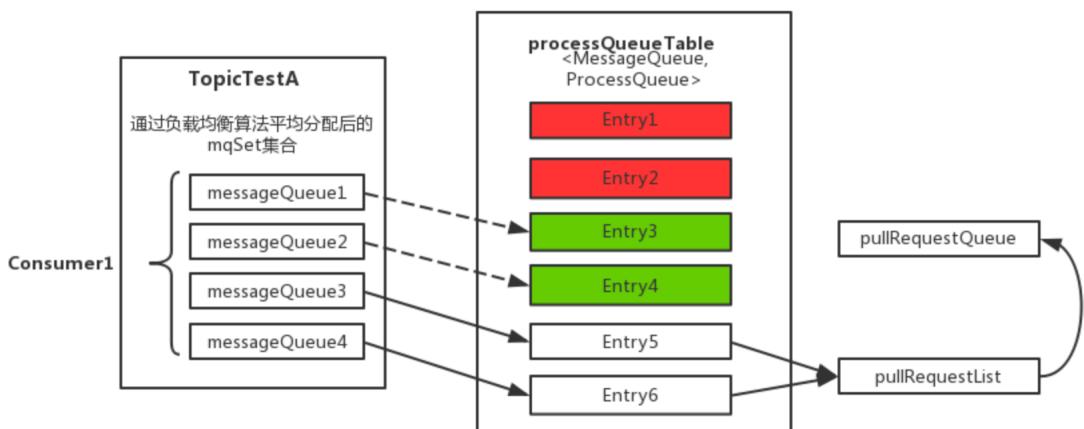
        balanced = allocateResultSet.equals(getWorkingMessageQueue(topic));
    }
    break;
}
```

- (1) 从rebalanceImpl实例的本地缓存变量—topicSubscribeInfoTable中， 获取该Topic主题下的消息消费队列集合（mqSet）；
- (2) 根据topic和consumerGroup为参数调用mQClientFactory.findConsumerIdList()方法向Broker端发送通信请求， 获取该消费组下消费者Id列表；

(3) 先对Topic下的消息消费队列、消费者Id排序，然后用消息队列分配策略算法（默认认为：消息队列的平均分配算法），计算出待拉取的消息队列。这里的平均分配算法，类似于分页的算法，将所有MessageQueue排好序类似于记录，将所有消费端Consumer排好序类似页数，并求出每一页需要包含的平均size和每个页面记录的范围range，最后遍历整个range而计算出当前Consumer端应该分配到的的MessageQueue。



(4) 然后，调用`updateProcessQueueTableInRebalance()`方法，具体的做法是，先将分配到的消息队列集合（`mqSet`）与`processQueueTable`做一个过滤比对。



- 上图中`processQueueTable`标注的红色部分，表示与分配到的消息队列集合`mqSet`互不包含。将这些队列设置`Dropped`属性为`true`，然后查看这些队列是否可以移除出`processQueueTable`缓存变量，这里具体执行`removeUnnecessaryMessageQueue()`

方法，即每隔1s查看是否可以获取当前消费处理队列的锁，拿到的话返回true。

如果等待1s后，仍然拿不到当前消费处理队列的锁则返回false。如果返回true，

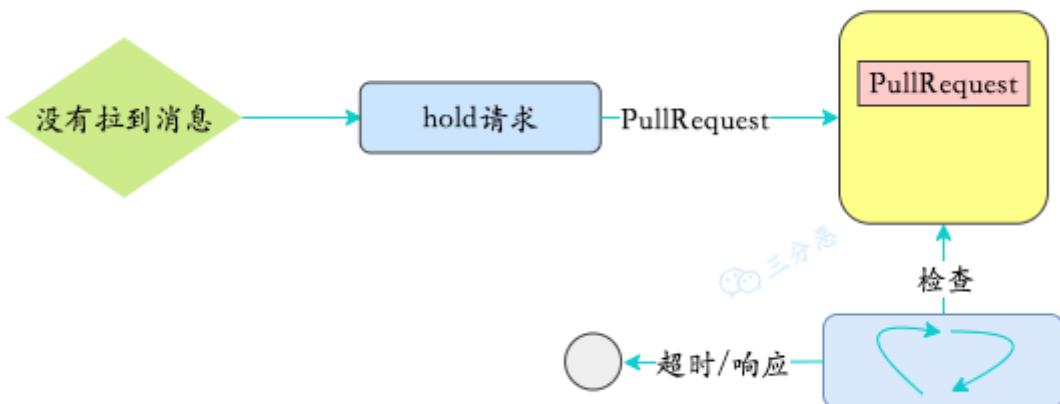
则从processQueueTable缓存变量中移除对应的Entry；

- 上图中processQueueTable的绿色部分，表示与分配到的消息队列集合mqSet的交集。判断该ProcessQueue是否已经过期了，在Pull模式的不用管，如果是Push模式的，设置Dropped属性为true，并且调用removeUnnecessaryMessageQueue()方法，像上面一样尝试移除Entry；
- 最后，为过滤后的消息队列集合（mqSet）中的每个MessageQueue创建一个ProcessQueue对象并存入RebalanceImpl的processQueueTable队列中（其中调用RebalanceImpl实例的computePullFromWhere(MessageQueue mq)方法获取该MessageQueue对象的下一个进度消费值offset，随后填充至接下来要创建的pullRequest对象属性中），并创建拉取请求对象—pullRequest添加到拉取列表—pullRequestList中，最后执行dispatchPullRequest()方法，将Pull消息的请求对象PullRequest依次放入PullMessageService服务线程的阻塞队列pullRequestQueue中，待该服务线程取出后向Broker端发起Pull消息的请求。其中，可以重点对比下，RebalancePushImpl和RebalancePullImpl两个实现类的dispatchPullRequest()方法不同，RebalancePullImpl类里面的该方法为空。

消息消费队列在同一消费组不同消费者之间的负载均衡，其核心设计理念是在一个消息消费队列在同一时间只允许被同一消费组内的一个消费者消费，一个消息消费者能同时消费多个消息队列。

23.RocketMQ消息长轮询了解吗？

所谓的长轮询，就是Consumer 拉取消息，如果对应的 Queue 如果没有数据，Broker 不会立即返回，而是把 PullReuqest hold起来，等待 queue 有了消息后，或者长轮询阻塞时间到了，再重新处理该 queue 上的所有 PullRequest。



- PullMessageProcessor#processRequest

```

1          //如果没有拉到数据
2      case ResponseCode.PULL_NOT_FOUND:
3          // broker 和 consumer 都允许 suspend, 默认开启
4          if (brokerAllowSuspend &&
5              hasSuspendFlag) {
6              long pollingTimeMills =
7                  suspendTimeoutMillisLong;
8              if
9                  (!this.brokerController.getBrokerConfig().isLongPollingEnabled()) {
10                  pollingTimeMills =
11                      this.brokerController.getBrokerConfig().getShortPollingTimeMills();
12              }
13
14          String topic =
15              requestHeader.getTopic();
16          long offset =
17              requestHeader.getQueueOffset();
18          int queueId =
19              requestHeader.getQueueId();
20          //封装一个PullRequest
21          PullRequest pullRequest = new
22              PullRequest(request, channel, pollingTimeMills,
23
24              this.brokerController.getMessageStore().now(), offset,
25              subscriptionData, messageFilter);
26          //把PullRequest挂起来
27
28          this.brokerController.getPullRequestHoldService().suspendPullRequest(topic, queueId, pullRequest);
29          response = null;
30          break;
31      }

```

挂起的请求，有一个服务线程会不停地检查，看queue中是否有数据，或者超时。

- PullRequestHoldService#run()

```
1     @Override
2     public void run() {
3         log.info("{} service started",
4             this.getServiceName());
5         while (!this.isStopped()) {
6             try {
7                 if
8                     (this.brokerController.getBrokerConfig().isLongPollingEnable(
9 )) {
10                     this.waitForRunning(5 * 1000);
11                 } else {
12
13                     this.waitForRunning(this.brokerController.getBrokerConfig().g
14 etShortPollingTimeMills());
15                 }
16
17                     long beginLockTimestamp =
18             this.systemClock.now();
19                     //检查hold住的请求
20                     this.checkHoldRequest();
21                     long costTime = this.systemClock.now() -
22             beginLockTimestamp;
23                     if (costTime > 5 * 1000) {
24                         log.info("[NOTIFYME] check hold request
25 cost {} ms.", costTime);
26                     }
27                     } catch (Throwable e) {
28                         log.warn(this.getServiceName() + " service
29 has exception. ", e);
30                     }
31                     }
32
33                     log.info("{} service end", this.getServiceName());
34     }
```

参考：

- [1]. 《RocketMQ实战与原理解析》
- [2]. 《RocketMQ技术内幕》

- [3]. [面试被问到RocketMq，我懵了](#)
- [4]. 艾小仙《我要进大厂》
- [5]. <http://dreamcat.ink/java-interview/docs/knows/classify/dis/RocketMQ/>
- [6]. [《浅入浅出》-RocketMQ](#)
- [7]. [十二张图，踹开消息队列的大门](#)
- [8]. [mq的那些破事儿，你不好奇吗？](#)
- [9]. [消息幂等（去重）如何解决？来看看这个方案！](#)
- [10]. [七万字，151张图，通宵整理消息队列核心知识点总结！这次彻底掌握MQ！](#)
- [11]. 极客时间《消息队列高手课》
- [12]. RocketMQ官网

关注公众号：三分恶

手册更新动态
即刻送达



添加个人微信：ThirdFighter

技术交流
加大佬云集微信群

