

## 概述

---

### 1. 面向对象和面向过程的区别?

答: 面向过程的思想是自顶向下, 逐步细化的实现功能, 面向对象是封装事物的属性和行为, 即抽象化后, 协调调用事物来实现功能。

#### 面向过程

**优点:** 性能比面向对象高, 因为类调用时需要实例化, 开销比较大, 比较消耗资源, 比如单片机、嵌入式开发、Linux/Unix等一般采用面向过程开发, 性能是最重要的因素。

**缺点:** 没有面向对象易维护、易复用、易扩展

#### 面向对象

**优点:** 易维护、易复用、易扩展, 由于面向对象有封装、继承、多态性的特性, 可以设计出低耦合的系统, 使系统更加灵活、更加易于维护。

**缺点:** 性能比面向过程低

### 2. 面向对象的特征有哪些?

答:

**抽象:** 抽象是将一类对象的共同特征总结出来构造类的过程, 包括数据抽象和行为抽象两方面, 抽象只关注对象的哪些属性和行为, 并不关注这些行为的细节是什么。

**封装:** 将对象抽象称为一个高度自治和抽象的个体, 对象的状态由这个对象自己的行为(方法)来读取。

**继承:** 是从已有类得到继承信息创建新类的过程, 继承让变化中的软件系统有了一定的延续性。

**多态:** 指允许不同类的对象对同一消息作出响应。

### 3. jdk与jre的区别与联系?

答:

**JDK:** Java Development Kit 的简称, Java 开发工具包, 提供了 Java 的开发环境和运行环境。

**JRE:** Java Runtime Environment 的简称, Java 运行环境, 为 Java 的运行提供了所需环境。

具体来说 JDK 其实包含了 JRE, 同时还包含了编译 Java 源码的编译器 Javac, 还包含了很多 Java 程序调试和分析的工具。简单来说: 如果你需要运行 Java 程序, 只需安装 JRE 就可以了, 如果你需要编写 Java 程序, 需要安装 JDK。

## 基本数据类型

---

### 1. 强类型语言和弱类型语言区别?

答:

**强类型定义语言:** 强制数据类型定义的语言。也就是说, 一旦一个变量被指定了某个数据类型, 如果不经过强制转换, 那么它就永远是这个数据类型了。

**弱类型定义语言:** 数据类型可以被忽略的语言。它与强类型定义语言相反, 一个变量可以赋不同数据类型的值。

强类型定义语言在速度上可能略逊色于弱类型定义语言, 但是强类型定义语言带来的严谨性能够有效的避免许多错误。

### 2. java基本数据类型有哪些?

答:

**byte:** 8位, 最大存储数据量是255, 存放的数据范围是-128~127之间。

**short:** 16位, 最大数据存储量是65536, 数据范围是-32768~32767之间。

**int:** 32位, 最大数据存储容量是2的32次方减1, 数据范围是负的2的31次方到正的2的31次方减1。

**long:** 64位, 最大数据存储容量是2的64次方减1, 数据范围为负的2的63次方到正的2的63次方减1。

**float:** 32位, 数据范围在3.4e-45~1.4e38, 直接赋值时必须在数字后加上f或F。

**double**: 64位, 数据范围在4.9e-324~1.8e308, 赋值时可以加d或D也可以不加。

**boolean**: 只有true和false两个取值。Java虚拟机不提供操作boolean类型的字节码指令, 程序在编译后boolean类型都转化成了int操作。但是Java虚拟机支持boolean类型的数组的访问和修改, 共用byte类型数组的字节码指令。

**char**: 16位, 存储Unicode码, 用单引号赋值。

**returnAddress类型**: 会被Java虚拟机的jsr、ret和jsr\_w指令所使用。returnAddress类型的值指向一条虚拟机指令的操作码。与前面介绍的那些数值类的原生类型不同, returnAddress类型在Java语言之中并不存在相应的类型, 也无法在程序运行期间更改returnAddress类型的值。

3. String属于基本数据类型吗?

答:

属于对象。

4. 数值类型转换规则?

答:

简单类型数据间的转换, 有两种方式:自动转换和强制转换

### 自动转换

具体地讲,当一个较"小"数据与一个较"大"的数据一起运算时,系统将自动将"小"数据转换成"大"数据,再进行运算。这些类型由"小"到"大"分别为 (byte, short, char)->int->long->float->double。这里我们所说的"大"与"小",并不是指占用字节的多少,而是指表示值的范围的大小。

①下面的语句可以在Java中直接通过:

```
byte b; int i=b; long l=b; float f=b; double d=b;
```

②如果低级类型为char型, 向高级类型(整型)转换时, 会转换为对应ASCII码值, 例如  
char c='c'; int i=c; System.out.println("output:"+i); 输出: output:99;

③对于byte,short,char三种类型而言, 他们是平级的, 因此不能相互自动转换, 可以使用下述的强制类型转换。

```
short i=99; char c=(char)i; System.out.println("output:"+c); 输出: output:c;
```

### 强制转换

将"大"数据转换为"小"数据时, 你可以使用强制类型转换。即你必须采用下面这种语句格式: int n=(int)3.14159/2; 可以想象, 这种转换肯定可能会导致溢出或精度的下降。

### 表达式的数据类型自动提升注意下面的规则:

- ①所有的byte,short,char型的值将被提升为int型;
- ②如果有一个操作数是long型, 计算结果是long型;
- ③如果有一个操作数是float型, 计算结果是float型;
- ④如果有一个操作数是double型, 计算结果是double型;

例, byte b; b=3; b=(byte)(b\*3); //必须声明byte。

对于short i=1; i=i+1; 由于i是int类型, 因此i+1运算结果也是int类型, 需要强制转换类型才能赋值给short类型; 而short i=1; i+=1; 可以正确编译, 因为i+=1; 相当于i=(short)(i+1); 其中有隐含的强制类型转换。拓展: short i=1; i++; 也是正确的。需要注意的是, 整数都默认为int类型, 除非你将其定义为short类型, 因此1+short类型的i还是int类型, 所以会报错; 而i++并没有经过i+1赋值语句, 因此不会报错, 而i+=1等价于i++, 所以也不会报错。

5. unicode详解?

答:

这里引用阮一峰的一篇日志《字符编码笔记: ASCII, Unicode 和 UTF-8》, 具体如下:

### ASCII 码

我们知道, 计算机内部, 所有信息最终都是一个二进制值。每一个二进制位(bit)有0和1两种状态, 因此八个二进制位就可以组合出256种状态, 这被称为一个字节(byte)。也就是说, 一个字节一共可以用来表示256种不同的状态, 每一个状态对应一个符号, 就是256个符号, 从00000000到11111111。

上个世纪60年代, 美国制定了一套字符编码, 对英语字符与二进制位之间的关系, 做了统一规定。这被称为 ASCII 码, 一直沿用至今。

ASCII 码一共规定了128个字符的编码, 比如空格SPACE是32(二进制00100000), 大写的字母A是65(二进制01000001)。这128个符号(包括32个不能打印出来的控制符号), 只占用了一个字节的后面7位, 最前面的一位统一规定为0。

## 非 ASCII 编码

英语用128个符号编码就够了，但是用来表示其他语言，128个符号是不够的。比如，在法语中，字母上方有注音符号，它就无法用 ASCII 码表示。于是，一些欧洲国家就决定，利用字节中闲置的最高位编入新的符号。比如，法语中的é的编码为130（二进制10000010）。这样一来，这些欧洲国家使用的编码体系，可以表示最多256个符号。

但是，这里又出现了新的问题。不同的国家有不同的字母，因此，哪怕它们都使用256个符号的编码方式，代表的字母却不一样。比如，130在法语编码中代表了é，在希伯来语编码中却代表了字母Gimel (ג)，在俄语编码中又会代表另一个符号。但是不管怎样，所有这些编码方式中，0-127表示的符号是一样的，不一样的只是128--255的这一段。

至于亚洲国家的文字，使用的符号就更多了，汉字就多达10万左右。一个字节只能表示256种符号，肯定是不够的，就必须使用多个字节表达一个符号。比如，简体中文常见的编码方式是 GB2312，使用两个字节表示一个汉字，所以理论上最多可以表示  $256 \times 256 = 65536$  个符号。

中文编码的问题需要专文讨论，这篇笔记不涉及。这里只指出，虽然都是用多个字节表示一个符号，但是GB类的汉字编码与后文的 Unicode 和 UTF-8 是毫无关系的。

## Unicode

正如上一节所说，世界上存在着多种编码方式，同一个二进制数字可以被解释成不同的符号。因此，要想打开一个文本文件，就必须知道它的编码方式，否则用错误的编码方式解读，就会出现乱码。为什么电子邮件常常出现乱码？就是因为发信人和收信人使用的编码方式不一样。

可以想象，如果有一种编码，将世界上所有的符号都纳入其中。每一个符号都给予一个独一无二的编码，那么乱码问题就会消失。这就是 Unicode，就像它的名字都表示的，这是一种所有符号的编码。

Unicode 当然是一个很大的集合，现在的规模可以容纳100多万个符号。每个符号的编码都不一样，比如，U+0639表示阿拉伯字母Ain，U+0041表示英语的大写字母A，U+4E25表示汉字严。具体的符号对应表，可以查询unicode.org，或者专门的汉字对应表。

## Unicode 的问题

需要注意的是，Unicode 只是一个符号集，它只规定了符号的二进制代码，却没有规定这个二进制代码应该如何存储。

比如，汉字严的 Unicode 是十六进制数4E25，转换成二进制数足足有15位（100111000100101），也就是说，这个符号的表示至少需要2个字节。表示其他更大的符号，可能需要3个字节或者4个字节，甚至更多。

这里就有两个严重的问题，第一个问题是，如何才能区别 Unicode 和 ASCII？计算机怎么知道三个字节表示一个符号，而不是分别表示三个符号呢？第二个问题是，我们已经知道，英文字母只用一个字节表示就够了，如果 Unicode 统一规定，每个符号用三个或四个字节表示，那么每个英文字母前都必然有二到三个字节是0，这对于存储来说是极大的浪费，文本文件的大小会因此大出二三倍，这是无法接受的。

它们造成的结果是：1) 出现了 Unicode 的多种存储方式，也就是说有许多种不同的二进制格式，可以用来表示 Unicode。2) Unicode 在很长一段时间内无法推广，直到互联网的出现。

## UTF-8

互联网的普及，强烈要求出现一种统一的编码方式。UTF-8 就是在互联网上使用最广的一种 Unicode 的实现方式。其他实现方式还包括 UTF-16（字符用两个字节或四个字节表示）和 UTF-32（字符用四个字节表示），不过在互联网上基本不用。重复一遍，这里的关系是，UTF-8 是 Unicode 的实现方式之一。

UTF-8 最大的一个特点，就是它是一种变长的编码方式。它可以使用1~4个字节表示一个符号，根据不同的符号而变化字节长度。

UTF-8 的编码规则很简单，只有二条：

1) 对于单字节的符号，字节的第一位设为0，后面7位为这个符号的 Unicode 码。因此对于英语字母，UTF-8 编码和 ASCII 码是相同的。

2) 对于n字节的符号（ $n > 1$ ），第一个字节的前n位都设为1，第n + 1位设为0，后面字节的前两位一律设为10。剩下的没有提及的二进制位，全部为这个符号的 Unicode 码。

下表总结了编码规则，字母x表示可用编码的位。

Unicode符号范围(十六进制)	UTF-8编码方式 (二进制)
0000 0000-0000 007F	0xxxxxxxx
0000 0080-0000 07FF	110xxxxx 10xxxxxx
0000 0800-0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000-0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

根据上表，解读 UTF-8 编码非常简单。如果一个字节的第一位是0，则这个字节单独就是一个字符；如果第一位是1，则连续有多少个1，就表示当前字符占用多少个字节。

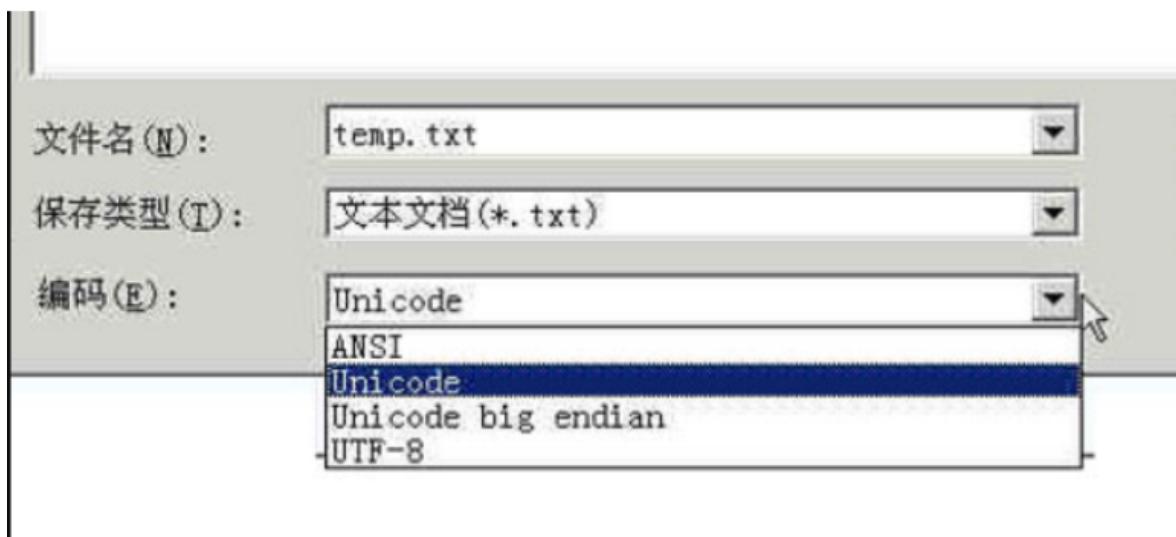
下面，还是以汉字严为例，演示如何实现 UTF-8 编码。

严的 Unicode 是4E25 (100111000100101)，根据上表，可以发现4E25处在第三行的范围内 (0000 0800 - 0000 FFFF)，因此严的 UTF-8 编码需要三个字节，即格式是1110xxxx 10xxxxxx 10xxxxxx。然后，从严的最后一个二进制位开始，依次从后向前填入格式中的x，多出的位补0。这样就得到了，严的 UTF-8 编码是11100100 10111000 10100101，转换成十六进制就是E4B8A5。

### Unicode 与 UTF-8 之间的转换

通过上一节的例子，可以看到严的 Unicode 码是4E25，UTF-8 编码是E4B8A5，两者是不一样的。它们之间的转换可以通过程序实现。

Windows 平台，有一个最简单的转化方法，就是使用内置的记事本小程序notepad.exe。打开文件后，点击文件菜单中的另存为命令，会跳出一个对话框，在最底部有一个编码的下拉条。



里面有四个选项：ANSI，Unicode，Unicode big endian和UTF-8。

- 1) ANSI是默认的编码方式。对于英文文件是ASCII编码，对于简体中文文件是GB2312编码（只针对 Windows 简体中文版，如果是繁体中文版会采用 Big5 码）。
- 2) Unicode编码这里指的是notepad.exe使用的 UCS-2 编码方式，即直接用两个字节存入字符的 Unicode 码，这个选项用的 little endian 格式。
- 3) Unicode big endian编码与上一个选项相对应。我在下一节会解释 little endian 和 big endian 的涵义。
- 4) UTF-8编码，也就是上一节谈到的编码方法。

选择完“编码方式”后，点击“保存”按钮，文件的编码方式就立刻转换好了。

### Little endian 和 Big endian

上一节已经提到，UCS-2 格式可以存储 Unicode 码（码点不超过0xFFFF）。以汉字严为例，Unicode 码是4E25，需要用两个字节存储，一个字节是4E，另一个字节是25。存储的时候，4E在前，

25在后，这就是 Big endian 方式；25在前，4E在后，这是 Little endian 方式。

这两个古怪的名称来自英国作家斯威夫特的《格列佛游记》。在该书中，小人国里爆发了内战，战争起因是人们争论，吃鸡蛋时究竟是从大头(Big-endian)敲开还是从小头(Little-endian)敲开。为了这件事情，前后爆发了六次战争，一个皇帝送了命，另一个皇帝丢了王位。

第一个字节在前，就是“大头方式”(Big endian)，第二个字节在前就是“小头方式”(Little endian)。

那么很自然的，就会出现一个问题：计算机怎么知道某一个文件到底采用哪一种方式编码？

Unicode 规范定义，每一个文件的最前面分别加入一个表示编码顺序的字符，这个字符的名字叫做“零宽度非换行空格”(zero width no-break space)，用FEFF表示。这正好是两个字节，而且FF比FE大1。

如果一个文本文件的头两个字节是FE FF，就表示该文件采用大头方式；如果头两个字节是FF FE，就表示该文件采用小头方式。

### 实例

下面，举一个实例。

打开“记事本”程序notepad.exe，新建一个文本文件，内容就是一个严字，依次采用ANSI，Unicode，Unicode big endian和UTF-8编码方式保存。

然后，用文本编辑软件UltraEdit 中的“十六进制功能”，观察该文件的内部编码方式。

- 1) ANSI：文件的编码就是两个字节D1 CF，这正是严的 GB2312 编码，这也暗示 GB2312 是采用大头方式存储的。
- 2) Unicode：编码是四个字节FF FE 25 4E，其中FF FE表明是小头方式存储，真正的编码是4E25。
- 3) Unicode big endian：编码是四个字节FE FF 4E 25，其中FE FF表明是大头方式存储。
- 4) UTF-8：编码是六个字节EF BB BF E4 B8 A5，前三个字节EF BB BF表示这是UTF-8编码，后三个E4B8A5就是严的具体编码，它的存储顺序与编码顺序是一致的。

## String详解

### 概述

String 类代表字符串。Java 程序中的所有字符串字面值（如 "abc"）都作为此类的实例实现。

字符串是常量；它们的值在创建之后不能更改。字符串缓冲区支持可变的字符串。因为 String 对象是不可变的，所以可以共享。

```
java.lang.String:  
    public final class String  
        implements java.io.Serializable, Comparable<String>, CharSequence {
```

### 源码分析

#### 1. 成员变量

```
    /** String的属性值 */  
    private final char value[];  
    /** The offset is the first index of the storage that is used. */  
    //数组被使用的开始位置/  
    private final int offset;  
    /** The count is the number of characters in the String. */  
    //String中元素的个数/  
    private final int count;
```

```
/** Cache the hash code for the string */
/String类型的hash值/
private int hash; // Default to 0
/** use serialVersionUID from JDK 1.0.2 for interoperability */
private static final long serialVersionUID = -6849794470754667710L;
private static final ObjectStreamField[] serialPersistentFields =
new ObjectStreamField[0];
```

从源码看出String底层使用一个字符数组来维护的。

成员变量可以知道String类的值是final类型的，不能被改变的，所以只要一个值改变就会生成一个新的String类型对象，存储String数据也不一定从数组的第0个元素开始的，而是从offset所指的元素开始。

## 2. String的构造方法

String()

初始化一个新创建的 String 对象，使其表示一个空字符序列。

String(byte[] bytes)

通过使用平台的默认字符集解码指定的 byte 数组，构造一个新的 String

String(byte[] bytes, Charset charset)

通过使用指定的 charset 解码指定的 byte 数组，构造一个新的 String。

String(byte[] bytes, int offset, int length)

通过使用平台的默认字符集解码指定的 byte 子数组，构造一个新的 String。

String(byte[] bytes, int offset, int length, Charset charset)

通过使用指定的 charset 解码指定的 byte 子数组，构造一个新的 String。

String(byte[] bytes, int offset, int length, String charsetName)

通过使用指定的字符集解码指定的 byte 子数组，构造一个新的 String。

String(byte[] bytes, String charsetName)

通过使用指定的 charset 解码指定的 byte 数组，构造一个新的 String。

String(char[] value)分配一个新的 String，使其表示字符数组参数中当前包含的字符序列。

String(char[] value, int offset, int count)

分配一个新的 String，它包含取自字符数组参数一个子数组的字符。

String(int[] codePoints, int offset, int count)

分配一个新的 String，它包含 Unicode 代码点数组参数一个子数组的字符。

String(String original)

初始化一个新创建的 String 对象，使其表示一个与参数相同的字符序列；换句话说，新创建的字符串是该参数字符串的副本。

String(StringBuffer buffer)

分配一个新的字符串，它包含字符串缓冲区参数中当前包含的字符序列。

String(StringBuilder builder)

分配一个新的字符串，它包含字符串生成器参数中当前包含的字符序列。

## 3. string重写了equals方法

```
public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = value.length;
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            while (n-- != 0) {
                if (v1[i] != v2[i])
```

```
    return false;
    i++;
}
return true;
}
}
return false;
}
```

## String不可变的好处

### 1. 便于实现String常量池

只有当字符串是不可变的，字符串池才有可能实现。字符串池的实现可以在运行时节约很多heap空间，因为不同的字符串变量都指向池中的同一个字符串。但如果字符串是可变的，那么String interning将不能实现(译者注：String interning是指对不同的字符串仅仅只保存一个，即不会保存多个相同的字符串。)，因为这样的话，如果变量改变了它的值，那么其它指向这个值的变量的值也会一起改变。

### 2. 避免网络安全问题

如果字符串是可变的，那么会引起很严重的安全问题。譬如，数据库的用户名、密码都是以字符串的形式传入来获得数据库的连接，或者在socket编程中，主机名和端口都是以字符串的形式传入。因为字符串是不可变的，所以它的值是不可改变的，否则黑客们可以钻到空子，改变字符串指向的对象的值，造成安全漏洞。

### 3. 使多线程安全

因为字符串是不可变的，所以是多线程安全的，同一个字符串实例可以被多个线程共享。这样便不用因为线程安全问题而使用同步。字符串自己便是线程安全的。

### 4. 避免本地安全性问题

类加载器要用到字符串，不可变性提供了安全性，以便正确的类被加载。譬如你想加载java.sql.Connection类，而这个值被改成了myhacked.Connection，那么会对你的数据库造成不可知的破坏。

### 5. 加快字符串处理速度

因为字符串是不可变的，所以在它创建的时候hashcode就被缓存了，不需要重新计算。这就使得字符串很适合作为Map中的键，字符串的处理速度要快过其它的键对象。这就是HashMap中的键往往都使用字符串。

## 字符串常量池

### 1、字符串常量池概述

#### 1) 常量池表 (Constant\_Pool table)

Class文件中存储所有常量（包括字符串）的table。

这是Class文件中的内容，还不是运行时的内容，不要理解它是个池子，其实就是Class文件中的字节码指令。

#### 2) 运行时常量池 (Runtime Constant Pool)

JVM内存中方法区的一部分，这是运行时的内容

这部分内容（绝大部分）是随着JVM运行时候，从常量池转化而来，每个Class对应一个运行时常量池

上一句中说绝大部分是因为：除了 Class中常量池内容，还可能包括动态生成并加入这里的內容

#### 3) 字符串常量池 (String Pool)

这部分也在方法区中，但与Runtime Constant Pool不是一个概念，String Pool是JVM实例全局共享的，全局只有一个

JVM规范要求进入这里的String实例叫“被驻留的interned string”，各个JVM可以有不同的实现，HotSpot是设置了一个哈希表StringTable来引用堆中的字符串实例，被引用就是被驻留。

### 2、享元模式

其实字符串常量池这个问题涉及到一个设计模式，叫“享元模式”，顾名思义 - - - > 共享元素模式，也就是说：一个系统中如果有多处用到了相同的一个元素，那么我们应该只存储一份此元素，而让所有地方都引用这一个元素

Java中String部分就是根据享元模式设计的，而那个存储元素的地方就叫做“字符串常量池 - String Pool”

### 3、详细分析

举例：

```
int x = 10;  
String y = "hello";
```

1)首先，10和"hello"会在经过javac（或者其他编译器）编译过后变为Class文件中constant\_pool table的内容

2)当我们的程序运行时，也就是说JVM运行时，每个Classconstant\_pool table中的内容会被加载到JVM内存中的方法区中各自Class的Runtime Constant Pool。

3)一个没有被String Pool包含的Runtime Constant Pool中的字符串（这里是"hello"）会被加入到String Pool中（HotSpot使用hashtable引用方式），步骤如下：

一是：在Java Heap中根据"hello"字面量create一个字符串对象（直接使用双引号声明出来的String对象会直接存储在常量池中）

二是：将字面量"hello"与字符串对象的引用在hashtable中关联起来，键 - 值形式是："hello" = 对象的引用地址。

另外来说，当一个新的字符串出现在Runtime Constant Pool中时怎么判断需不需要在Java Heap中创建新对象呢？

策略是这样：会先去根据equals来比较Runtime Constant Pool中的这个字符串是否和String Pool中某一个是相等的（也就是找是否存在），如果有那么就不创建，直接使用其引用；反之，如上3如此，就实现了享元模式，提高的内存利用效率。

举例：

使用String s = new String("hello");会创建几个对象  
会创建2个对象

首先，出现了字面量"hello"，那么去String Pool中查找是否有相同字符串存在，因为程序就这一行代码所以肯定没有，那么就在Java Heap中用字面量"hello"首先创建1个String对象。

接着，new String("hello")，关键字new又在Java Heap中创建了1个对象，然后调用接收String参数的构造器进行了初始化。最终s的引用是这个String对象。

## String.intern()

```
public native String intern();
```

这个方法是一个native的方法，但注释写的非常明了。“如果常量池中存在当前字符串，就会直接返回当前字符串。如果常量池中没有此字符串，会将此字符串放入常量池中后，再返回”。

## 实例

一个String的题目：

```
String s1 = "ABCD";  
String s2 = "A" + "B" + "C" + "D";  
String s3 = "AB" + "CD";  
String s4 = new String("ABCD");  
String s = "AB";  
String s5 = s + "CD";  
String s6 = getSt() + "CD";  
final String ss = "AB";  
String s7 = ss + "CD";  
public static String getSt()  
return "AB";  
}
```

其中s1,s2,s3,s7相等，其他都不等

分析：

s1,s2,s3在编译的时候都被优化为"ABCD"了，所以相等。s5,s6中有变量，都是运行时使用new StringBuilder连接（如果有变量，那么会调用StringBuilder，最后调用Sb的toString），s7是final修饰的常量，也会在编译器被优化

## Integer详解

### 1.介绍

关于Integer和int在面试的时候出现的频率很高。而我们所熟知的是Integer是int 的包装类型，int的初始值为0，Integer的初始值为null，这是基本都知道的。至于Integer的自动装箱和拆箱，以及Integer的缓存等小细节需要深入思考

### 2.自动装箱与拆箱

#### 1. 自动装箱

```
1 | public class CaseTest {  
2 |  
3 |     public static void main(String[] args) {  
4 |         Integer i = 100;  
5 |     }  
6 | }
```

#### 2. 自动拆箱

```
1 | public class CaseTest {  
2 |  
3 |     public static void main(String[] args) {  
4 |         Integer i = 100;  
5 |         int j = i;  
6 |     }  
7 | }
```

### 3.Integer的缓存

Integer定义了一个静态内部类IntegerCache作为缓存

```
1 | private static class IntegerCache {  
2 |     //常量最小值-128  
3 |     static final int low = -128;  
4 |     //常量最大值  
5 |     static final int high;  
6 |     //Integer缓存数组  
7 |     static final Integer cache[];  
8 |  
9 |     static {  
10 |         //初始化h变量为127  
11 |         int h = 127;  
12 |         String integerCacheHighPropValue =  
13 |             sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");  
14 |         if (integerCacheHighPropValue != null) {  
15 |             int i = parseInt(integerCacheHighPropValue);  
16 |             //说明i最小可以取到127，规定了i的下限，所以i>=127  
17 |             i = Math.max(i, 127);  
18 |             //由于i>=127并且Integer.MAX_VALUE - (-low)>128，因此可以得到h>=127  
19 |             h = Math.min(i, Integer.MAX_VALUE - (-low));  
20 |         }  
21 |         high = h;  
22 |         //初始化Integer缓存数组  
23 |         cache = new Integer[(high - low) + 1];  
24 |         //数组中的初始值是-128  
25 |         int j = low;
```

```

27     //循环为数组赋值，数组0位是-128，最大是127（临界值）
28     for(int k = 0; k < cache.length; k++)
29         cache[k] = new Integer(j++);
30     }
31
32     private IntegerCache() {}
33 }
34 /**
35 * 返回一个表示指定的 int 值的 Integer 实例。
36 * 如果不需要新的 Integer 实例，则通常应优先使用该方法，而不是构造方法 Integer(int)
37 * 因为该方法有可能通过缓存经常请求的值而显著提高空间和时间性能
38 */
39 public static Integer valueOf(int i) {
40     assert IntegerCache.high >= 127;
41     //当-128<=i<=127时，直接从缓存数组中取出值
42     if (i >= IntegerCache.low && i <= IntegerCache.high)
43         return IntegerCache.cache[i + (-IntegerCache.low)];
44     //否则返回新的Integer对象
45     return new Integer(i);
}

```

例子：

```

1 public static void main(String[] args) {
2     //标准的装箱过程
3     Integer i =new Integer(100);
4     Integer i1 =100;
5     Integer i2 =100;
6     Integer i3 =128;
7     Integer i4 =128;
8     System.out.println("i==i1 :" +(i==i1));
9     System.out.println("i1==i2 :" +(i1==i2));
10    System.out.println("i3==i4 :" +(i3==i4));
11 }

```

说明：

- (1) `i == i1`：由于`i`本身是`new Integer`的方式在堆中单独开辟空间，`i1`是`-128 <= i1 <= 127`，所以`i1`是从缓存中取出数据的。而缓存的地址和`new Integer`单独开辟空间对应的地址不同，返回`false`。
- (2) `i1 == i2`：`i1`是`-128 <= i1 <= 127`，所以`i1`是从缓存中取出数据的。`i2`是`-128 <= i2 <= 127`，所以`i2`也是从缓存中取出数据的。所以`i1`和`i2`对应的是同一个缓存地址，返回`true`。
- (3) `i3 == i4`：`i3 > 127`并且`i4 > 127`,所以`i3`和`i4`分别是`new Integer`这种单独开辟空间，地址不同，返回`false`。

## java内部类

链接：<https://www.cnblogs.com/dolphin0520/p/3811445.html>

##关于hashCode重写

正确书写hashCode的办法：

【原则】按照`equals()`中比较两个对象是否一致的条件用到的属性来重写`hashCode()`。

{1}常用的办法就是利用涉及到的属性进行线性组合。

{2}线性组合过程中涉及到的组合系数自定义即可。

注意，拼接之后的数值不能超过整形的表达范围。

{3}公式：属性1的int形式+ C1属性2的int形式+ C2属性3的int形式+ ...

【技巧】当属性是引用类型的时候，如果已经重写过`hashCode()`，那么这个引用属性的int形式就是直接调用属性已有的`hashCode`值。

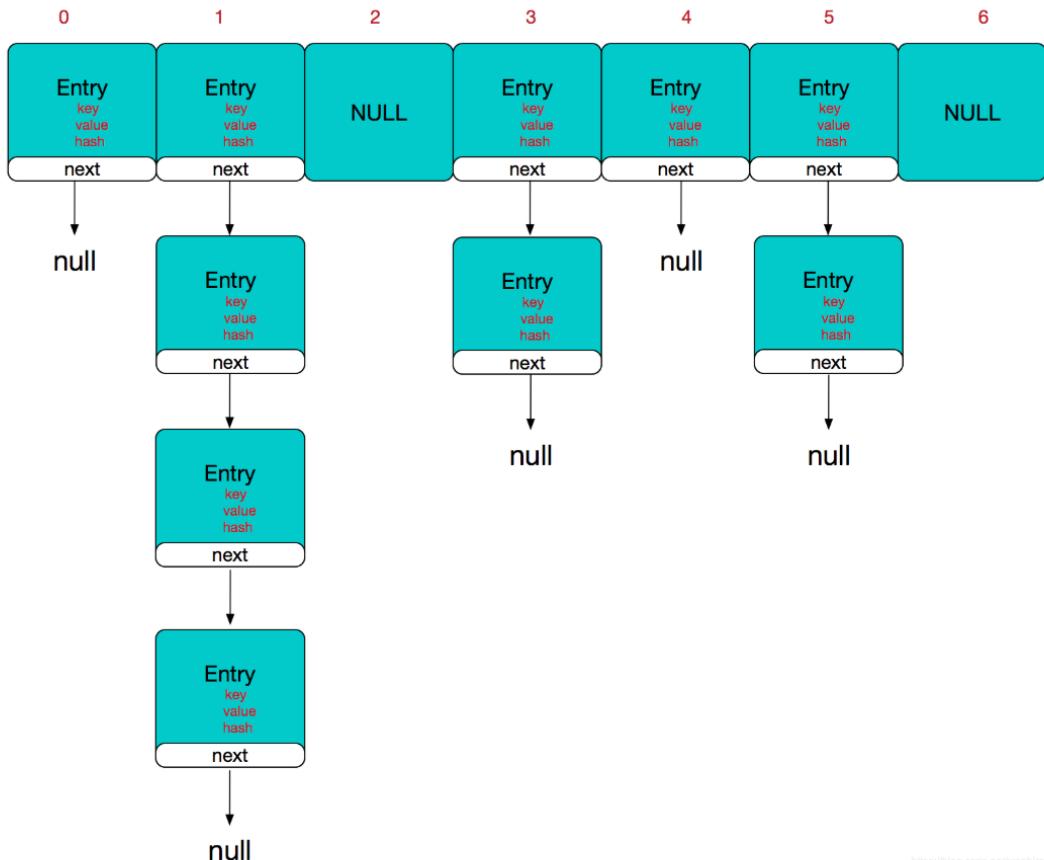
最典型的就是这个属性是字符串类型的，`String`类型已经重写了`hashCode()`方法，所以直接拿来使用即可。

如：大表弟的两个字符串拼接(`deviceId.hashCode() >> 2`) + (`courseScheduleId.hashCode() >> 2`)

# HashMap

## 结构

结构上由数组与链表构成，JDK 8后由数组+链表+红黑树构成。JDK8采用尾插法，之前采用头插法。结构图如下：



<http://blog.csdn.net/woshimao101>

## 源码

### 1.重要字段

```
/**  
 * The number of key-value mappings contained in this map.  
 * 即实际存储的键值对个数  
 */  
transient int size;  
/**  
 * The number of times this HashMap has been structurally modified  
 * Structural modifications are those that change the number of mappings in  
 * the HashMap or otherwise modify its internal structure (e.g.,  
 * rehash). This field is used to make iterators on Collection-views of  
 * the HashMap fail-fast. (See ConcurrentModificationException).  
 * HashMap被改变的次数，由于HashMap非线程安全，在对HashMap进行迭代时，  
 * 如果期间其他线程的参与导致HashMap的结构发生了变化了（比如put, remove等操作），  
 * 需要抛出异常ConcurrentModificationException  
 */  
transient int modCount;  
/**  
 * The next size value at which to resize (capacity * load factor).  
 */
```

```

        * 当size到达此值时，将触发扩容
        * @serial
        */
int threshold;
/***
     * The load factor for the hash table.
     *
     * 加载因子
     */
final float loadFactor;
/***
     * The table, initialized on first use, and resized as
     * necessary. When allocated, length is always a power of two.
     * (We also tolerate length zero in some operations to allow
     * bootstrapping mechanics that are currently not needed.)
     * 内部数组，数组的长度就是initialCapacity.且initialCapacity会取最接近的那个2的幂的数
     * 通过tableSizeFor(int cap)方法。比如new HashMap<>(10);其实初始的table的长度是16
     */
transient Node<K,V>[] table;

//创建 HashMap 时未指定初始容量情况下的默认容量 16
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;

//HashMap 的最大容量，2的30幂
static final int MAXIMUM_CAPACITY = 1 << 30;

//HashMap 默认的装载因子，当 HashMap 中元素数量超过 容量*装载因子 时，进行
resize() 操作
static final float DEFAULT_LOAD_FACTOR = 0.75f;

//用来确定何时将解决 hash 冲突的链表转变为红黑树
static final int TREEIFY_THRESHOLD = 8;

// 用来确定何时将解决 hash 冲突的红黑树转变为链表
static final int UNTREEIFY_THRESHOLD = 6;

/* 当需要将解决 hash 冲突的链表转变为红黑树时，需要判断下此时数组容量，若是由于数组容量太
小（小于 MIN_TREEIFY_CAPACITY ）导致的 hash 冲突太多，则不进行链表转变为红黑树操作，转为
利用 resize() 函数对 hashMap 扩容 */
static final int MIN_TREEIFY_CAPACITY = 64;

```

## 2.重要内部类

```

static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;

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

    public final K getKey()           { return key; }

```

```

        public final V getValue() { return value; }

        public final String toString() { return key + "=" + value; }

        public final int hashCode() {
            return Objects.hashCode(key) ^ Objects.hashCode(value);
        }

        public final V setValue(V newValue) {
            V oldValue = value;
            value = newValue;
            return oldValue;
        }

        public final boolean equals(Object o) {
            if (o == this)
                return true;
            if (o instanceof Map.Entry) {
                Map.Entry<?, ?> e = (Map.Entry<?, ?>)o;
                if (Objects.equals(key, e.getKey()) &&
                    Objects.equals(value, e.getValue()))
                    return true;
            }
            return false;
        }

    }

// 截取部分 LinkedHashMap 代码
public class LinkedHashMap<K,V> {
    static class Entry<K,V> extends HashMap.Node<K,V> {
        Entry<K,V> before, after;
        Entry(int hash, K key, V value, Node<K,V> next) {
            super(hash, key, value, next);
        }
    }
}

// TreeNode<K,V> 继承 LinkedHashMap.Entry<K,V>, 用来实现红黑树相关的存储结构
static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
    TreeNode<K,V> parent; // 存储当前节点的父节点
    TreeNode<K,V> left; // 存储当前节点的左孩子
    TreeNode<K,V> right; // 存储当前节点的右孩子
    TreeNode<K,V> prev; // 存储当前节点的前一个节点
    boolean red; // 存储当前节点的颜色（红、黑）
    TreeNode(int hash, K key, V val, Node<K,V> next) {
        super(hash, key, val, next);
    }
}

```

### 3.重要构造方法

```

public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
                                         initialCapacity);
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))

```

```

        throw new IllegalArgumentException("Illegal load factor: " +
                                           loadFactor);
        this.loadFactor = loadFactor;
        this.threshold = tableSizeFor(initialCapacity);
    }

    /**
     * Returns a power of two size for the given target capacity.
     */
    static final int tableSizeFor(int cap) {
        int n = cap - 1;
        n |= n >>> 1;
        n |= n >>> 2;
        n |= n >>> 4;
        n |= n >>> 8;
        n |= n >>> 16;
        return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
    }
}

```

**注意：**构造方法并没有初始化table，其实table初始化在put操作中。tableSizeFor用于向上取最接近的2的幂。

cap = 10;	0000 1001
n = cap - 1; //9	
n  = n >>> 1;	0000 1001 或 0000 0100
	右移1位
	0000 1101
n  = n >>> 2;	0000 1101 或 0000 0011
	右移2位
	0000 1111
n  = n >>> 4;	0000 1111 或 0000 0000
	右移4位
	0000 1111
n  = n >>> 8;	0000 1111 或 0000 0000
	右移8位 对这个数据没什么作用
	0000 1111
n  = n >>> 16;	0000 1111 或 0000 0000
	右移16位 对这个数据没什么作用
	0000 1111
n = n + 1;	0001 0000
	得到结果 $2^4=16$
	<a href="#">查看原图</a>

让cap-1再赋值给n的目的是另找到的目标值大于或等于原值。例如二进制1000，十进制数值为8。如果不对其减1而直接操作，将得到答案10000，即16。显然不是结果。减1后二进制为111，再进行操作则会得到原来的数值1000，即8。

HashMap里的MAXIMUM\_CAPACITY是2的30幂。我结合tableSizeFor()的实现，猜测设置原因如下：int的正数最大可达2的31幂-1，而没办法取到2的31幂，所以容量也无法达到2的31幂。又需要让容量满足2的幂次。所以设置为2的30幂。

#### 4.重要方法

- get(Object key):其中通过k的hashCode高16位与hashCode低16位与或得到hash。而数组下标是通过数组长度减去1后和hash进行&运算得到。即getNode里的 (n-1) & hash

```
static final int hash(object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}

public v get(object key) {
    Node<K,V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}

final Node<K,V> getNode(int hash, object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        if ((e = first.next) != null) {
            if (first instanceof TreeNode)
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
                } while ((e = e.next) != null);
            }
        }
    return null;
}
```

- put方法

```
public v put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}

//onlyIfAbsent为true表示新值不会覆盖旧值，不过旧值是null依旧会覆盖
final v putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
```



```

/*要进入下面这个else if,代表有以下几个含义:
1、当前节点与待插入节点 key 不同, hash 值相同
2、k 是不可比较的, 即 k 并未实现 comparable<K> 接口
(若 k 实现了comparable<K> 接口, comparableClassFor (k) 返回的是 k 的
class,而不是null)
或者 compareComparables(kc, k, pk) 返回值为 0
(pk 为空 或者 按照 k.compareTo(pk) 返回值为 0,
返回值为 0 可能是由于 k 的compareTo 方法实现不当引起的, compareTo 判定相
等, 而上个 else
if 中 equals 判定不等)*/
else if ((kc == null &&
          (kc = comparableClassFor(k)) == null) ||
          (dir = compareComparables(kc, k, pk)) == 0) {
    //在以当前节点为根的整个树上搜索是否存在待插入节点(只会搜索一次)
    if (!searched) {
        TreeNode<K,V> q, ch;
        searched = true;
        if (((ch = p.left) != null &&
             (q = ch.find(h, k, kc)) != null) ||
            ((ch = p.right) != null &&
             (q = ch.find(h, k, kc)) != null))
            //若树中存在待插入节点, 直接返回
            return q;
    }
    dir = tieBreakOrder(k, pk);
}

TreeNode<K,V> xp = p;
if ((p = (dir <= 0) ? p.left : p.right) == null) {
    //找到了待插入的位置, xp 为待插入节点的父节点
    //注意TreeNode节点中既存在树状关系, 也存在链式关系, 并且是双端链表
    Node<K,V> xpn = xp.next;
    TreeNode<K,V> x = map.newTreeNode(h, k, v, xpn);
    if (dir <= 0)
        xp.left = x;
    else
        xp.right = x;
    xp.next = x;
    x.parent = x.prev = xp;
    if (xp != null)
        ((TreeNode<K,V>)xp).prev = x;
    //插入节点后进行二叉树的平衡操作
    moveRootToFront(tab, balanceInsertion(root, x));
    return null;
}
}

static int tieBreakOrder(Object a, Object b) {
    int d;
    //System.identityHashCode()实际是利用对象 a,b 的内存地址进行比较
    if (a == null || b == null ||
        (d = a.getClass().getName().
        compareTo(b.getClass().getName())) == 0)
        d = (System.identityHashCode(a) <= System.identityHashCode(b) ?
-1 : 1);
    return d;
}

```

```

final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                  oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    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;
    @SuppressWarnings({"rawtypes","unchecked"})
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
    table = newTab;
    if (oldTab != null) {
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null;
                if (e.next == null)
                    newTab[e.hash & (newCap - 1)] = e;
                else if (e instanceof TreeNode)
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                else { // preserve order
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    do {
                        next = e.next;
                        // 对应下图中的那个多出来的1bit位，为0表示index不变，否则加
oldCap
                        // 扩容前是oldCap-1（比如16-1=15，就是1111）和hash的&，现
在要看多出来的1bit，
                        // 就需要1111+1，其实就变回oldCap了，因此用e.hash &
oldCap判断多出来的1bit
                        if ((e.hash & oldCap) == 0) {
                            if (loTail == null)
                                loHead = e;
                            else
                                loTail.next = e;
                            loTail = e;
                        }
                    }
                }
            }
        }
    }
}

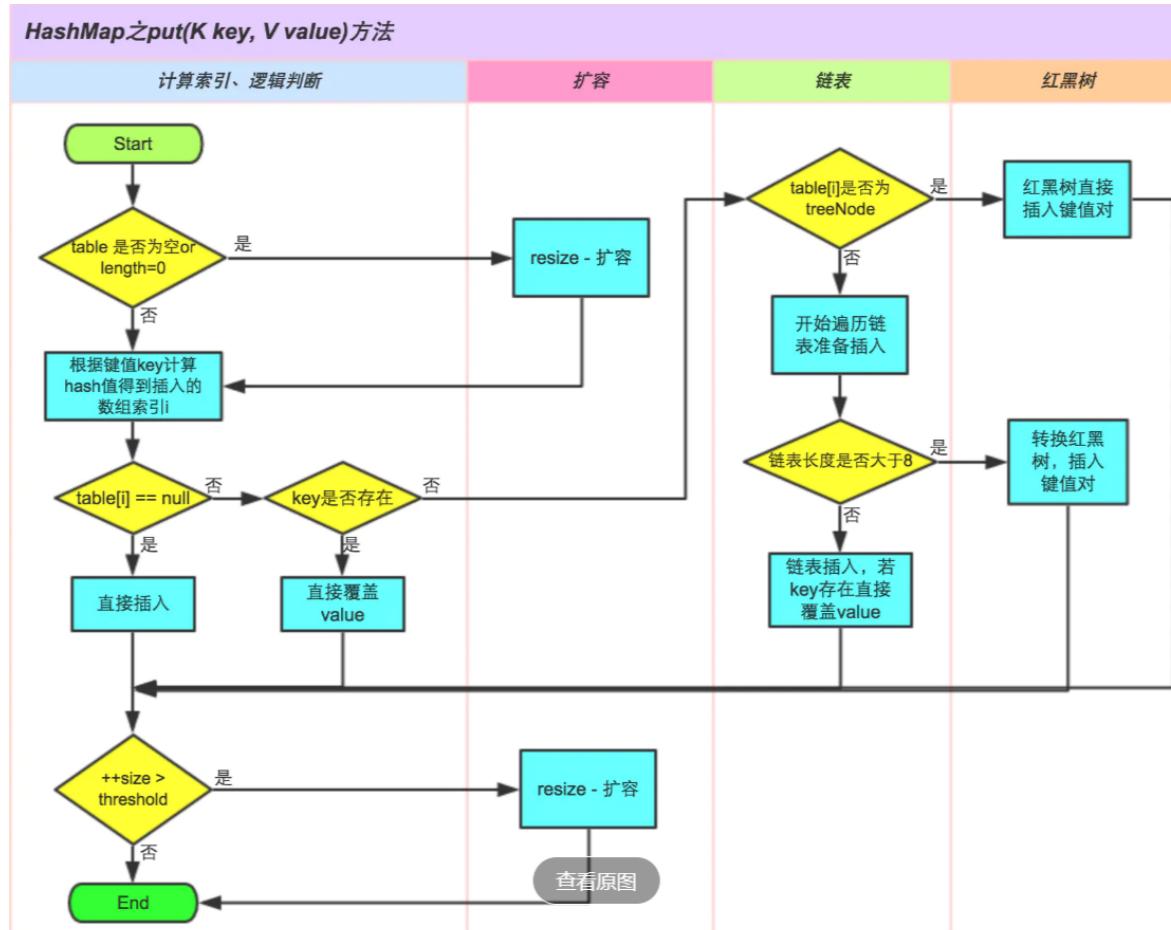
```

```

        else {
            if (hiTail == null)
                hiHead = e;
            else
                hiTail.next = e;
            hiTail = e;
        }
    } while ((e = next) != null);
if (loTail != null) {
    loTail.next = null;
    newTab[j] = loHead;
}
if (hiTail != null) {
    hiTail.next = null;
    newTab[j + oldCap] = hiHead;
}
}
}
}
return newTab;
}
}

```

## 5.put方法过程



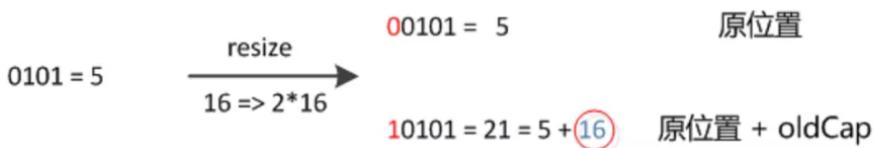
## 6.resize原理

`resize()` 方法中比较重要的是链表和红黑树的 rehash 操作，先来说下 rehash 的实现原理：

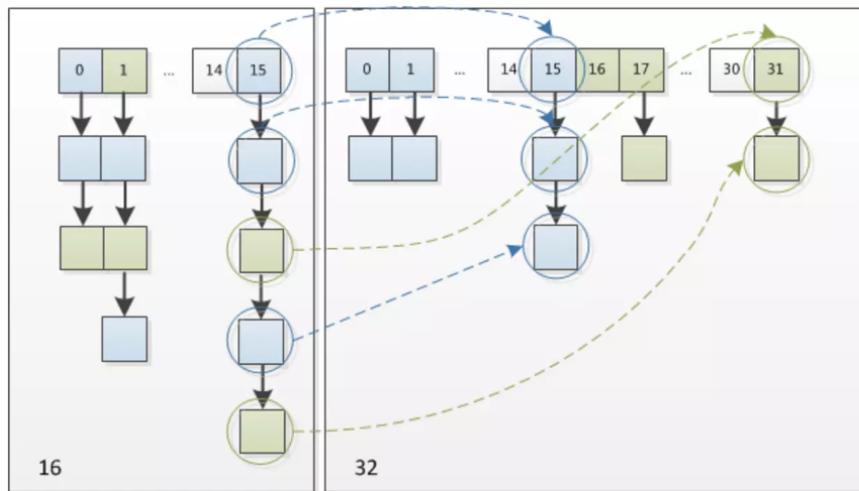
我们在扩容的时候，一般是把长度扩为原来2倍，所以，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。看下图可以明白这句话的意思， $n$ 为table的长度，图(a)表示扩容前的key1和key2两种key确定索引位置的示例，图(b)表示扩容后key1和key2两种key确定索引位置的示例，其中hash1是key1对应的哈希与高位运算结果。



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：



因此，我们在扩充HashMap的时候，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”，可以看看下图为16扩充为32的resize示意图：



这个算法很巧妙，既省去了重新计算hash值的时间，而且同时，由于新增的1bit是0还是1可以认为是随机的，因此resize的过程，均匀的把之前的冲突的节点分散到新的槽中了。

1.7是全部rehash，采用头插法；在JDK1.8中发生hashmap扩容时，遍历hashmap每个bucket里的链表，每个链表可能会被拆分成两个链表，不需要移动的元素置入loHead为首的链表，需要移动的元素置入hiHead为首的链表，然后分别分配给老的buket和新的buket

## 线程不安全分析

1.put的时候导致的多线程数据不一致

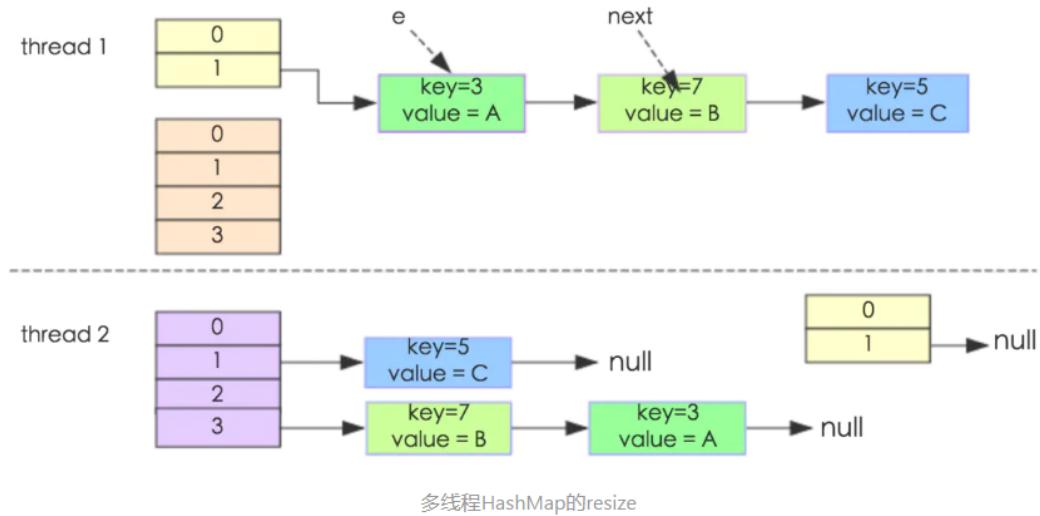
比如有两个线程A和B，首先A希望插入一个key-value对到HashMap中，首先计算记录所要落到的桶的索引坐标，然后获取到该桶里面的链表头结点，此时线程A的时间片用完了，而此时线程B被调度得以执行，和线程A一样执行，只不过线程B成功将记录插到了桶里面，假设线程A插入的记录计算出来的桶索引和线程B要插入的记录计算出来的桶索引是一样的，那么当线程B成功插入之后，线程A再次被调度运行时，它依然持有过期的链表头但是它对此一无所知，以至于它认为它应该这样做，如此一来就覆盖了线程B插入的记录，这样线程B插入的记录就凭空消失了，造成了数据不一致的行为。

2.get操作可能因为resize而引起死循环 (cpu100%)

就是两个线程同时resize的时候，A线程时间片用完的中间迁移状态，线程Bresize完毕，线程A继续执行，有可能造成某个桶下面的两个节点的next相互引用，此时执行get方法的key如果正好在这个桶的位置，且和这两个节点不相等，就会在遍历链表的时候造成死循环。

ps:JDK 7才会有死循环的问题（头插法导致节点顺序倒置，产生循环），JDK 8采用尾插法已经解决。JDK 8会有数据覆盖的不安全情况，比如A,B两个线程，put操作**不同的key**落入了相同的桶中，A刚执行完成`tab[i] = newNode(hash, key, value, null);`就让出来时间片，B顺利完成所有操作，则本应有两个节点的这个桶，只剩下B插入的节点。

### 3. 死循环过程如下



原数组table[0],table[1]。两个线程扩容后，变成两个newtable，都是4个位置。

扩容过程是，先存下当前节点的next，然后将当前节点放入新桶，同时新桶的第一个元素作为当前节点的next，再继续遍历。（就是开始处理之前存下的next）。

首先，thread1当前节点是3A，把7B先存下来作为下一个要处理的节点。此时，3A还没有插入新位置，时间片没了。

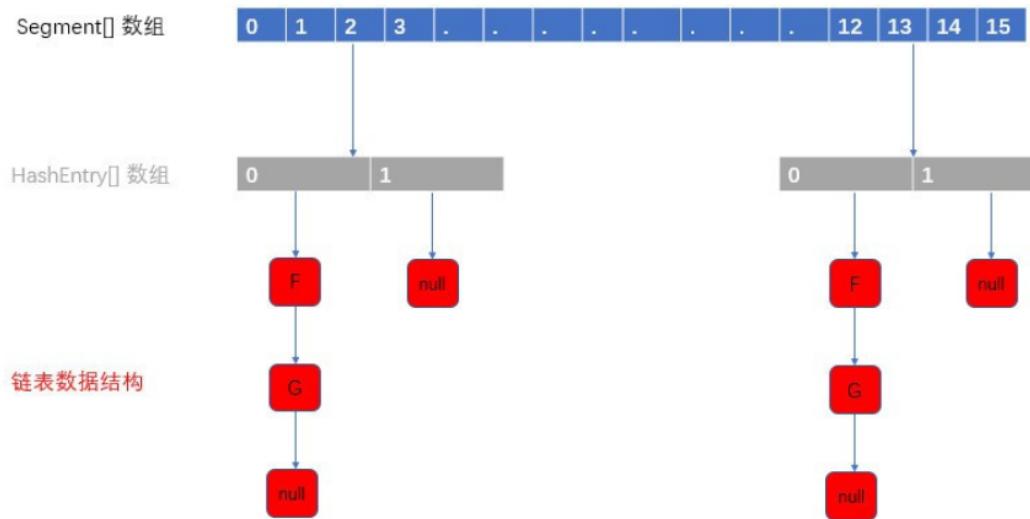
thread2一步到位执行完了，如上图。thread1继续执行，把3A插入新桶，并且next置为null（新桶没元素，自己作为第一个），插入完成后，拿出存的那个next，也就是7B继续处理。thread1处理7B,拿出7B的next存下（由于thread2处理过了，所以存的又是3A），将7B插入新桶，发现3A占据了新桶第一个元素，就把7B的next置为3A（其实next本来就是3A了，是thread2处理的）。插入完成后，thread1取出存的next（就是3A），拿出3A的next存下（是null了，被thread2处理的），准备将3A插入新桶第一个元素（此时是7B了），3A的next将被置为7B，循环产生，因为7B的next已经是3A了。结果thread1里产生了3A和7B互相引用。归根结底是头插法导致的顺序反向，导致循环。

## ConCurrentHashMap

### JDK7 (分段锁)

ConcurrentHashMap是由Segments数组结构和HashEntry数组结构组成.Segment是一种可重入锁(ReentrantLock),在ConcurrentHashMap里扮演锁的角色;HashEntry则用于存储键值对数据.一个ConcurrentHashMap里包含一个Segment组.Segment的结构和HashMap类似,是一种数组加链表的结构.一个Segment里包含一个HashEntry数组,每个HashEntry是一个链表结构的元素,每个Segment守护者一个HashEntry数组里面的元素,当对HashEntry数组的数据进行修改时,必须先获得与它对应的Segment锁,如下图所示.

### ConcurrentHashMap结构 1.7



## 1.重要字段

```

/**
 * 默认的初始容量 16
 */
static final int DEFAULT_INITIAL_CAPACITY = 16;
/**
 * 默认的负载因子
 */
static final float DEFAULT_LOAD_FACTOR = 0.75f;
/**
 * 默认的并发数量,会影响segments数组的长度(初始化后不能修改)
 */
static final int DEFAULT_CONCURRENCY_LEVEL = 16;
/**
 * 最大容量,构造ConcurrentHashMap时指定的值超过,就用该值替换
 * ConcurrentHashMap大小必须是2^n,且小于等于2^30
 */
static final int MAXIMUM_CAPACITY = 1 << 30;
/**
 * 每个segment中table数组的长度,必须是2^n,至少为2
 */
static final int MIN_SEGMENT_TABLE_CAPACITY = 2;
/**
 * 允许最大segment数量,用于限定concurrencyLevel的边界,必须是2^n
 */
static final int MAX_SEGMENTS = 1 << 16;
/**
 * 非锁定情况下调用size和contains方法的重试次数,避免由于table连续被修改导致无限重试
 */
static final int RETRIES_BEFORE_LOCK = 2;
/**
 * 用于segment的掩码值,用于与hash的高位进行取&
 */
final int segmentMask;
/**
 * 用于算segment位置时,hash参与运算的位数
 */
final int segmentsShift;
/**
 */

```

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

## 2.重要内部类

```
static final class HashEntry<K,V> {  
    // hash值  
    final int hash;  
    // key  
    final K key;  
    // 保证内存可见性,每次从内存中获取  
    volatile V value;  
    volatile HashEntry<K,V> next;  
  
    HashEntry(int hash, K key, V value, HashEntry<K,V> next) {  
        this.hash = hash;  
        this.key = key;  
        this.value = value;  
        this.next = next;  
    }  
  
    /**  
     * 使用volatile语义写入next,保证可见性  
     */  
    final void setNext(HashEntry<K,V> n) {  
        UNSAFE.putOrderedObject(this, nextOffset, n);  
    }  
}  
  
static final class Segment<K,V> extends ReentrantLock implements Serializable {  
    private static final long serialVersionUID = 2249069246763182397L;  
  
    /**  
     * 对segment加锁时,在阻塞之前自旋的次数  
     */  
    static final int MAX_SCAN_RETRIES =  
        Runtime.getRuntime().availableProcessors() > 1 ? 64 : 1;  
  
    /**  
     * 每个segment的HashEntry table数组,访问数组元素可以通过entryAt/setEntryAt提供的  
     * volatile语义来完成  
     * volatile保证可见性  
     */  
    transient volatile HashEntry<K,V>[] table;  
  
    /**  
     * 元素的数量,只能在锁中或者其他保证volatile可见性之间进行访问  
     */  
    transient int count;  
  
    /**  
     * 当前segment中可变操作发生的次数,put,remove等,可能会溢出32位  
     * 它为chm isEmpty() 和size()方法中的稳定性检查提供了足够的准确性.  
     * 只能在锁中或其他volatile读保证可见性之间进行访问  
     */
```

```

transient int modCount;

/**
 * 当table大小超过阈值时,对table进行扩容,值为(int)(capacity *loadFactor)
 */
transient int threshold;

/**
 * 负载因子
 */
final float loadFactor;

/**
 * 构造方法
 */
Segment(float lf, int threshold, HashEntry<K,V>[] tab) {
    this.loadFactor = lf;
    this.threshold = threshold;
    this.table = tab;
}

```

### 3.构造方法

```

/**
 * ConcurrentHashMap 构造方法
 * @param initialCapacity 初始化容量
 * @param loadFactor 负载因子
 * @param concurrencyLevel 并发segment,segments数组的长度
 */
public ConcurrentHashMap(int initialCapacity,
                      float loadFactor, int concurrencyLevel) {
    if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)
        throw new IllegalArgumentException();
    // 大于最大segments容量,取最大容量
    if (concurrencyLevel > MAX_SEGMENTS)
        concurrencyLevel = MAX_SEGMENTS;
    // Find power-of-two sizes best matching arguments
    // 2^sshift = ssize 例如:sshift = 4,ssize = 16
    // 根据concurrencyLevel计算出ssize为segments数组的长度
    int sshift = 0;
    int ssize = 1;
    while (ssize < concurrencyLevel) { // 第一次 满足
        ++sshift; // 第一次 1
        ssize <= 1; // 第一次 ssize = ssize << 1 (1 * 2^1)
    }
    // segmentsShift和segmentMask的定义
    this.segmentsShift = 32 - sshift; // 用于计算hash参与运算位数
    this.segmentMask = ssize - 1; // segments位置范围
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    // 计算每个segment中table的容量
    int c = initialCapacity / ssize;
    if (c * ssize < initialCapacity)
        ++c;
    // HashEntry[]默认 容量
    int cap = MIN_SEGMENT_TABLE_CAPACITY;
    // 确保cap是2^n
}

```

```

        while (cap < c)
            cap <= 1;
        // create segments and segments[0]
        // 创建segments并初始化第一个segment数组,其余的segment延迟初始化
        Segment<K,V> s0 =
            new Segment<K,V>(loadFactor, (int)(cap * loadFactor),
                (HashEntry<K,V>[])new HashEntry[cap]);
        Segment<K,V>[] ss = (Segment<K,V>[])new Segment[ssize];
        UNSAFE.putOrderedObject(ss, SBASE, s0); // ordered write of segments[0]
        this.segments = ss;
    }
}

```

## 4.重要方法

```

// 一些UNSAFE方法
// HashEntry
/**
 * 使用volatile语义写入next,保证可见性
 */
final void setNext(HashEntry<K,V> n) {
    UNSAFE.putOrderedObject(this, nextOffset, n);
}

/**
 * 获取给定table的第i个元素,使用volatile读语义
 */
static final <K,V> HashEntry<K,V> entryAt(HashEntry<K,V>[] tab, int i) {
    return (tab == null) ? null :
        (HashEntry<K,V>) UNSAFE.getObjectVolatile(
            tab, ((long)i << TSHIFT) + TBASE);
}

/**
 * 设置给定的table的第i个元素,使用volatile写语义
 */
static final <K,V> void setEntryAt(HashEntry<K,V>[] tab, int i,
        HashEntry<K,V> e) {
    UNSAFE.putOrderedObject(tab, ((long)i << TSHIFT) + TBASE, e);
}

```

## 5.put过程

### 执行流程分析

- (1) map的put方法就做了三件事情,找出segments的位置;判断当前位置有没有初始化,没有就调用ensureSegment()方法初始化,然后调用segment的put方法.
- (2) segment的put方法,获取当前segment的锁,成功接着执行,失败调用scanAndLockForPut方法自旋获取锁,成功后也是接着往下执行.
- (3) 通过hash计算出位置,获取节点,找出相同的key和hash替换value,返回.没有找到相同的,设置找出的节点为当前创建节点的next节点,设置创建节点前,判断是否需要扩容,需要调用扩容方法rehash();不需要,设置节点,返回,释放锁.

```

/**
 * map的put方法,定位segment
 */
public V put(K key, V value) {
    Segment<K,V> s;

```

```

// value不能为空
if (value == null)
    throw new NullPointerException();
// 获取hash
int hash = hash(key);
// 定位segments 数组的位置
int j = (hash >>> segmentsShift) & segmentMask;
// 获取这个segment
if ((s = (Segment<K,V>)UNSAFE.getObject           // nonvolatile; recheck
      (segments, (j << SSHIFT) + SBASE)) == null) // in ensureSegment
    // 为null 初始化当前位置的segment
    s = ensureSegment(j);
return s.put(key, hash, value, false);
}

/**
 * put到table方法
 */
final V put(K key, int hash, V value, boolean onlyIfAbsent) {
    // 是否获取锁,失败自旋获取锁(直到成功)
    HashEntry<K,V> node = tryLock() ? null :
        scanAndLockForPut(key, hash, value);
    V oldValue;
    try {
        HashEntry<K,V>[] tab = table;
        // 定义位置
        int index = (tab.length - 1) & hash;
        // 获取第一个桶的第一个元素
        // entryAt 底层调用getObjectVolatile 具有volatile读语义
        HashEntry<K,V> first = entryAt(tab, index);
        for (HashEntry<K,V> e = first;;) {
            if (e != null) { // 证明链式结构有数据 遍历节点数据替换,直到e=null
                K k;
                if ((k = e.key) == key ||
                    (e.hash == hash && key.equals(k))) { // 找到了相同的
key
                    oldValue = e.value;
                    if (!onlyIfAbsent) { // 默认值false
                        e.value = value; // 替换value
                        ++modCount;
                    }
                    break; // 结束循环
                }
                e = e.next;
            }
        else { // e=null (1) 之前没有数据 (2) 没有找到替换的元素
            // node是否为空,这个获取锁的是有关系的
            // (1) node不为null,设置node的next为first
            // (2) node为null,创建头节点,指定next为first
            if (node != null)
                // 底层使用 putOrderedObject 方法 具有volatile写语义
                node.setNext(first);
            else
                node = new HashEntry<K,V>(hash, key, value, first);
            int c = count + 1;
            // 扩容条件 (1)entry数量大于阈值 (2) 当前table的数量小于最大容量
满足以上条件就扩容
            if (c > threshold && tab.length < MAXIMUM_CAPACITY)
                // 扩容方法,方法里面具体讲

```

```

        rehash(node);
    } else
        // 给table的index位置设置为node,
        // node为头结点,原来的头结点first为node的next节点
        // 底层也是调用的 putOrderedObject 方法 具有volatile写语义
        setEntryAt(tab, index, node);
        ++modCount;
        count = c;
        oldvalue = null;
        break;
    }
}
} finally {
    unlock();
}
return oldvalue;
}

```

解释下(hash >>> segmentShift) & segmentMask定位segment位置(个人理解)

(hash >>> segmentShift) & segmentMask算segment位置

segmentShift=32-ssize,假如ssize是2的n次方,那么ssize就是n,n最大为16,32就代表了32位

segmentMask就是2的n次方-1,其实就是segments数组的所有位置范围[0,2^n-1]

假设ssize是16,由于16=2^4,所以segmentShift=32-4=28,segmentMask=16-1=15

hash >>> segmentShift,其实就把hash无符号右移28位,那么就只剩下高4位,其实就是高4位于segmentMask取&.可以这么理解n是几,就代表了几位参与运算,n最大16.

hash >>> segmentShift可以理解为高n位,segmentMask可以理解为低n位,然后两者取&.

## 6.ensureSegment

初始化segment方法

执行流程

(1) 计算位置,使用UNSAFE的方法判断当前位置有没有初始化,然后使用segmets[0]的模板创建一个新的HashEntry[],再次判断当前位置有没有初始化,可能存在多线程同时初始化,然后创建一个新的segment,最后使用自旋cas设置新的segment的位置,保证只有一个线程初始化成功.

```

/**
 *
 * @param k 位置
 * @return segments
 */
private Segment<K,V> ensureSegment(int k) {
    final Segment<K,V>[] ss = this.segments; // 当前的segments数组
    long u = (k << SSHIFT) + SBASE; // raw offset // 计算原始偏移量,在segments数组的位置
    Segment<K,V> seg;
    if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u)) == null) { // 判断没有被初始化
        Segment<K,V> proto = ss[0]; // use segment 0 as prototype // 获取第一个segment ss[0]
        // 这就是为什么要在初始化化map时要初始化一个segment,需要用cap和loadFactoe 为模板
        int cap = proto.table.length; // 容量
        float lf = proto.loadFactor; // 负载因子
        int threshold = (int)(cap * lf); // 阈值
        // 初始化ss[k] 内部的tab数组 // recheck
        HashEntry<K,V>[] tab = (HashEntry<K,V>[])(new HashEntry[cap]);
    }
}

```

```

    // 再次检查这个ss[k] 有没有被初始化
    if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u))
        == null) { // recheck
        // 创建一个Segment
        Segment<K,V> s = new Segment<K,V>(lf, threshold, tab);
        // 这里用自旋CAS来保证把segments数组的u位置设置为s
        // 万一有多线程执行到这一步,只有一个成功,break
        // getObjectVolatile 保证了读的可见性,所以一旦有一个线程初始化了,那么就结束自
    旋
    while ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u))
        == null) {
        if (UNSAFE.compareAndSwapObject(ss, u, null, seg = s))
            break;
    }
}
return seg;
}

```

### scanAndLockForPut自旋获取锁方法

具体流程看代码注释,解释下这个方法的优化(看的某位大佬的博客)

(1) 我们在put方法获取锁失败,才会进入这个方法,这个方法采用自旋获取锁,直到成功才返回,但是使用了自旋次数的限制,这么做的好处是什么了,就是竞争太激烈的话,这个线程可能一直获取不到锁,自旋也是消耗cpu性能的,所以当达到自旋次数时,就阻塞当前线程,直到有线程释放了锁,通知这些线程.在等待过程中是不消耗cpu的.

(2) 当我们进入这个方法时,说明获取锁失败,那么可别是别的线程在对这个segment进行修改操作,所以说如果别的线程在操作之后,我们自己的工作内存中的数据可能已经不是最新的了,这个时候我们使用具有volatile语义的方法重新读了数据,在自旋过程中遍历这些数据,把最新的数据缓存在工作内存中,当前线程再次获取锁时,我们的数据是最新的,就不用重新去往内存中获取,这样在自旋获取的锁的过程中就预热了这些数据,在获取锁之后的执行中就提升了效率.

```

private HashEntry<K,V> scanAndLockForPut(K key, int hash, V value) {
    HashEntry<K,V> first = entryForHash(this, hash); // 根据hash获取头结点
    HashEntry<K,V> e = first;
    HashEntry<K,V> node = null;
    int retries = -1; // 是为了找到对应hash桶,遍历链表时找到就停止
    while (!tryLock()) { // 尝试获取锁,成功就返回,失败就开始自旋
        HashEntry<K,V> f; // to recheck first below
        if (retries < 0) {
            if (e == null) { // 结束遍历节点
                if (node == null) // 创造新的节点
                    node = new HashEntry<K,V>(hash, key, value, null);
                retries = 0; // 结束遍历
            }
            else if (key.equals(e.key)) // 找到节点 停止遍历
                retries = 0;
            else
                e = e.next; // 下一个节点 直到为null
        }
        else if (++retries > MAX_SCAN_RETRIES) { // 达到自旋的最大次数
            lock(); // 进入加锁方法,失败进入队列,阻塞当前线程
            break;
        }
        else if ((retries & 1) == 0 &&
            (f = entryForHash(this, hash)) != first) {
            e = first = f; // 头结点变化,需要重新遍历,说明有新的节点加入或者移除
        }
    }
}

```

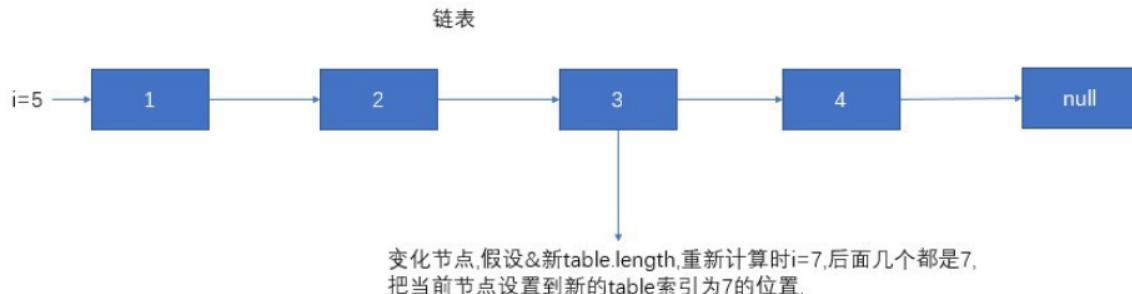
```

        retries = -1;
    }
}
return node;
}

```

### rehash扩容方法

解释下节点位置变化这一块的处理,如下图所示.



那么我们在第二次遍历时,3后面的节点在都不用处理了,因为它在新table的位置是对的,所以我们只需要重新计算1和2的位置即可.但是我们发现不需要第一次遍历也可以,直接重新计算所有的节点位置.所以这就看这个位置在前或者在后了,变化的位置越在前面第二次遍历越划算,处理的节点越少.

```

/**
 * 扩容方法
 */
private void rehash(HashEntry<K,V> node) {

    // 旧的table
    HashEntry<K,V>[] oldTable = table;
    // 旧的table的长度
    int oldCapacity = oldTable.length;
    // 扩容原来capacity的一倍
    int newCapacity = oldCapacity << 1;
    // 新的阈值
    threshold = (int)(newCapacity * loadFactor);
    // 新的table
    HashEntry<K,V>[] newTable =
        (HashEntry<K,V>[]) new HashEntry[newCapacity];
    // 新的掩码
    int sizeMask = newCapacity - 1;
    // 遍历旧的table
    for (int i = 0; i < oldCapacity ; i++) {
        // table中的每一个链表元素
        HashEntry<K,V> e = oldTable[i];
        if (e != null) { // e不等于null
            HashEntry<K,V> next = e.next; // 下一个元素
            int idx = e.hash & sizeMask; // 重新计算位置,计算在新的table的位置
            if (next == null) // single node on list 证明只有一个元素
                newTable[idx] = e; // 把当前的e设置给新的table
            else { // Reuse consecutive sequence at same slot
                HashEntry<K,V> lastRun = e; // 当前
                int lastIdx = idx; // 在新table的位置
                for (HashEntry<K,V> last = next;
                     last != null;
                     last = last.next) { // 遍历链表
                    int k = last.hash & sizeMask; // 确定在新table的位置
                    newTable[k] = last;
                }
            }
        }
    }
}

```

```

        if (k != lastIdx) { // 头结点和头结点的next元素的节点发生了变化
            lastIdx = k;    // 记录变化位置
            lastRun = last; // 记录变化节点
        }
    }
    // 以下把链表设置到新table分为两种情况
    // (1) lastRun 和 lastIdx 没有发生变化,也就是整个链表的每个元素位置和一样,都没有发生变化
    // (2) lastRun 和 lastIdx 发生了变化,记录变化位置和变化节点,然后把变化的这个节点设置到新table
    //      ,但是整个链表的位置只有变化节点和它后面关联的节点是对的
    //      下面的这个遍历就是处理这个问题,遍历当前头节点e,找出不等于变化节点(lastRun)的节点重新处理
    newTable[lastIdx] = lastRun;
    // clone remaining nodes
    for (HashEntry<K,V> p = e; p != lastRun; p = p.next) {
        V v = p.value;
        int h = p.hash;
        int k = h & sizeMask;
        HashEntry<K,V> n = newTable[k];
        newTable[k] = new HashEntry<K,V>(h, p.key, v, n);
    }
}
}

// 处理扩容时那个添加的节点

// 计算位置
int nodeIndex = node.hash & sizeMask; // add the new node
// 设置next节点,此时已经扩容完成,要从新table里面去当前位置的头结点为next节点
node.setNext(newTable[nodeIndex]);
// 设置位置
newTable[nodeIndex] = node;
// 新table替换旧的table
table = newTable;
}
}

```

## 7.get过程 (没有加锁,效率高)

注意:get方法使用了getObjectVolatile方法读取segment和hashentry,保证是最新的,具有锁的语义,可见性

分析:为什么get不加锁可以保证线程安全

- (1) 首先获取value,我们要先定位到segment,使用了UNSAFE的getObjectVolatile具有读的volatile语义,也就表示在多线程情况下,我们依旧能获取最新的segment.
- (2) 获取hashentry[],由于table是每个segment内部的成员变量,使用volatile修饰的,所以我们也能够获取最新的table.
- (3) 然后我们获取具体的hashentry,也时使用了UNSAFE的getObjectVolatile具有读的volatile语义,然后遍历查找返回.
- (4) 总结我们发现整个get过程中使用了大量的volatile关键字,其实质是保证了可见性(加锁也可以,但是降低了性能),get只是读取操作,所以我们只需要保证读取的是最新的数据即可.

```

/**
 * get 方法
 */
public V get(Object key) {

```

```

Segment<K,V> s; // manually integrate access methods to reduce overhead
HashEntry<K,V>[] tab;
int h = hash(key);
long u = (((h >>> segmentsShift) & segmentMask) << SSHIFT) + SBASE; // 获取
segment的位置
// getObjectVolatile getObjectVolatile语义读取最新的segment,获取table
if ((s = (Segment<K,V>)UNSAFE.getObjectVolatile(segments, u)) != null &&
    (tab = s.table) != null) {
    // getObjectVolatile getObjectVolatile语义读取最新的hashEntry,并遍历
    for (HashEntry<K,V> e = (HashEntry<K,V>) UNSAFE.getObjectVolatile(
        tab, ((long)((tab.length - 1) & h)) << TSHIFT) + TBASE);
        e != null; e = e.next) {
        K k;
        // 找到相同的key 返回
        if ((k = e.key) == key || (e.hash == h && key.equals(k)))
            return e.value;
    }
}
return null;
}

```

## 8.size方法

尝试3次不加锁获取sum,如果发生变化就全部加锁,size和containsValue方法的思想也是基本类似.

执行流程

- (1) 第一次,retries++=0,不满足全部加锁条件,遍历所有的segment,sum就是所有segment的容量,last等于0,第一次不相等,last=sum.
- (2) 第二次,retries++=1,不满足加锁条件,计算所有的segment,sum就是所有的segment的容量,last是上一次的sum,相等结束循环,不相等下次循环.
- (3) 第三次,retries++=2,先运算后赋值,所以此时还是不满足加锁条件和上面一样统计sum,判断这一次的sum和last(上一次的sum)是否相等,相等结束,不相等,下一次循环.
- (4) 第四次,retries++=2,满足加锁条件,给segment全部加锁,这样所有线程就没有办法进行修改操作,统计每个segment的数量求和,然后返回size.(ps:全部加锁提高了size的准确率,但是降低了吞吐量,统计size的过程中如果其它线程进行修改操作这些线程全部自旋或者阻塞).

```

/**
 * size
 * @return
 */
public int size() {
    // Try a few times to get accurate count. On failure due to
    // continuous async changes in table, resort to locking.
    final Segment<K,V>[] segments = this.segments;
    int size;
    boolean overflow; // 为true表示size溢出32位
    long sum;          // modCounts的总和
    long last = 0L;   // previous sum
    int retries = -1; // 第一次不计算次数,所以会重试三次
    try {
        for (;;) {
            if (retries++ == RETRIES_BEFORE_LOCK) { // 重试次数达到3次 对所有
                segment加锁
                    for (int j = 0; j < segments.length; ++j)
                        ensureSegment(j).lock(); // force creation
            }
            sum = 0L;
        }
    }
}

```

```

        size = 0;
        overflow = false;
        for (int j = 0; j < segments.length; ++j) {
            Segment<K,V> seg = segmentAt(segments, j);
            if (seg != null) { // seg不等于空
                sum += seg.modCount; // 不变化和size一样
                int c = seg.count; // seg 的size
                if (c < 0 || (size += c) < 0)
                    overflow = true;
            }
        }
        if (sum == last) // 没有变化
            break;
        last = sum; // 变化,记录这一次的变化值,下次循环时对比.
    }
} finally {
    if (retries > RETRIES_BEFORE_LOCK) {
        for (int j = 0; j < segments.length; ++j)
            segmentAt(segments, j).unlock();
    }
}
return overflow ? Integer.MAX_VALUE : size;
}

```

**remove** replace和remove都是用了scanAndLock这个方法

解释下scanAndLock这个方法,和put方法的scanAndLockForPut方法思想类似,都采用了同样的优化手段.

```

/**
 * 删除方法
 */
final V remove(Object key, int hash, Object value) {
    if (!tryLock()) // 获取锁
        scanAndLock(key, hash); // 自旋获取锁
    V oldValue = null;
    try {
        HashEntry<K,V>[] tab = table; // 当前table
        int index = (tab.length - 1) & hash; // 获取位置
        HashEntry<K,V> e = entryAt(tab, index); // 找到元素
        HashEntry<K,V> pred = null;
        while (e != null) {
            K k;
            HashEntry<K,V> next = e.next; // 下一个元素
            if ((k = e.key) == key ||
                (e.hash == hash && key.equals(k))) { // e的key=传入的key
                V v = e.value; // 获取value
                if (value == null || value == v || value.equals(v)) { // 如果
                    value相等
                    if (pred == null) // 说明是头结点,让next节点为头结点即可
                        setEntryAt(tab, index, next);
                    else // 说明不是头结点,就把当前节点e的上一个节点pred的next节点
                        设置为当前e的next节点
                        pred.setNext(next);
                    ++modCount;
                    --count;
                    oldValue = v;
                }
            }
        }
    }
}

```

```

        break;
    }
    pred = e;
    e = next;
}
} finally {
    unlock();
}
return oldValue;
}

```

**isEmpty** 其实也和size的思想类似,不过这个始终没有加锁,提高了性能

执行流程

(1) 第一次遍历就干了一件事,确定map的每个segment是否为0,其中任何一个segment的count不为0,就返回,都为0,就累加modCount为sum.

(2) 判断sum是否为0,不为0,就代表了map之前是有数据的,被remove和clean了,modCount指的是操作次数,再次确定map的每个segment是否为0,其中任何一个segment的count不为0,就返回,并且累减sum,最后判断sum是否为0,为0就代表没有任何变化,不为0,就代表在第一次统计过程中有线程又添加了元素,所以返回false.但是如果在第二次统计时又发生了变化了,所以这个不是那么的准确,但是不加锁提高了性能,也是一个可以接受的方案.

```

public boolean isEmpty() {
    long sum = 0L;
    final Segment<K,V>[] segments = this.segments;
    for (int j = 0; j < segments.length; ++j) {
        Segment<K,V> seg = segmentAt(segments, j);
        if (seg != null) {
            if (seg.count != 0)
                return false; // 某一个不为null,立即返回
            sum += seg.modCount;
        }
    }
    // 上面执行完 说明不为空,并且过程可能发生了变化
    // 发生变化
    if (sum != 0L) { // recheck unless no modifications
        for (int j = 0; j < segments.length; ++j) {
            Segment<K,V> seg = segmentAt(segments, j);
            if (seg != null) {
                if (seg.count != 0)
                    return false;
                sum -= seg.modCount;
            }
        }
        if (sum != 0L) // 变化值没有为0,说明不为空
            return false;
    }
    // 没有发生变化
    return true;
}

```

## 9.总结

1.7 ConcurrentHashMap 使用了分段锁的思想提高了并发的访问量,就是使用很多把锁,每一个 segment 代表了一把锁,每一段只能有一个线程获取锁;但是 segment 的数量初始化了,就不能修改,所以这也代表了并发的不能修改,这也是 1.7 的一个局限性.从 get 方法可以看出使用了 UNSAFE 的一些方法和 volatile 关键字来代替锁,提高了并发性.在 size 和 containsValue 这些方法提供一种尝试思想,先不加锁尝试统计,如果其中没有变化就返回,有变化接着尝试,达到尝试次数再加锁,这样也避免了立即加锁对并发的影响.

## JDK8 (Synchronized + CAS + Node + Unsafe)

### HashSet

底层维护一个HashMap, add 方法就是把对象作为 hashmap 的 key, 利用 key 的不可重复性, 保证存入对象的唯一性。

### List

#### CopyOnWriteArrayList

在很多应用场景中, 读操作可能会远远大于写操作。由于读操作根本不会修改原有的数据, 因此如果每次读取都进行加锁操作, 其实是一种资源浪费。我们应该允许多个线程同时访问 List 的内部数据, 毕竟读操作是线程安全的。只保证最终一致性。

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

写入操作 `add()` 方法在添加集合的时候加了锁, 保证同步, 避免多线程写的时候会 copy 出多个副本。不过多线程中 remove 有数组越位的可能, 因为删除也是拷贝到新数组后对新数组进行删除, 此时如果有线程读取数组最后一个元素, 由于数组长度改变, 将可能发生越位。

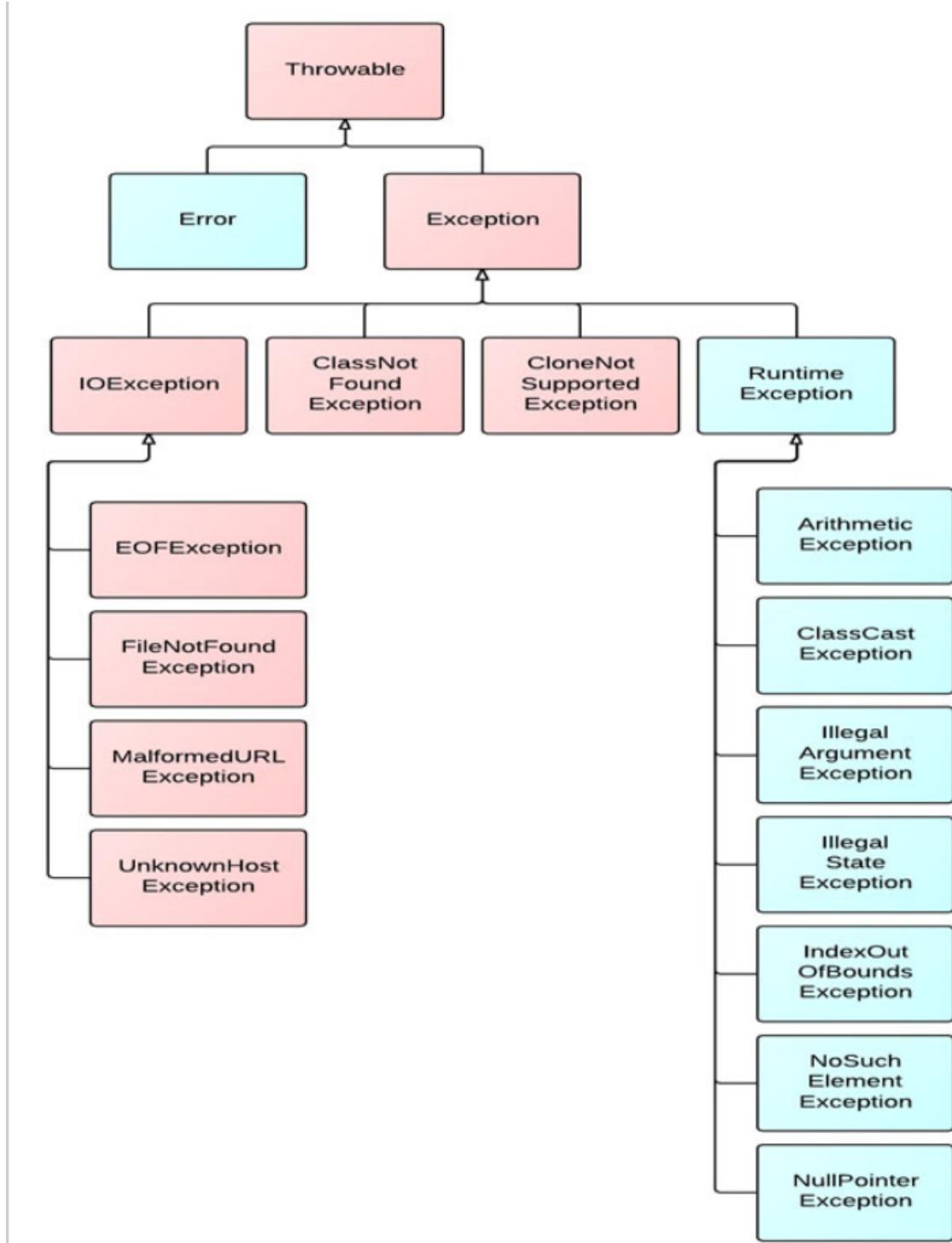
```
public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock(); // 加锁
    try {
        Object[] elements = getArray();
        int len = elements.length;
        Object[] newElements = Arrays.copyOf(elements, len + 1); // 拷贝新数组
        newElements[len] = e;
        setArray(newElements);
        return true;
    } finally {
        lock.unlock(); // 释放锁
    }
}
```

### 泛型

### 反射

### 异常

异常的继承结构：Throwable为基类，Error和Exception继承Throwable，RuntimeException和IOException等继承Exception。Error和RuntimeException及其子类成为未检查异常（unchecked），其它异常成为已检查异常（checked）。



## Error异常

Error表示程序在运行期间出现了十分严重、不可恢复的错误，在这种情况下应用程序只能中止运行，例如JAVA虚拟机出现错误。Error是一种unchecked Exception，编译器不会检查Error是否被处理，在程序中不用捕获Error类型的异常。一般情况下，在程序中也不应该抛出Error类型的异常。

## RuntimeException异常

Exception异常包括RuntimeException异常和其他非RuntimeException的异常。

RuntimeException 是一种Unchecked Exception，即表示编译器不会检查程序是否对 RuntimeException作了处理，在程序中不必捕获RuntimException类型的异常，也不必在方法体声明抛出 RuntimeException类。RuntimeException发生的时候，表示程序中出现了编程错误，所以应该找出错误修改程序，而不是去捕获RuntimeException。

## Checked Exception异常

Checked Exception异常，这也是在编程中使用最多的Exception，所有继承自Exception并且不是 RuntimeException的异常都是checked Exception，上图中的IOException和 ClassNotFoundException。JAVA语言规定必须对checked Exception作处理，编译器会对此作检查，要么在方法体中声明抛出checked Exception，要么使用catch语句捕获checked Exception进行处理，不然不能通过编译。

# 序列化

---

## 概念

基于对象能够在程序不运行的情况下仍能存在并保存其信息的需求，对象的序列化功能孕育而生。对象的序列化是指通过某种方法把对象以字节序列的形式保存起来。反之通过字节序列得到原对象就是反序列化

## 使用

只要对象实现了Serializable接口，该对象就可以进行序列化，其中serialVersionUID是实现 Serializable接口的类都要生成的一个静态常量，有两种生成方式：**一是直接定义为1L，一是随机生成；**其作用是为了保证反序列化时找到正确的版本。（如果没有显式声明serialVersionUID，JVM将根据您的Serializable类的各个方面自动为您执行此操作，而不同的jvm计算的结果也不同）。

Externalizable继承自Serializable，该接口中定义了两个抽象方法：writeExternal()与readExternal()。当使用Externalizable接口来进行序列化与反序列化的时候需要开发人员重写writeExternal()与readExternal()方法。否则所有变量的值都会变成默认值。

transient关键字的作用是控制变量的序列化，在变量声明前加上该关键字，可以阻止该变量被序列化到文件中，在被反序列化后，transient变量的值被设为初始值，如 int型的是0，对象型的是null。

在进行反序列化时，JVM会把传来的字节流中的serialVersionUID与本地相应实体类的serialVersionUID进行比较，如果相同就认为是一致的，可以进行反序列化，否则就会出现序列化版本不一致的异常，即是InvalidCastException

序列化保存的是对象的状态，静态变量属于类的状态，因此序列化并不保存静态变量。A进程里这个类的静态变量是1，B进程里是2，实例变量则正常序列化反序列化

引用类型的属性会随着对象序列化而序列化，不过引用的类型也要实现Serializable接口

## 原理

## 枚举特别处理

## JDK 8

---

## 新特性概述

# Lambda

## Stream

### 新日期API

#### 概述

JDK8中增加了一套全新的日期时间API，位于 java.time 包中，下面是一些关键类.不可变的线程安全类

- Instant: 代表的是时间戳
- LocalDate: 不包含具体时间的日期，格式为 2020-01-03
- LocalTime: 不含日期的时间，格式为 09:53:56.749
- LocalDateTime: 包含了日期及时间，格式为 2020-01-03T09:53:56.749

#### 代码块示例

```
1     Instant instant = Instant.now();
2     System.out.println("当前时间戳是: " + instant); //当前时间戳是: 2018-09-
06T10:14:29.460Z
3     Date date = Date.from(instant);
4     System.out.println("当前时间戳是: " + date); //当前时间戳是: Thu Sep 06
18:14:29 CST 2018
5     instant = date.toInstant();
```

```
1     LocalDate nowDate = LocalDate.now();
2     System.out.println("今天的日期: " + nowDate); //今天的日期: 2018-09-06
3     int year = nowDate.getYear(); //年: 一般用这个方法获取年
4     System.out.println("year: " + year); //year: 2018
5     int month = nowDate.getMonthValue(); //月: 一般用这个方法获取月
6     System.out.println("month: " + month); //month: 9
7     int day = nowDate.getDayOfMonth(); //日: 当月的第几天, 一般用这个方法获取日
8     System.out.println("day: " + day); //day: 6
9
10    int dayOfYear = nowDate.getDayOfYear(); //日: 当年的第几天
11    System.out.println("dayOfYear: " + dayOfYear); //dayOfYear: 249
12
13    //星期
14    System.out.println(nowDate.getDayOfWeek()); //THURSDAY
15    System.out.println(nowDate.getDayOfWeek().getValue()); //4
16    //月份
17    System.out.println(nowDate.getMonth()); //SEPTEMBER
18    System.out.println(nowDate.getMonth().getValue()); //9
19
20    System.out.println(LocalDate.of(1991, 11, 11)); //直接传入对应的年月日
21    System.out.println(LocalDate.of(1991, Month.NOVEMBER, 11)); //相对上面只是把月换成了枚举
22    LocalDate birDay = LocalDate.of(1991, 11, 11);
23    System.out.println(LocalDate.ofYearDay(1991,
birDay.getDayOfYear())); //第一个参数为年, 第二个参数为当年的第多少天
24    System.out.println(LocalDate.ofEpochDay(birDay.toEpochDay())); //参数为距离1970-01-01的天数
```

```
24     System.out.println(LocalDate.parse("1991-11-11"));
25
System.out.println(LocalDate.parse("19911111", DateTimeFormatter.ofPattern("yyyyM
dd")));
```

```
1     LocalTime nowTime = LocalTime.now();
2     System.out.println("今天的时间: " + nowTime); //今天的时间: 15:33:56.749
3     int hour = nowTime.getHour(); //时
4     System.out.println("hour: " + hour); //hour: 15
5     int minute = nowTime.getMinute(); //分
6     System.out.println("minute: " + minute); //minute: 33
7     int second = nowTime.getSecond(); //秒
8     System.out.println("second: " + second); //second: 56
9     int nano = nowTime.getNano(); //纳秒
10    System.out.println("nano: " + nano); //nano: 749000000
```

```
1     LocalDateTime nowDateTime = LocalDateTime.now();
2     System.out.println("今天是: " + nowDateTime); //今天是: 2018-09-
06T15:33:56.750
3     System.out.println(nowDateTime.getYear()); //年
4     System.out.println(nowDateTime.getMonthValue()); //月
5     System.out.println(nowDateTime.getDayOfMonth()); //日
6     System.out.println(nowDateTime.getHour()); //时
7     System.out.println(nowDateTime.getMinute()); //分
8     System.out.println(nowDateTime.getSecond()); //秒
9     System.out.println(nowDateTime.getNano()); //纳秒
10    //日: 当年的第几天
11    System.out.println("dayOfYear: " +
nowDateTime.getDayOfYear()); //dayOfYear: 249
12    //星期
13    System.out.println(nowDateTime.getDayOfWeek()); //THURSDAY
14    System.out.println(nowDateTime.getDayOfWeek().getValue()); //4
15    //月份
16    System.out.println(nowDateTime.getMonth()); //SEPTEMBER
17    System.out.println(nowDateTime.getMonth().getValue()); //9
```

## 日期时间比较

在JDK8中， LocalDate类中使用isBefore()、 isAfter()、 equals()方法来比较两个日期，可直接进行比较

```
1     LocalDate myDate = LocalDate.of(2018, 9, 5);
2     LocalDate nowDate = LocalDate.now();
3     System.out.println("今天是2018-09-06吗? " + nowDate.equals(myDate));
4     System.out.println(myDate + "是否在" + nowDate + "之前?" +
myDate.isBefore(nowDate)); //2018-09-05是否在2018-09-06之前? true
5     System.out.println(myDate + "是否在" + nowDate + "之后?" +
myDate.isAfter(nowDate)); //2018-09-05是否在2018-09-06之后? false
```

## 日期时间格式化

在JDK8之前，时间日期的格式化非常麻烦，经常使用SimpleDateFormat来进行格式化，但是 SimpleDateFormat并不是线程安全的，要结合ThreadLocal使用。在JDK8中，引入了一个全新的线程安全的日期与时间格式器DateTimeFormatter

```

1     LocalDateTime ldt = LocalDateTime.now();
2     System.out.println(ldt); //2018-09-06T18:22:47.366
3     DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss");
4     String ldtStr = ldt.format(dtf);
5     System.out.println(ldtStr); //2018-09-06 18:22:47
6     String ldtStr1 = dtf.format(ldt);
7     System.out.println(ldtStr1); //2018-09-06 18:22:47

```

# java 并发编程

## synchronized

Mark word记录了对象和锁有关的信息，当某个对象被synchronized关键字当成同步锁时，那么围绕这个锁的一系列操作都和Mark word有关系。Mark Word在32位虚拟机的长度是32bit、在64位虚拟机的长度是64bit。

Mark Word里面存储的数据会随着锁标志位的变化而变化，Mark Word可能变化为存储以下5中情况

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否偏向锁	锁标志位
无锁	对象的HashCode		分代年龄	0	01
偏向锁	线程ID	Epoch	分代年龄	1	01
轻量级锁	指向栈中锁记录的指针				
重量级锁	指向重量级锁的指针				
GC标记	空				

微信号: kxxmoye\_112  
10  
11

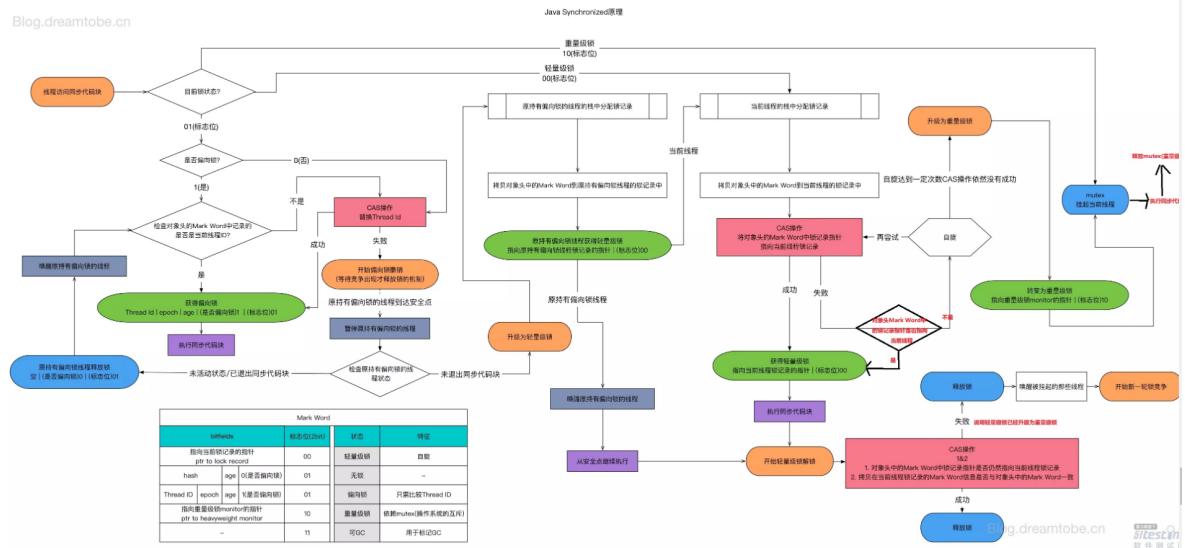
锁状态	25bit	31bit	1bit	4bit	1bit	2bit
			cms_free	分代年龄	偏向锁	锁标志位
无锁	unused	hashCode			0	01
偏向锁	ThreadID(54bit)	Epoch(2bit)			1	01

微信号: kxxmoye\_112  
1  
01

- 锁标识 lock=00 表示轻量级锁
- 锁标识 lock=10 表示重量级锁
- 偏向锁标识 biased\_lock=1表示偏向锁
- 偏向锁标识 biased\_lock=0且锁标识=01表示无锁状态

**自旋**: 当有个线程 A 去请求某个锁的时候，这个锁正在被其它线程占用，但是线程 A 并不会马上进入阻塞状态，而是循环请求锁(自旋)。这样做的目的是因为很多时候持有锁的线程会很快释放锁的，线程 A 可以尝试一直请求锁，没必要被挂起放弃 CPU 时间片，因为线程被挂起然后到唤醒这个过程开销很大，当然如果线程 A 自旋指定的时间还没有获得锁，仍然会被挂起。

**自适应性自旋**: 自适应性自旋是自旋的升级、优化，自旋的时间不再固定，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态决定。例如线程如果自旋成功了，那么下次自旋的次数会增多，因为 JVM 认为既然上次成功了，那么这次自旋也很有可能成功，那么它会允许自旋的次数更多。反之，如果对于某个锁，自旋很少成功，那么在以后获取这个锁的时候，自旋的次数会变少甚至忽略，避免浪费处理器资源。有了自适应性自旋，随着程序运行和性能监控信息的不断完善，JVM 对程序锁的状况预测就会变得越来越准确，JVM 也就变得越来越聪明。



## 偏向锁的获取

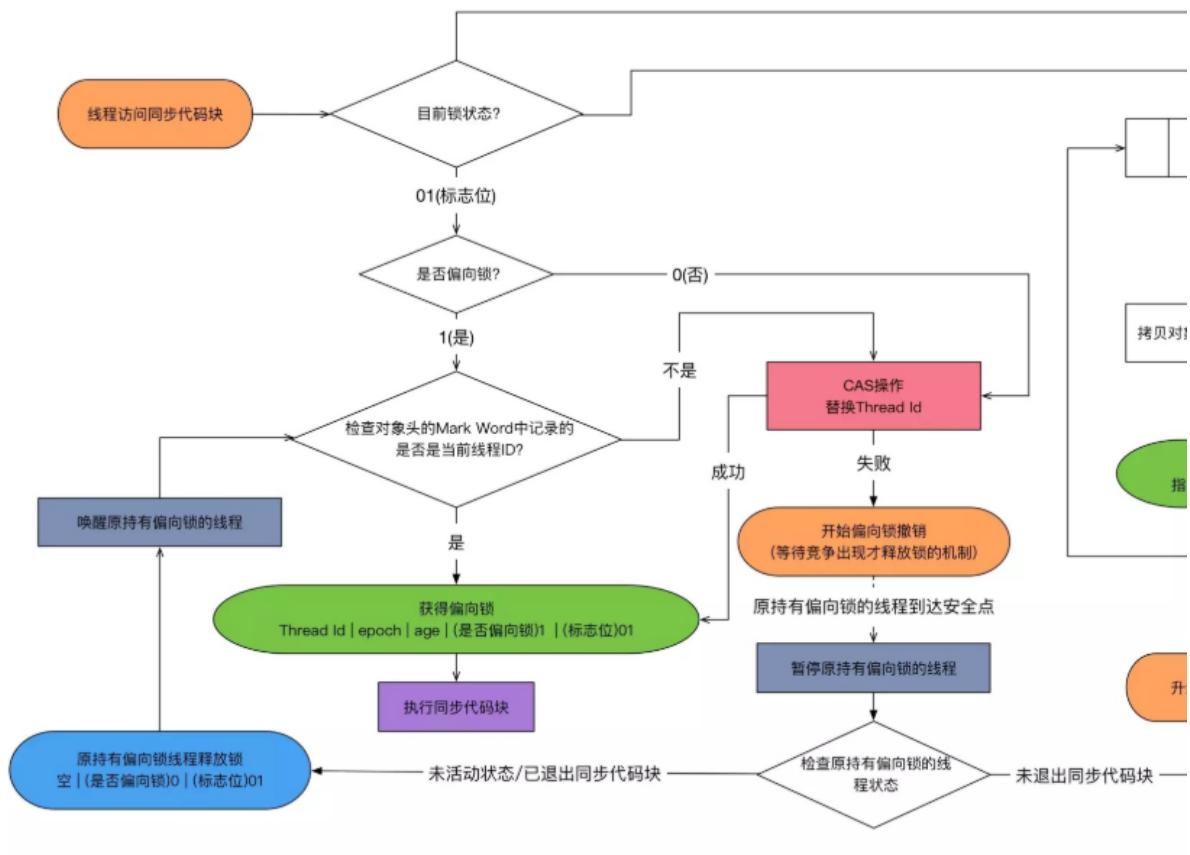
当一个线程访问同步块获取锁时，会在对象头和栈帧中的锁记录里存储偏向锁的线程ID，表示哪个线程获得了偏向锁，结合前面分析的Mark Word来分析一下偏向锁的获取逻辑

- 检查 **Mark Word 偏向锁的状态**，为1表示有偏向锁，为0表示无锁。无锁直接执行步骤3
- 如果是偏向锁，则 **检查 Mark Word 储存的线程 ID 是否为当前线程 ID**，如果是则执行同步块，否则执行步骤3。
- 如果检查到 **Mark word 的 ID 不是本线程的 ID**，则通过 **CAS 操作去修改线程 ID** 修改成本线程的 ID，如果修改成功则执行同步代码块，否则执行步骤4。
- 当拥有该锁的线程到达安全点之后，挂起这个线程，检查该线程状态，如果未退出同步代码块，则升级为轻量级锁并持有轻量级锁（先创建锁记录，复制mark word），唤醒全局安全点的当前线程继续竞争轻量级锁；如果该线程死去或者退出了同步代码块，就将锁状态改为无锁，并唤醒全局安全点的当前线程继续操作。

## 偏向锁的撤销

当其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放偏向锁，撤销偏向锁的过程需要等待一个全局安全点(所有工作线程都停止字节码的执行)。对应上面的步骤4

偏向锁获取流程

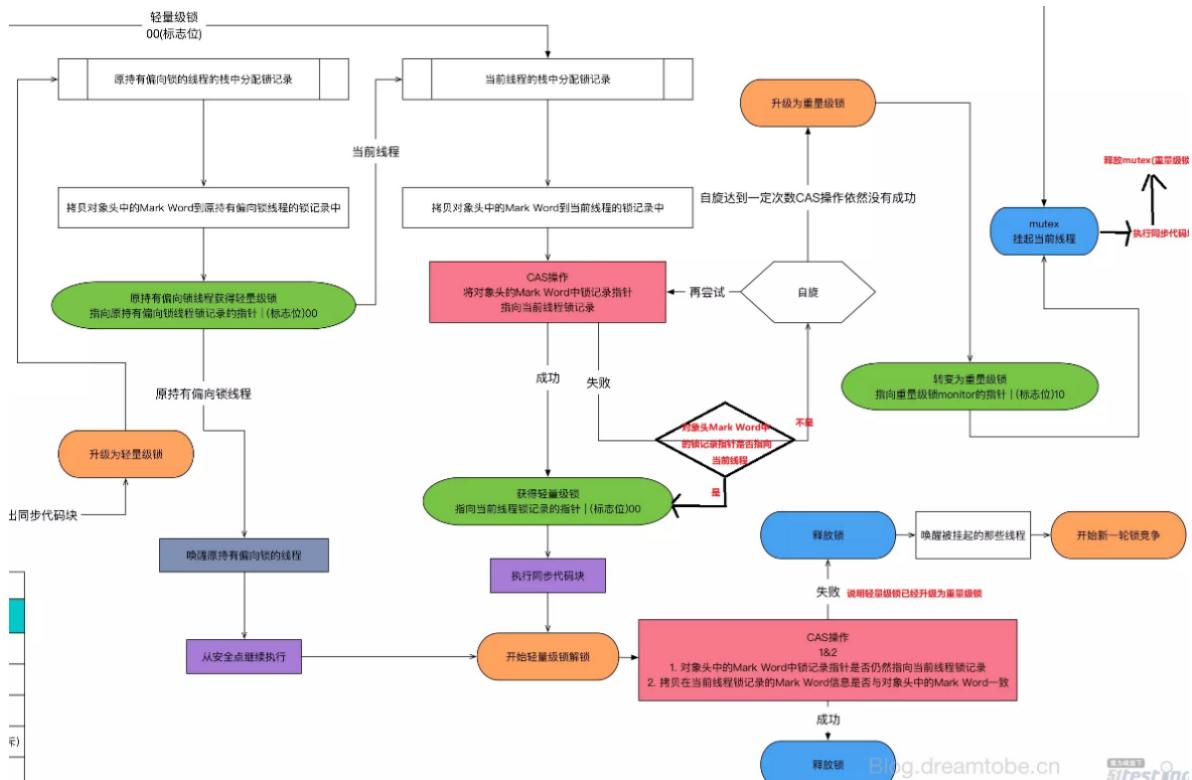


## 轻量级锁

前面我们知道，当存在超过一个线程在竞争同一个同步代码块时，会发生偏向锁的撤销。偏向锁撤销以后对象会可能会处于两种状态

1. 一种是无锁状态，简单来说就是已经获得偏向锁的线程已经退出了同步代码块
2. 一种是已偏向锁状态，简单来说就是已经获得偏向锁的线程正在执行同步代码块，那么这个时候会升级到轻量级锁并且被原持有锁的线程获得锁，唤醒全局安全点其他线程继续竞争。

加锁



1. JVM会先在当前线程的栈帧中创建用于存储锁记录的空间(LockRecord)
  2. 将对象头中的Mark Word复制到锁记录中，称为Displaced Mark Word.
  3. 线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针
  4. 如果替换成功，表示当前线程获得轻量级锁，如果失败，表示存在其他线程竞争锁，那么当前线程会尝试使用CAS来获取锁，当自旋超过指定次数(可以自定义)时仍然无法获得锁，此时锁会膨胀升级为重量级锁

解锁



1. 检查mark word中锁记录指针是否当前线程锁记录;尝试CAS操作将所记录中的Mark Word替换回到对象头中
  2. 如果成功, 表示没有竞争发生
  3. 如果失败, 表示当前锁存在竞争, 锁会膨胀成重量级锁

一旦锁升级成重量级锁，就不会再恢复到轻量级锁状态。当锁处于重量级锁状态，其他线程尝试获取锁时，都会被阻塞，也就是 BLOCKED 状态。当持有锁的线程释放锁之后会唤醒这些线程，被唤醒之后的线程会进行新一轮的竞争。

重量级锁

重量级锁依赖对象内部的monitor锁来实现，而monitor又依赖操作系统的MutexLock（互斥锁）

大家如果对MutexLock有兴趣，可以抽时间去了解，假设Mutex变量的值为1，表示互斥锁空闲，这个时候某个线程调用lock可以获得锁，而Mutex的值为0表示互斥锁已经被其他线程获得，其他线程调用lock只能挂起等待

为什么重量级锁的开销比较大呢？

原因是当系统检查到是重量级锁之后，会把等待想要获取锁的线程阻塞，被阻塞的线程不会消耗CPU，但是阻塞或者唤醒一个线程，都需要通过操作系统来实现，也就是相当于从用户态转化到内核态，而转化状态是需要消耗时间的

## 总结

1. 检测Mark Word里面是不是当前线程的ID，如果是，表示当前线程处于偏向锁
2. 如果不是，则使用CAS将当前线程的ID替换Mark Word，如果成功则表示当前线程获得偏向锁，置偏向标志位1
3. 如果失败，则说明发生竞争，撤销偏向锁，进而升级为轻量级锁。
4. 当前线程使用CAS将对象头的Mark Word替换为锁记录指针，如果成功，当前线程获得锁
5. 如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。
6. 如果自旋成功则依然处于轻量级状态。
7. 如果自旋失败，则升级为重量级锁。

ps:

在HotSpot VM里，Java对象会在首次真正使用到它的identity hash code（例如通过Object.hashCode() / System.identityHashCode()）时调用VM里的函数来计算出值，然后会保存在对象里，后面对同一对象查询其identity hash code时总是会返回最初记录的值。所以前面说“当前”指的是计算identity hash code的时间点，不是对象创建时。

这组实现代码在HotSpot VM里自从JDK6的早期开发版开始就没变过，只是hashCode选项的默认值变了而已。

当一个对象已经计算过identity hash code，它就无法进入偏向锁状态；

当一个对象当前正处于偏向锁状态，并且需要计算其identity hash code的话，则它的偏向锁会被撤销，并且锁会膨胀为重量锁；其实意思就是identity hash code和偏向锁无法共存，由于轻量级锁是拷贝的mark word，所以也无法共存。

重量锁的实现中，ObjectMonitor类里有字段可以记录非加锁状态下的mark word，其中可以存储identity hash code的值。或者简单说就是**重量锁可以存下identity hash code**。

## AQS

### 概念

AQS即AbstractQueuedSynchronizer，抽象队列同步器，用于实现锁机制，其核心思想是如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并将共享资源设置为锁定状态，如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS使用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中。CLH（Craig, Landin, and Hagersten）队列是一个虚拟的双向队列，虚拟的双向队列即不存在队列实例，仅存在节点之间的关联关系。（简单解释一下J.U.C，是JDK中提供的并发工具包，`java.util.concurrent`）

AQS是一个**抽象类**，主要是通过**继承**的方式来使用，它本身没有实现任何的同步接口，仅仅是定义了同步状态的获取以及释放的方法来提供自定义的同步组件

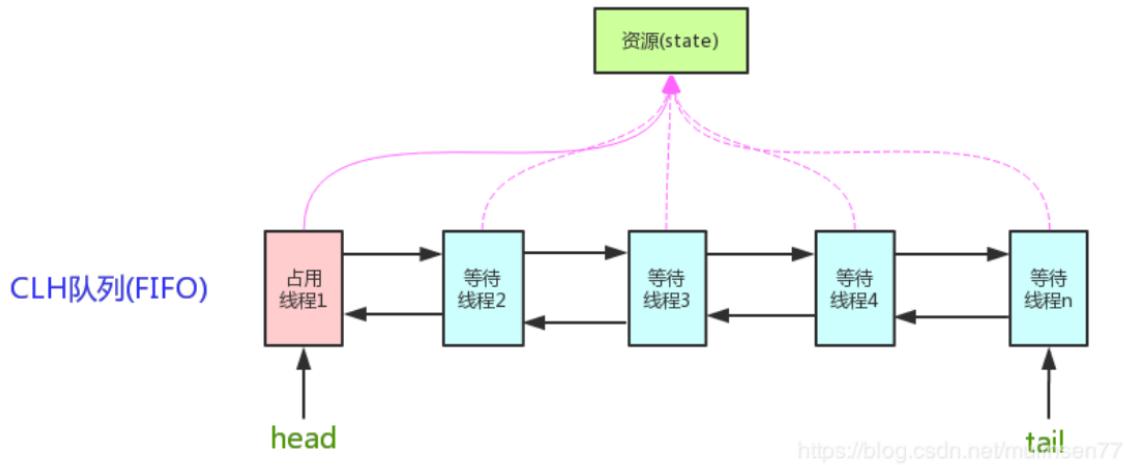
**AQS是将每一条请求共享资源的线程封装成一个CLH锁队列的一个结点（Node），来实现锁的分配。**

用大白话来说，AQS就是基于CLH队列，用volatile修饰共享变量state，线程通过CAS去改变状态符，成功则获取锁成功，失败则进入等待队列，等待被唤醒。

**注意：AQS是自旋锁：**在等待唤醒的时候，经常会使用自旋（while(!cas())）的方式，不停地尝试获取锁，直到被其他线程获取成功

实现了AQS的锁有：自旋锁、互斥锁、读锁写锁、条件产量、信号量、栅栏都是AQS的衍生物

## 原理



如图示，AQS维护了一个volatile int state和一个FIFO线程等待队列，多线程争用资源被阻塞的时候就会进入这个队列。state就是共享资源，其访问方式有如下三种：

getState();setState();compareAndSetState();

AQS 定义了两种资源共享方式：

1. **Exclusive**: 独占，只有一个线程能执行，如ReentrantLock
2. **Share**: 共享，多个线程可以同时执行，如Semaphore、CountDownLatch、ReadWriteLock, CyclicBarrier

不同的自定义的同步器争用共享资源的方式也不同。

## 从LOCK说起

Lock是一个接口，部分api如下

```
void lock() // 如果锁可用就获得锁，如果锁不可用就阻塞直到锁释放
void lockInterruptibly() // 和 lock()方法相似，但阻塞的线程可中断，抛出
java.lang.InterruptedIOException异常
boolean tryLock() // 非阻塞获取锁；尝试获取锁，如果成功返回true
boolean tryLock(long timeout, TimeUnit timeUnit) // 带有超时时间的获取锁方法
void unlock() // 释放锁
```

实现Lock接口的类有很多，以下为几个常见的锁实现

- ReentrantLock：表示重入锁，它是唯一一个实现了Lock接口的类。重入锁指的是线程在获得锁之后，再次获取该锁不需要阻塞，而是直接关联一次计数器增加重入次数
- ReentrantReadWriteLock：重入读写锁，它实现了ReadWriteLock接口，在这个类中维护了两个锁，一个是ReadLock，一个是WriteLock，他们都分别实现了Lock接口。读写锁是一种适合读多写少的场景下解决线程安全问题的工具，基本原则是：**读和读不互斥、读和写互斥、写和写互斥**。也就是说涉及到影响数据变化的操作都会存在互斥。

- StampedLock: stampedLock是JDK8引入的新的锁机制，可以简单认为是读写锁的一个改进版本，读写锁虽然通过分离读和写的功能使得读和写之间可以完全并发，但是读和写是有冲突的，如果大量的读线程存在，可能会引起写线程的饥饿。stampedLock是一种乐观的读策略，使得乐观锁完全不会阻塞写线程

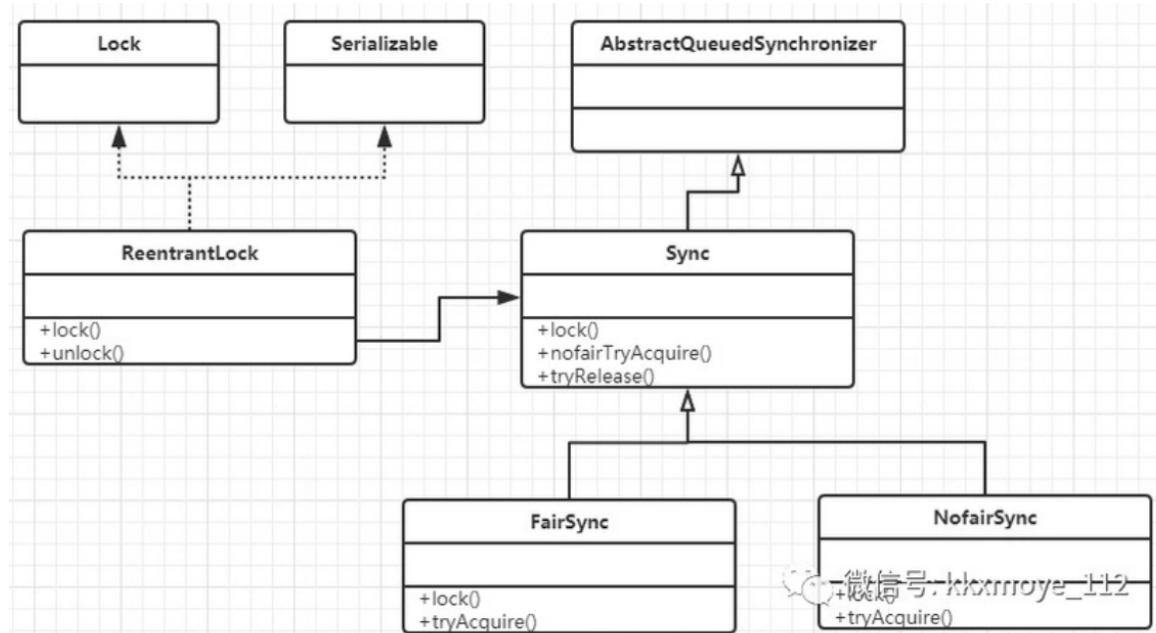
如何在实际应用中使用ReentrantLock呢？我们通过一个简单的demo来演示一下

```
public class Demo {
    private static int count=0;
    static Lock lock=new ReentrantLock();
    public static void inc(){
        lock.lock();
        try {
            Thread.sleep(1);
            count++;
        } catch (InterruptedException e) {
            e.printStackTrace();
        }finally{
            lock.unlock();
        }
    }
}
```

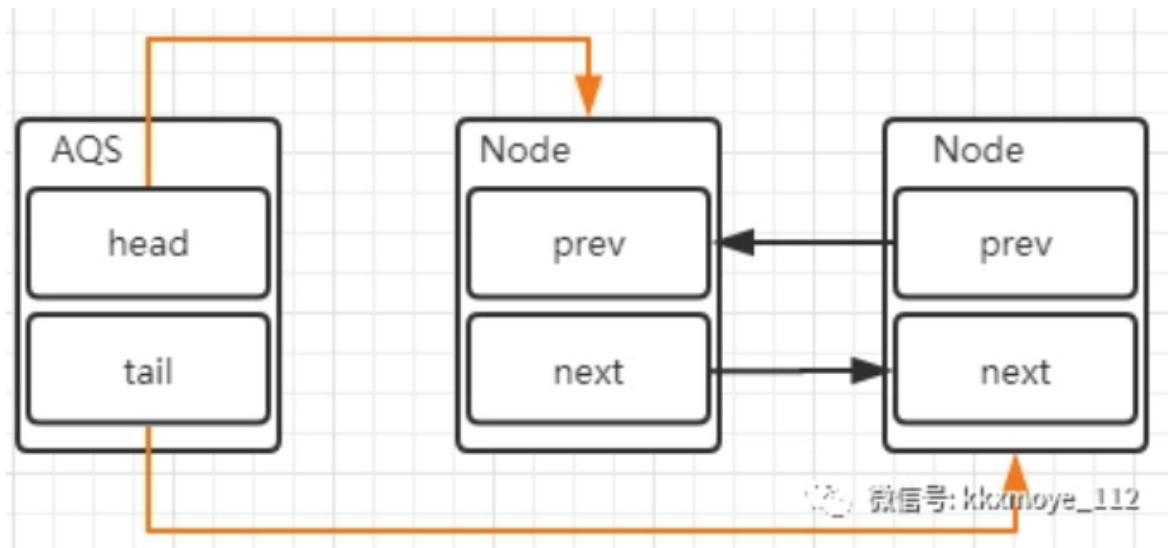
这段代码主要做一件事，就是通过一个静态的 `incr()` 方法对共享变量 `count` 做连续递增，在没有加同步锁的情况下多线程访问这个方法一定会存在线程安全问题。所以用到了 `ReentrantLock` 来实现同步锁，并且在 `finally` 语句块中释放锁。

那多个线程通过lock竞争锁时，当竞争失败的锁是如何实现等待以及被唤醒的呢？

仍然以ReentrantLock为例，来分析AQS在重入锁中的使用。毕竟单纯分析AQS没有太多的含义。先理解这个类图，可以方便我们理解AQS的原理



AQS的实现依赖内部的同步队列，也就是FIFO的双向队列，如果当前线程竞争锁失败，那么AQS会把当前线程以及等待状态信息构造成一个Node加入到同步队列中，同时再阻塞该线程。当获取锁的线程释放锁以后，会从队列中唤醒一个阻塞的节点(线程)。



AQS队列内部维护的是一个FIFO的双向链表，这种结构的特点是每个数据结构都有两个指针，分别指向直接的后继节点和直接前驱节点。所以双向链表可以从任意一个节点开始很方便的访问前驱和后继。每个Node其实是由线程封装，当线程争抢锁失败后会封装成Node加入到ASQ队列中去。

```

static final class Node {
    static final Node SHARED = new Node();
    static final Node EXCLUSIVE = null;
    static final int CANCELLED = 1;
    static final int SIGNAL = -1;
    static final int CONDITION = -2;
    static final int PROPAGATE = -3;
    volatile int waitStatus;
    volatile Node prev; //前驱节点
    volatile Node next; //后继节点
    volatile Thread thread; //当前线程
    Node nextwaiter; //存储在condition队列中的后继节点
    //是否为共享锁
    final boolean isshared() {
        return nextwaiter == SHARED;
    }

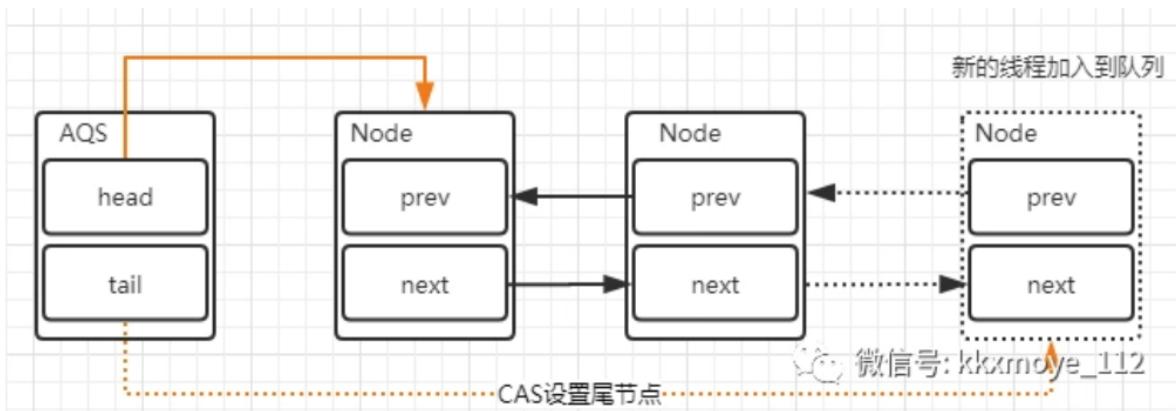
    final Node predecessor() throws NullPointerException {
        Node p = prev;
        if (p == null)
            throw new NullPointerException();
        else
            return p;
    }

    Node() { // Used to establish initial head or SHARED marker
    }
    //将线程构造成一个Node，添加到等待队列
    Node(Thread thread, Node mode) { // Used by addwaiter
        this.nextwaiter = mode;
        this.thread = thread;
    }
    //这个方法会在Condition队列使用，后续单独写一篇文章分析condition
    Node(Thread thread, int waitStatus) { // Used by Condition
        this.waitStatus = waitStatus;
        this.thread = thread;
    }
}

```

}

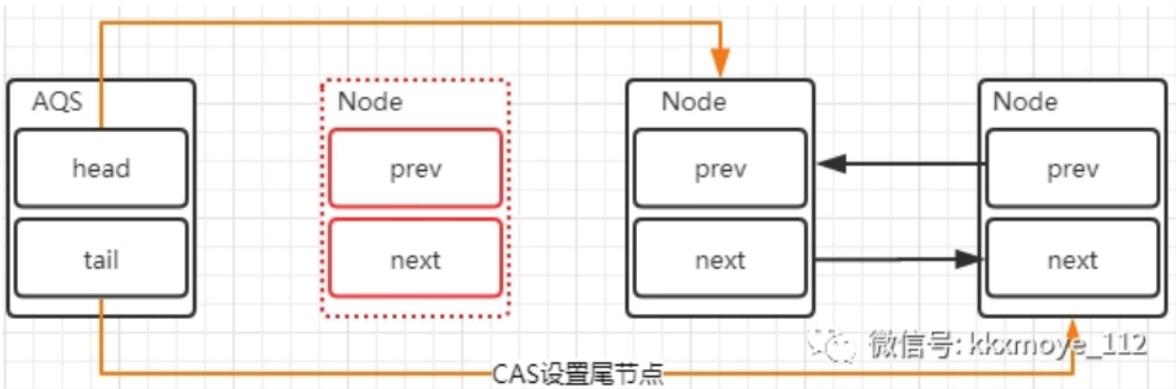
当出现锁竞争以及释放锁的时候，AQS同步队列中的节点会发生变化，首先看一下添加节点的场景（Node才是节点，head指向的就是持有锁的节点）。



这里会涉及到两个变化

- 新的线程封装成Node节点追加到同步队列中，设置prev节点以及修改当前节点的前置节点的next节点指向自己
- 通过CAS将tail重新指向新的尾部节点

释放锁，head指向的节点表示获取锁成功的节点，当头结点在释放同步状态时，会唤醒后继节点，如果后继节点获得锁成功，会把自己设置为头结点，节点的变化过程如下



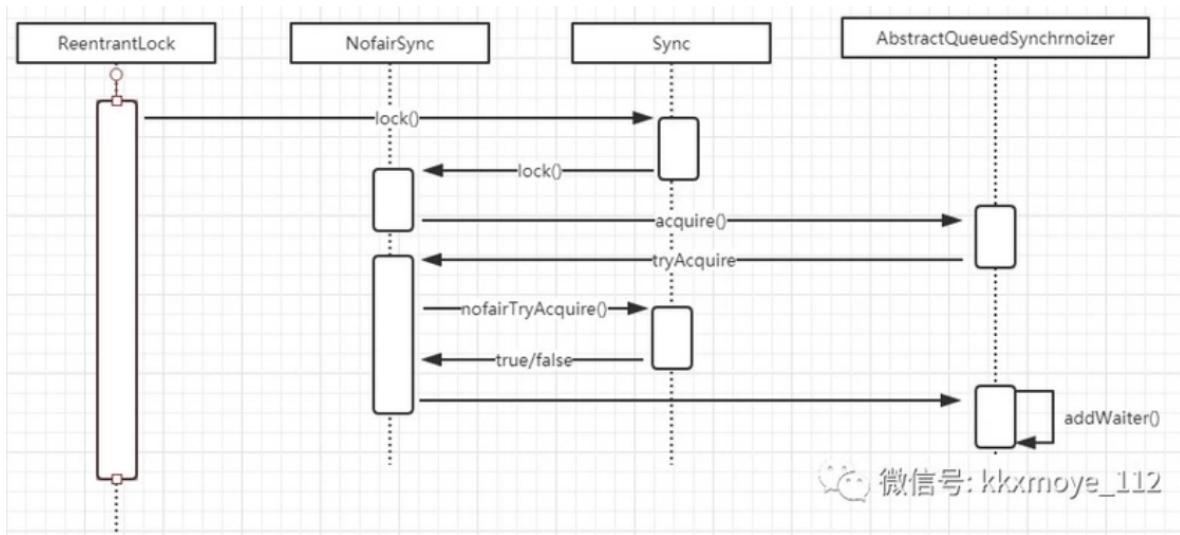
这个过程也是涉及到两个变化

- 修改head节点指向下一个获得锁的节点
- 新的获得锁的节点，将prev的指针指向null

这里有一个小的变化，就是设置head节点不需要用CAS，原因是设置head节点是由获得锁的线程来完成的，而同步锁只能由一个线程获得，所以不需要CAS保证，**只需要把head节点设置为原首节点的后继节点，并且断开原head节点的next引用即可**

## 源码分析

调用ReentrantLock中的lock()方法，源码的调用过程我使用了时序图来展现



## 1. ReentrantLock.lock()

```
public void lock() {
    sync.lock();
}

abstract static class Sync extends AbstractQueuedSynchronizer
```

sync是一个静态内部类，它继承了AQS这个抽象类，前面说过AQS是一个同步工具，主要用来实现同步控制。我们在利用这个工具的时候，会继承它来实现同步控制功能。

通过进一步分析，发现Sync这个类有两个**具体的实现**，分别是 NofairSync(非公平锁), FairSync(公平锁)。

- 公平锁 表示所有线程严格按照FIFO来获取锁
- 非公平锁 表示可以存在抢占锁的功能，也就是说不管当前队列上是否存在其他线程等待，新线程都有机会抢占锁

## 2. NonfairSync.lock

```
final void lock() {
    if (compareAndSetState(0, 1)) //通过cas操作来修改state状态，表示争抢锁的操作
        setExclusiveOwnerThread(Thread.currentThread()); //设置当前获得锁状态的线程
    else
        acquire(1); //尝试去获取锁
}
private volatile int state;
```

- 由于这里是公平锁，所以调用lock方法时，先去通过cas去抢占锁
- 如果抢占锁成功，保存获得锁成功的当前线程
- 抢占锁失败，调用acquire来走锁竞争逻辑

compareAndSetState的代码实现逻辑如下

```
// See below for intrinsics setup to support this
return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
```

- 当state=0时，表示无锁状态

- 当state>0时，表示已经有线程获得了锁，也就是state=1，但是因为ReentrantLock允许重入，所以同一个线程多次获得同步锁的时候，state会递增，比如重入5次，那么state=5。而在释放锁的时候，同样需要释放5次直到state=0其他线程才有资格获得锁

### 3.acquire(1)

是AQS中的方法，如果CAS操作未能成功，说明state已经不为0，此时继续acquire(1)操作，

```
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

- 通过tryAcquire尝试获取独占锁，如果成功返回true，失败返回false
- 如果tryAcquire失败，则会通过addWaiter方法将当前线程封装成Node添加到AQS队列尾部
- acquireQueued，将Node作为参数，通过自旋去尝试获取锁。

### 4.tryAcquire(int arg)

这个方法的作用是尝试获取锁，如果成功返回true，不成功返回false

它是重写AQS类中的tryAcquire方法，并且大家仔细看一下AQS中tryAcquire方法的定义，并没有实现，而是抛出异常。

```
// AQS里的
protected boolean tryAcquire(int arg) {
    throw new UnsupportedOperationException();
}

// NonfairSync继承重写了
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}
// 注意：NonfairSync继承Sync，Sync继承了AQS，Sync自己没有重写tryAcquire
```

```
// 代码在Sync中，由NonfairSync继承使用
final boolean nonfairTryAcquire(int acquires) {
    //获得当前执行的线程
    final Thread current = Thread.currentThread();
    int c = getState(); //获得state的值
    if (c == 0) { //state=0说明当前是无锁状态
        //通过cas操作来替换state的值改为1，大家想想为什么要用cas呢？
        //理由是，在多线程环境中，直接修改state=1会存在线程安全问题
        if (compareAndSetState(0, acquires)) {
            //保存当前获得锁的线程
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    //这段逻辑就很简单了。如果是同一个线程来获得锁，则直接增加重入次数
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires; //增加重入次数
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
    }
}
```

```

        return true;
    }
    return false;
}

```

- 获取当前线程，判断当前的锁的状态
- 如果state=0表示当前是无锁状态，通过cas更新state状态的值
- 如果当前线程是属于重入，则增加重入次数

当tryAcquire方法获取锁失败以后，则会先调用addWaiter将当前线程封装成Node，然后添加用acquireQueued添加到AQS队列。

## 5.addWaiter(Node mode)

acquireQueued(addWaiter(Node.EXCLUSIVE), arg)

```

private transient volatile Node tail;
private transient volatile Node head;
// 代码在AQS中实现了
private Node addWaiter(Node mode) { //mode=Node.EXCLUSIVE
    //将当前线程封装成Node，并且mode为独占锁
    Node node = new Node(Thread.currentThread(), mode);
    // Try the fast path of enq; backup to full enq on failure
    // tail是AQS中的表示同步队列队尾的属性，刚开始为null，所以进行enq(node)方法
    Node pred = tail;
    //这里为了提高性能，首先执行一次快速入队操作，即直接尝试将新节点加入队尾
    if (pred != null) { //tail不为空的情况，说明队列中存在节点数据
        node.prev = pred; //当前线程的Node的prev节点指向tail
        //这里根据CAS的逻辑，即使并发操作也只能有一个线程成功并返回，其余的都要执行后面的
        //入队操作。即enq()方法
        if (compareAndSetTail(pred, node)) { //通过cas把node添加到AQS队列
            pred.next = node; //cas成功，把旧的tail的next指针指向新的tail
            return node;
        }
    }
    enq(node); //tail=null，或者cas添加到AQS队列失败，将node添加到同步队列中
    return node;
}

```

- 将当前线程封装成Node
- 判断当前链表中的tail节点是否为空，如果不为空，则通过cas操作把当前线程的node添加到AQS队列
- 如果为空或者cas失败，调用enq将节点添加到AQS队列

enq就是通过自旋操作把当前节点加入到队列中

```

private Node enq(final Node node) {
    //自旋
    for (;;) {
        //如果是第一次添加到队列，那么tail=null
        Node t = tail;
        if (t == null) {
            //CAS的方式创建一个空的Node作为头结点
            if (compareAndSetHead(new Node()))
                //此时队列中只有一个头结点，所以tail也指向它
                tail = head;
        }
    }
}

```

```

        } else {
            //进行第二次循环时, tail不为null, 进入else区域。将当前线程的Node结点的
            prev指向tail, 然后使          用CAS将tail指向Node
                // 和上面addWaiter里是一样的操作
                node.prev = t;
                // 失败就自旋, 直到正常入队
                if (compareAndSetTail(t, node)) {
                    t.next = node;
                    return t;
                }
            }
        }
    }
}

```

仔细看会发现enq代码和addWaiter很像，以下是网友的一种解释

其实这种做法在并发编程中还挺常见，把最有可能成功执行的代码直接写在最常用的调用处，和C++中的inline方法一个意思。

具体到本例子，addWaiter的代码是最有可能一次成功的，因为在线程数不多的情况下，CAS还是很难失败的。因此这种写法可以节省多条指令。因为调用enq需要一次方法调用，进入循环，比较null，然后才到了addWaiter中一样的代码。

总而言之，节省指令，提高效率。

假如有两个线程t1,t2同时进入enq方法，t==null表示队列是首次使用，需要先初始化  
另外一个线程cas失败，则进入下次循环，通过cas操作将node添加到队尾

## 6.acquireQueued

将添加到队列中的Node作为参数传入acquireQueued方法，这里面会做抢占锁的操作

```

final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        // 自旋
        for (;;) {
            final Node p = node.predecessor(); // 获取prev节点,若为null即刻抛出
NullPointException
            if (p == head && tryAcquire(arg)) { // 前驱为head指向的节点时, 又进行一次
锁的抢夺
                setHead(node); //拿到锁, 把自己作为head指向的节点
                //凡是head指向的节点,head.thread与head.prev永远为null, 但是head.next不
为null
                p.next = null; // help GC, 把之前的head指向的节点干掉
                failed = false; //获取锁成功
                return interrupted;
            }
            //如果获取锁失败, 则根据节点的waitStatus决定是否需要挂起线程
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt()) // 若前面为true,则执行挂起,待下次唤醒的时候检
测中断的标志
                // 被中断唤醒后parkAndCheckInterrupt()返回true。继续执行 interrupted
= true;
                // 被当前获得锁的线程unpark唤醒, 则跳过这句;
                interrupted = true;
            }
        } finally {
    }
}

```

```

        if (failed) // 如果抛出异常则取消锁的获取,进行出队(sync queue)操作
            cancelAcquire(node);
    }

private void setHead(Node node) {
    head = node;
    node.thread = null;
    node.prev = null;
}

/**
 * 该方法实现某个node取消获取锁。
 */
private void cancelAcquire(Node node) {

    if (node == null)
        return;

    node.thread = null;

    // 遍历并更新节点前驱, 把node的prev指向头部第一个非取消节点。
    Node pred = node.prev;
    while (pred.waitStatus > 0)
        node.prev = pred = pred.prev;

    // 记录pred节点的后继为predNext, 后续CAS会用到。
    Node predNext = pred.next;

    // 直接把当前节点的等待状态置为取消, 后继节点即便也在cancel可以跨越node节点。
    node.waitStatus = Node.CANCELLED;

    /*
     * 如果CAS将tail从node置为pred节点了
     * 则剩下要做的事情就是尝试用CAS将pred节点的next更新为null以彻底切断pred和node的联系。
     * 这样一来就断开了pred与pred的所有后继节点, 这些节点由于变得不可达, 最终会被回收掉。
     * 由于node没有后继节点, 所以这种情况到这里整个cancel就算是处理完毕了。
     *
     * 这里的CAS更新pred的next即使失败了也没关系, 说明有其它新入队线程或者其它取消线程更新掉了。
     */
    if (node == tail && compareAndSetTail(node, pred)) {
        compareAndSetNext(pred, predNext, null);
    } else {
        // 如果node还有后继节点, 这种情况要做的事情是把pred和后继非取消节点拼起来。
        int ws;
        if (pred != head &&
            ((ws = pred.waitStatus) == Node.SIGNAL ||
             (ws <= 0 && compareAndSetWaitStatus(pred, ws, Node.SIGNAL))) &&
            pred.thread != null) {
            Node next = node.next;
            /*
             * 如果node的后继节点next非取消状态的话, 则用CAS尝试把pred的后继置为node的后继
             * 节点
             * 这里if条件为false或者CAS失败都没关系, 这说明可能有多个线程在取消, 总归会有一个
             * 能成功的。
             */
            if (next != null && next.waitStatus <= 0)

```

```

        compareAndSetNext(pred, predNext, next);
    } else {
        /*
         * 这时说明pred == head或者pred状态取消或者pred.thread == null
         * 在这些情况下为了保证队列的活跃性，需要去唤醒一次后继线程。
         * 举例来说pred == head完全有可能实际上目前已经没有线程持有锁了，
         * 自然就不会有释放锁唤醒后继的动作。如果不唤醒后继，队列就挂掉了。
         *
         * 这种情况下看似由于没有更新pred的next的操作，队列中可能会留有一大把的取消节点。
         * 实际上不要紧，因为后继线程唤醒之后会走一次试获取锁的过程，
         * 失败的话会走到shouldParkAfterFailedAcquire的逻辑。
         * 那里面的if中有处理前驱节点如果为取消则维护pred/next，踢掉这些取消节点的逻辑。
         */
        unparkSuccessor(node);
    }

    /*
     * 取消节点的next之所以设置为自己本身而不是null，
     * 是为了方便AQS中Condition部分的isOnSyncQueue方法，
     * 判断一个原先属于条件队列的节点是否转移到了同步队列。
     *
     * 因为同步队列中会用到节点的next域，取消节点的next也有值的话，
     * 可以断言next域有值的节点一定在同步队列上。
     *
     * 在GC层面，和设置为null具有相同的效果。
     */
    node.next = node;
}
}

```

- 获取当前节点的prev节点
- 如果prev节点为head节点，那么它就有资格去争抢锁，调用tryAcquire抢占锁
- 抢占锁成功以后，把获得锁的节点设置为head，并且移除原来的初始化head节点
- 如果获得锁失败，则根据waitStatus决定是否需要挂起线程
- 最后，通过cancelAcquire取消获得锁的操作

## 7.shouldParkAfterFailedAcquire

从上面的分析可以看出，只有队列的第二个节点可以有机会争用锁，如果成功获取锁，则此节点晋升为头节点。对于第三个及以后的节点，`if (p == head)`条件不成立，首先进行`shouldParkAfterFailedAcquire(p, node)`操作。  
`shouldParkAfterFailedAcquire`方法是判断一个**争用锁的线程是否应该被阻塞**。它首先判断一个节点的前置节点的状态是否为`Node.SIGNAL`，如果是，是说明此节点已经将状态设置为：如果锁释放，则应当通知它，所以它可以安全的阻塞了，返回`true`。

```

private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    int ws = pred.waitStatus; //前继节点的状态
    if (ws == Node.SIGNAL) //如果是SIGNAL状态，意味着当前线程后面需要被unpark唤醒，所以挂起当前线程
        return true;
    //如果前节点的状态大于0，即为CANCELLED状态时，则会从前节点开始逐步循环找到一个没有
    //被“CANCELLED”节点设置为当前节点的前节点，返回false。在下次循环执行
    //shouldParkAfterFailedAcquire时，返回true。这个操作实际是把队列中CANCELLED的节点剔除掉。
    if (ws > 0) {
        do {
            node.prev = pred = pred.prev;

```

```

        } while (pred.waitStatus > 0);
        pred.next = node;
    } else { // 如果前继节点为“0”或者“共享锁”状态，则设置前继节点为SIGNAL状态。自己这次不挂起
        /*
         * waitStatus must be 0 or PROPAGATE. Indicate that we
         * need a signal, but don't park yet. Caller will need to
         * retry to make sure it cannot acquire before parking.
         */
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}

```

如果shouldParkAfterFailedAcquire返回了true，则会执行：parkAndCheckInterrupt()方法，它是通过LockSupport.park(this)将当前线程挂起到WAITING状态，它需要等待一个中断、unpark方法来唤醒它，通过这样一种FIFO的机制的等待，来实现了Lock的操作。

```

private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);
    return Thread.interrupted();
}

```

注：Thread.interrupted()方法在返回中断标记的同时会清除中断标记，也就是说当由于中断醒来然后获取锁成功，那么整个acquireQueued方法就会返回true表示是因为中断醒来，但如果中断醒来以后没有获取到锁，继续挂起，由于这次的中断已经被清除了，下次如果是被正常唤醒，那么acquireQueued方法就会返回false，表示没有中断。

unpark函数为线程提供“许可(permit)”，线程调用park函数则等待“许可”。这个有点像信号量，但是这个“许可”是不能叠加的，“许可”是一次性的。

permit相当于0/1的开关，默认是0，调用一次unpark就加1变成了1.调用一次park会消费permit，又会变成0。如果再调用一次park会阻塞，因为permit已经是0了。直到permit变成1.这时调用unpark会把permit设置为1.每个线程都有一个相关的permit，permit最多只有一个，重复调用unpark不会累积

## 8.ReentrantLock.unlock

加锁的过程分析完以后，再来分析一下释放锁的过程，调用release方法，这个方法里面做两件事，1，释放锁；2，唤醒park的线程（就是AQS里的代码）

```

public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}

```

tryRelease这个动作可以认为就是一个设置锁状态的操作，而且是将状态减掉传入的参数值（参数是1），如果结果状态为0，就将排它锁的Owner设置为null，以使得其它的线程有机会进行执行。在排它锁中，加锁的时候状态会增加1（当然可以自己修改这个值），在解锁的时候减掉1，同一个锁，在可以重入后，可能会被叠加为2、3、4这些值，只有unlock()的次数与lock()的次数对应才会将Owner线程设置为空，而且也只有这种情况下才会返回true。

```

protected final boolean tryRelease(int releases) {
    int c = getState() - releases; // 这里是将锁的数量减1
    if (Thread.currentThread() != getExclusiveOwnerThread()) // 如果释放的线程和获取
    锁的线程不是同一个，抛出非法监视器状态异常
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        // 由于重入的关系，不是每次释放锁c都等于0，
        // 直到最后一次释放锁时，才会把当前线程释放
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}

```

在方法unparkSuccessor(Node)中，就意味着真正要释放锁了，它传入的是head节点 (head节点是占用锁的节点)，当前线程被释放之后，需要唤醒下一个节点的线程。

```

private void unparkSuccessor(Node node) {
    int ws = node.waitStatus;
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0);
    Node s = node.next;
    if (s == null || s.waitStatus > 0) { // 判断后继节点是否为空或者是否是取消状态，
        s = null;
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0) // 然后从队列尾部向前遍历找到最前面的一个waitStatus小
                于0的节点，至于为什么从尾部开始向前遍历，因为在doAcquireInterruptibly.cancelAcquire方法
                的处理过程中只设置了next的变化，没有设置prev的变化，在最后有这样一行代码：node.next = node,
                如果这时执行了unparkSuccessor方法，并且向后遍历的话，就成了死循环了，所以这时只有prev是稳定的
                s = t;
    }
    // 内部首先会发生的动作是获取head节点的next节点，如果获取到的节点不为空，则直接通
    过：“LockSupport.unpark()”方法来释放对应的被挂起的线程，这样一来将会有一个节点唤醒后继续进入
    循环进一步尝试tryAcquire()方法来获取锁
    if (s != null)
        LockSupport.unpark(s.thread); // 释放许可
}

```

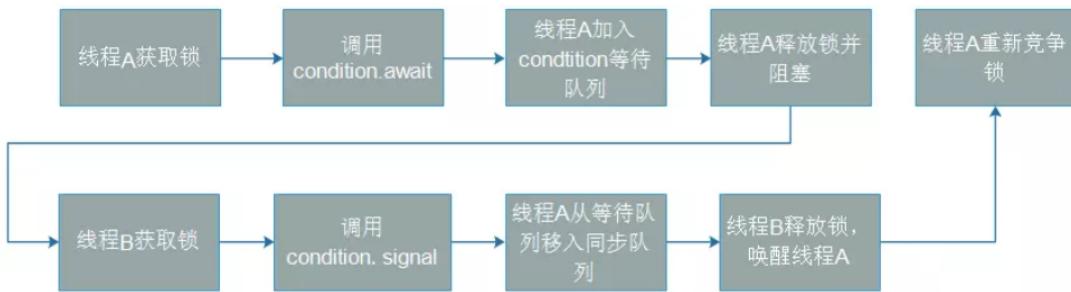
## 9.补充说明

第一个执行的线程A为独占锁线程，但是此时双向队列的head和tail都是null。只有后续某个线程比如B来的时候，没拿到锁，才会new一个Node（在enq代码里），head和tail都指向它，然后自旋，把自己封装成BNode加在这个Node后面，此时head依旧指向Node，tail指向BNode。所以A释放锁的时候，要获取head，然后唤醒head的next里的线程。

## 10.condition梳理

- 1、Condition提供了await()方法将当前线程阻塞，并提供signal()方法支持另外一个线程将已经阻塞的线程唤醒。
- 2、Condition需要结合Lock使用
- 3、线程调用await()方法前必须获取锁，调用await()方法时，将线程构造成节点加入等待队列，同时释放锁，并挂起当前线程

4、其他线程调用signal()方法前也必须获取锁，当执行signal()方法时将等待队列的节点移入到同步队列，当线程退出临界区释放锁的时候，唤醒同步队列的首个节点



## 公平锁

## 共享锁

## CountDownLatch

CountDownLatch是同步工具类之一，可以指定一个计数值，在并发环境下由线程进行减1操作，当计数值变为0之后，被await方法阻塞的线程将会唤醒，实现线程间的同步。

```
public class CountDownLatchTest implements Runnable {  
  
    private CountDownLatch countDownLatch;  
  
    CountDownLatchTest(CountDownLatch countDownLatch){  
        this.countDownLatch = countDownLatch;  
    }  
  
    @Override  
    public void run() {  
        System.out.println(Thread.currentThread().getName() + " start");  
        Random random = new Random();  
        try {  
            Thread.sleep(random.nextInt(10000) + 1000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println(Thread.currentThread().getName() + " finish");  
        countDownLatch.countDown();  
    }  
  
    public static void main(String[] args) {  
        CountDownLatch countDownLatch = new CountDownLatch(10);  
        for (int i = 0; i < 10; i++) {  
            new Thread(new CountDownLatchTest(countDownLatch), "Thread-" +  
i).start();  
        }  
        try {  
            countDownLatch.await();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println("all thread finish,main can do next job");  
    }  
}
```

```
    }  
}
```

## 原理

## 使用场景

实现最大的并行性：有时我们想同时启动多个线程，实现最大程度的并行性。例如，我们想测试一个单例类。  
\* 如果我们创建一个初始计数为1的CountDownLatch，并让所有线程都在这个锁上等待，那么我们可以很轻松地完成测试。  
\* 我们只需调用一次countDown()方法就可以让所有的等待线程同时恢复执行。

开始执行前等待n个线程完成各自任务：例如应用程序启动类要确保在处理用户请求前，所有N个外部系统已经启动和运行了。

## CyclicBarrier

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

```
public class CyclicBarrierTest implements Runnable {  
  
    private CyclicBarrier cyclicBarrier;  
  
    CyclicBarrierTest(CyclicBarrier cyclicBarrier) {  
        this.cyclicBarrier = cyclicBarrier;  
    }  
  
    @Override  
    public void run() {  
        try {  
            Thread.sleep(1000);  
            System.out.println(Thread.currentThread().getName() + " 到达栅栏 A");  
            cyclicBarrier.await();  
            System.out.println(Thread.currentThread().getName() + " 冲破栅栏 A");  
  
            Thread.sleep(2000);  
            System.out.println(Thread.currentThread().getName() + " 到达栅栏 B");  
            cyclicBarrier.await();  
            System.out.println(Thread.currentThread().getName() + " 冲破栅栏 B");  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
  
    public static void main(String[] args) {  
        // 5个线程参与，最后一个到达的线程执行最后的任务  
        CyclicBarrier cyclicBarrier = new CyclicBarrier(5, () ->  
            System.out.println(Thread.currentThread().getName() + " 来完成最后任务"));  
        for (int i = 0; i < 5; i++) {  
            new Thread(new CyclicBarrierTest(cyclicBarrier), "Thread-" +  
                i).start();  
        }  
    }  
}
```

有五个人，一个裁判。这五个人同时跑，裁判开始计时，五个人都到终点了，裁判喊停，然后统计这五个人从开始跑到最后一个撞线用了多长时间，这是CountDownLatch。继续，还是这五个人（这五个人真无聊..），这次没裁判。规定五个人只要都跑到终点了，大家可以喝啤酒。但是，只要有一个人没到终点，就不能喝。CyclicBarrier强调的是n个线程，大家相互等待，只要有一个没完成，所有人都得等着。正如上例，只有5个人全部跑到终点，大家才能开喝，否则只能全等着。

场景：可以用于多线程计算数据，最后合并计算结果的场景

## 原理

### CountDownLatch和CyclicBarrier区别

CountDownLatch	CyclicBarrier
减计数方式	加计数方式
计算为0时释放所有等待的线程	计数达到指定值时释放所有等待线程
计数为0时，无法重置	计数达到指定值时，计数置为0重新开始
调用countDown()方法计数减一，调用await()方法只进行阻塞，对计数没任何影响	调用await()方法计数加1，若加1后的值不等于构造方法的值，则线程阻塞
不可重复利用	可重复利用

## Semaphore

对于有限资源的循环使用，例如商场厕所、停车场位置、限流处理

```
public class SemaphoreTest implements Runnable {

    private Semaphore semaphore;

    SemaphoreTest(Semaphore semaphore) {
        this.semaphore = semaphore;
    }

    @Override
    public void run() {
        // 使用信号量限流
        try {
            // 尝试获取许可成功
            if (semaphore.tryAcquire()) {
                System.out.println(Thread.currentThread().getName() + " 获得许可");
                // 业务操作
                Thread.sleep(10 * 1000);
                System.out.println(Thread.currentThread().getName() + " end, 释放许可");
                semaphore.release();
            } else {
                System.out.println(Thread.currentThread().getName() + " 降级处理: 抛弃");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

        //一般用法
    //    try {
    //        System.out.println(Thread.currentThread().getName() + " start");
    //        // 从信号量中获取许可
    //        semaphore.acquire();
    //        System.out.println(Thread.currentThread().getName() + " 获得许可");
    //        Thread.sleep(10 * 1000);
    //        System.out.println(Thread.currentThread().getName() + " end,释放许
    可");
    //        semaphore.release();
    //    } catch (Exception e) {
    //        e.printStackTrace();
    //    }
    }

    public static void main(String[] args) {
        // 信号量，初始数量是2个，也就是共享资源总共只有2个，先进先出公平模式
        Semaphore semaphore = new Semaphore(2, true);
        for (int i = 0; i < 5; i++) {
            new Thread(new SemaphoreTest(semaphore), "Thread-" + i).start();
        }
    }
}

```

## 原理

# Exchanger

Exchanger 是 JDK 1.5 开始提供的一个用于两个工作线程之间交换数据的封装工具类，简单说就是一个线程在完成一定的事务后想与另一个线程交换数据，则第一个先拿出数据的线程会一直等待第二个线程，直到第二个线程拿着数据到来时才能彼此交换对应数据。

其定义为 Exchanger 泛型类型，其中 V 表示可交换的数据类型，对外提供的接口很简单，具体如下：

- `Exchanger():` 无参构造方法。
- `V exchange(V v):` 等待另一个线程到达此交换点（除非当前线程被中断），然后将给定的对象传送给该线程，并接收该线程的对象。
- `V exchange(V v, long timeout, TimeUnit unit):` 等待另一个线程到达此交换点（除非当前线程被中断或超出了指定的等待时间），然后将给定的对象传送给该线程，并接收该线程的对象。

可以看出，当一个线程到达 `exchange` 调用点时，如果其他线程此前已经调用了此方法，则其他线程会被调度唤醒并与之进行对象交换，然后各自返回；如果其他线程还没到达交换点，则当前线程会被挂起，直至其他线程到达才会完成交换并正常返回，或者当前线程被中断或超时返回。

```

public class ExchangerTest {
    // 生产者
    static class Producer implements Runnable {

        private Exchanger<String> exchanger;

        Producer(Exchanger<String> exchanger) {
            this.exchanger = exchanger;
        }
    }
}

```

```

@Override
public void run() {
    for (int i = 1; i < 5; i++) {
        try {
            Thread.sleep(2000);
            String data = String.valueOf(i);
            System.out.println(Thread.currentThread().getName() + " 交换
前:" + data);
            data = exchanger.exchange(data);
            System.out.println(Thread.currentThread().getName() + " 交换
后:" + data);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

// 消费者
static class Consumer implements Runnable {
    private Exchanger<String> exchanger;

    Consumer(Exchanger<String> exchanger) {
        this.exchanger = exchanger;
    }

    @Override
    public void run() {
        while (true) {
            String data = "a";
            System.out.println(Thread.currentThread().getName() + " 交换前:"
+ data);
            try {
                data = exchanger.exchange(data);
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + " 交换后:"
+ data);
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    Exchanger<String> exchanger = new Exchanger<String>();
    new Thread(new Producer(exchanger), "Producer").start();
    new Thread(new Consumer(exchanger), "Consumer").start();
    Thread.sleep(7000);
    System.exit(-1);
}
}

```

## ThreadPoolExecutor

```
public enum State {  
    //线程刚创建  
    NEW,  
  
    //在JVM中正在运行的线程  
    RUNNABLE,  
  
    //线程处于阻塞状态，等待监视锁，可以重新进行同步代码块中执行  
    BLOCKED,  
  
    //等待状态  
    WAITING,  
  
    //调用sleep() join() wait()方法可能导致线程处于等待状态  
    TIMED_WAITING,  
  
    //线程执行完毕，已经退出  
    TERMINATED;  
}
```

### 如果你提交任务时，线程池队列已满，这时会发生什么

(1) 如果使用的是无界队列 LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为 LinkedBlockingQueue 可以近乎认为是一个无穷大的队列，可以无限存放任务

(2) 如果使用的是有界队列比如 ArrayBlockingQueue，任务首先会被添加到ArrayBlockingQueue 中，ArrayBlockingQueue 满了，会根据maximumPoolSize 的值增加线程数量，如果增加了线程数量还是处理不过来，ArrayBlockingQueue 继续满，那么则会使用拒绝策略RejectedExecutionHandler 处理满了的任务，默认是 AbortPolicy

- 如何核心线程数满了，但是阻塞队列没有满的话，就会将该线程先放入阻塞队列中
- 如果核心线程和阻塞队列都满了，但是最大线程数没有满的话，就会新建一个非核心线程去执行该任务

## java IO

### 零拷贝

#### 概念

零拷贝(Zero-copy)技术指在计算机执行操作时，CPU 不需要先将数据从一个内存区域复制到另一个内存区域，从而可以减少上下文切换以及 CPU 的拷贝时间。

它的作用是在数据报从网络设备到用户程序空间传递的过程中，减少数据拷贝次数，减少系统调用，实现 CPU 的零参与，彻底消除 CPU 在这方面的负载。

实现零拷贝用到的最主要技术是 DMA 数据传输技术和内存区域映射技术：

- 零拷贝机制可以减少数据在内核缓冲区和用户进程缓冲区之间反复的 I/O 拷贝操作。
- 零拷贝机制可以减少用户进程地址空间和内核地址空间之间因为上下文切换而带来的 CPU 开销。

### 物理内存和虚拟内存

由于操作系统的进程与进程之间是共享 CPU 和内存资源的，因此需要一套完善的内存管理机制防止进程之间内存泄漏的问题。

为了更加有效地管理内存并减少出错，现代操作系统提供了一种对主存的抽象概念，即虚拟内存(Virtual Memory)。

虚拟内存为每个进程提供了一个一致的、私有的地址空间，它让每个进程产生了一种自己在独享主存的错觉(每个进程拥有一片连续完整的内存空间)。

**物理内存**(Physical Memory)是相对于虚拟内存(Virtual Memory)而言的。

物理内存指通过物理内存条而获得的内存空间，而虚拟内存则是指将硬盘的一块区域划分来作为内存。内存主要作用是在计算机运行时为操作系统和各种程序提供临时储存。

在应用中，自然是顾名思义，物理上，真实存在的插在主板内存槽上的内存条的容量的大小。

**虚拟内存**是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续的可用的内存(一个连续完整的地址空间)。

而实际上，**虚拟内存通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换，加载到物理内存中来。**

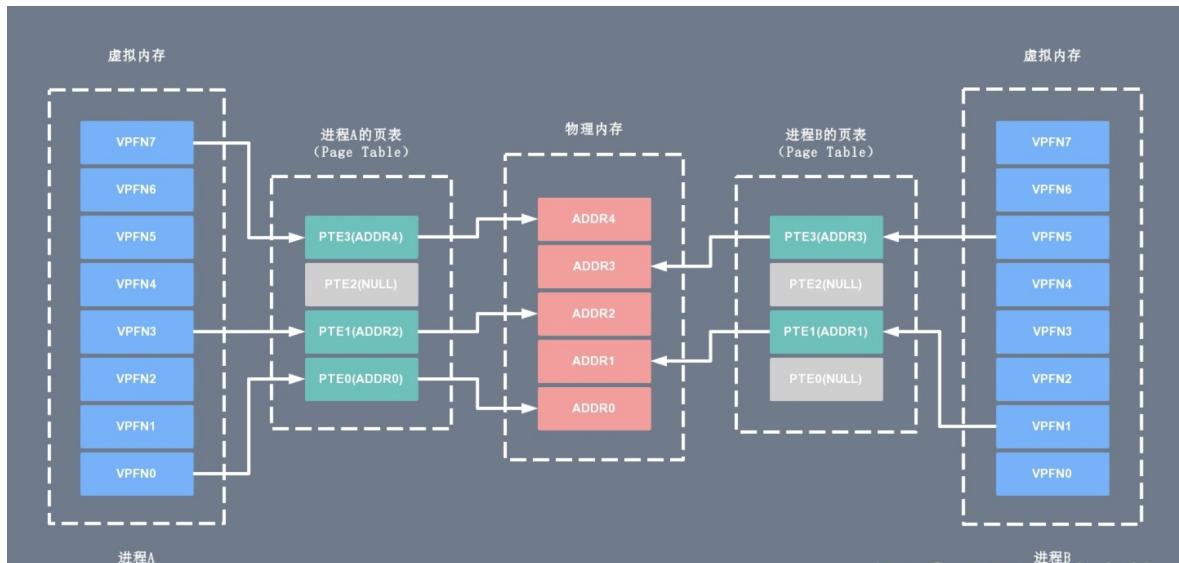
目前，大多数操作系统都使用了虚拟内存，如 Windows 系统的虚拟内存、Linux 系统的交换空间等等。

**虚拟内存地址**和用户进程紧密相关，一般来说不同进程里的同一个虚拟地址指向的物理地址是不一样的，所以离开进程谈虚拟内存没有任何意义。每个进程所能使用的虚拟地址大小和 CPU 位数有关。

在 32 位的系统上，虚拟地址空间大小是  $2^{32}=4G$ ，在 64 位系统上，虚拟地址空间大小是  $2^{64}=16G$ ，而实际的物理内存可能远远小于虚拟内存的大小。

每个用户进程维护了一个单独的**页表(Page Table)**，**虚拟内存和物理内存就是通过这个页表实现地址空间的映射的。**

下面给出两个进程 A、B 各自的虚拟内存空间以及对应的物理内存之间的地址映射示意图：



当进程执行一个程序时，需要先从内存中读取该进程的指令，然后执行，获取指令时用到的就是虚拟地址。

这个虚拟地址是程序链接时确定的(内核加载并初始化进程时会调整动态库的地址范围)。

为了获取到实际的数据，CPU 需要将虚拟地址转换成物理地址，CPU 转换地址时需要用到进程的页表(Page Table)，而页表(Page Table)里面的数据由操作系统维护。

其中页表(Page Table)可以简单的理解为单个内存映射(Memory Mapping)的链表(当然实际结构很复杂)。

页表里面的每个内存映射(Memory Mapping)都将一块虚拟地址映射到一个特定的地址空间(**物理内存或者磁盘存储空间**)。

每个进程拥有自己的页表(Page Table)，和其他进程的页表(Page Table)没有关系。

通过上面的介绍，我们可以简单的将用户进程申请并访问物理内存(或磁盘存储空间)的过程总结如下：

1. 用户进程向操作系统发出内存申请请求。
2. 系统会检查进程的虚拟地址空间是否被用完，如果有剩余，给进程分配虚拟地址。
3. 系统为这块虚拟地址创建内存映射(Memory Mapping)，并将它放进该进程的页表(Page Table)。
4. 系统返回虚拟地址给用户进程，用户进程开始访问该虚拟地址。
5. CPU 根据虚拟地址在此进程的页表(Page Table)中找到了相应的内存映射(Memory Mapping)，但是这个内存映射(Memory Mapping)没有和物理内存关联，于是产生缺页中断。
6. 操作系统收到缺页中断后，分配真正的物理内存并将它关联到页表相应的内存映射(Memory Mapping)。中断处理完成后，CPU 就可以访问内存了
7. 当然缺页中断不是每次都会发生，只有系统觉得有必要延迟分配内存的时候才用的着，也即很多时候在上面的第 3 步系统会分配真正的物理内存并和内存映射(Memory Mapping)进行关联。

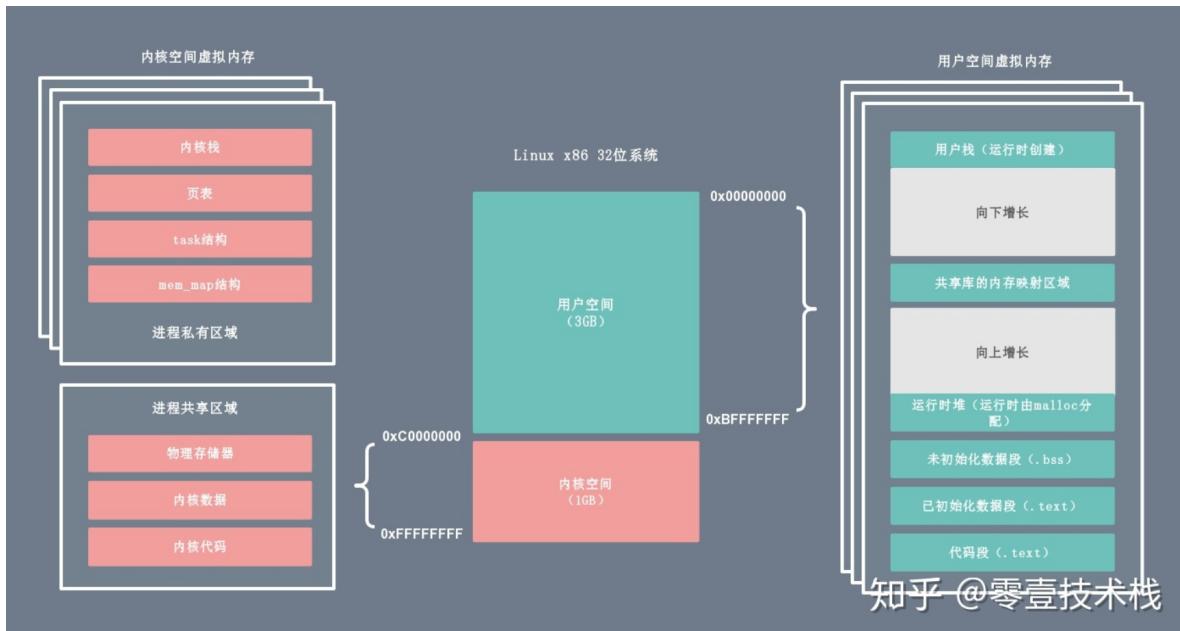
在用户进程和物理内存(磁盘存储器)之间引入虚拟内存主要有以下的优点：

- 地址空间：提供更大的地址空间，并且地址空间是连续的，使得程序编写、链接更加简单。
- 进程隔离：不同进程的虚拟地址之间没有关系，所以一个进程的操作不会对其他进程造成影响。
- 数据保护：每块虚拟内存都有相应的读写属性，这样就能保护程序的代码段不被修改，数据块不能被执行等，增加了系统的安全性。
- 内存映射：**有了虚拟内存之后，可以直接映射磁盘上的文件(可执行文件或动态库)到虚拟地址空间。这样可以做到物理内存延时分配，只有在需要读相应的文件的时候，才将它真正的从磁盘上加载到内存中来，而在内存吃紧的时候又可以将这部分内存清空掉，提高物理内存利用效率，并且所有这些对应用程序都是透明的。**
- 共享内存：比如动态库只需要在内存中存储一份，然后将它映射到不同进程的虚拟地址空间中，让进程觉得自己独占了这个文件。进程间的内存共享也可以通过映射同一块物理内存到进程的不同虚拟地址空间来实现共享
- 物理内存管理：物理地址空间全部由操作系统管理，进程无法直接分配和回收，从而系统可以更好的利用内存，平衡进程间对内存的需求

## 内核空间和用户空间

操作系统的**核心是内核**，独立于普通的应用程序，可以访问受保护的内存空间，也有访问底层硬件设备的权限。为了避免用户进程直接操作内核，保证内核安全，操作系统将**虚拟内存划分为两部分**，一部分是**内核空间 (Kernel-space)**，一部分是**用户空间 (User-space)**。在 Linux 系统中，内核模块运行在内核空间，对应的进程处于内核态；而用户程序运行在用户空间，对应的进程处于用户态。

内核进程和用户进程所占的虚拟内存比例是 1:3，而 Linux x86\_32 系统的寻址空间(虚拟存储空间)为 4G (2的32次方)，将最高的 1G 的字节(从虚拟地址 0xC0000000 到 0xFFFFFFFF)供内核进程使用，称为内核空间；而较低的 3G 的字节(从虚拟地址 0x00000000 到 0xBFFFFFFF)，供各个用户进程使用，称为用户空间。下图是一个进程的用户空间和内核空间的内存布局：



## 内核空间

内核空间总是驻留在内存中，它是为操作系统的内核保留的。应用程序是不允许直接在该区域进行读写或直接调用内核代码定义的函数的。

上图左侧区域为内核进程对应的虚拟内存，按访问权限可以分为进程私有和进程共享两块区域：

- 进程私有的虚拟内存：每个进程都有单独的内核栈、页表、task 结构以及 mem\_map 结构等。
- 进程共享的虚拟内存：属于所有进程共享的内存区域，包括物理存储器、内核数据和内核代码区域。

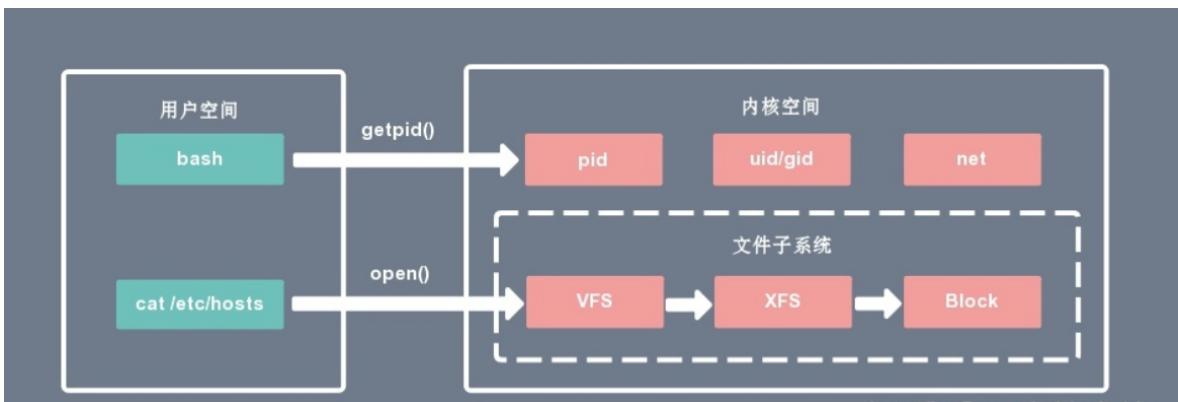
## 用户空间

每个普通的用户进程都有一个单独的用户空间，处于用户态的进程不能访问内核空间中的数据，也不能直接调用内核函数的，因此要进行**系统调用的时候，就要将进程切换到内核态才行**。用户空间包括以下几个内存区域：

- 运行时栈：由编译器自动释放，存放函数的参数值，局部变量和方法返回值等。每当一个函数被调用时，该函数的返回类型和一些调用的信息被存储到栈顶，调用结束后调用信息会被弹出并释放掉内存。栈区是从高地址位向低地址位增长的，是一块连续的内在区域，最大容量是由系统预先定义好的，申请的栈空间超过这个界限时会提示溢出，用户能从栈中获取的空间较小。
- 运行时堆：用于存放进程运行中被动态分配的内存段，位于 BSS 和栈中间的地址位。由卡发人员申请分配 (malloc) 和释放 (free)。堆是从低地址位向高地址位增长，采用链式存储结构。频繁地 malloc/free 造成内存空间的不连续，产生大量碎片。当申请堆空间时，库函数按照一定的算法搜索可用的足够大的空间。因此堆的效率比栈要低的多。
- 代码段：存放 CPU 可以执行的机器指令，该部分内存只能读不能写。通常代码区是共享的，即其它执行程序可调用它。假如机器中有数个进程运行相同的一个程序，那么它们就可以使用同一个代码段。
- 未初始化的数据段：存放未初始化的全局变量，BSS 的数据在程序开始执行之前被初始化为 0 或 NULL。
- 已初始化的数据段：存放已初始化的全局变量，包括静态全局变量、静态局部变量以及常量。
- **内存映射区域**：例如将动态库，共享内存等虚拟空间的内存映射到物理空间的内存，一般是 mmap 函数所分配的虚拟内存空间。

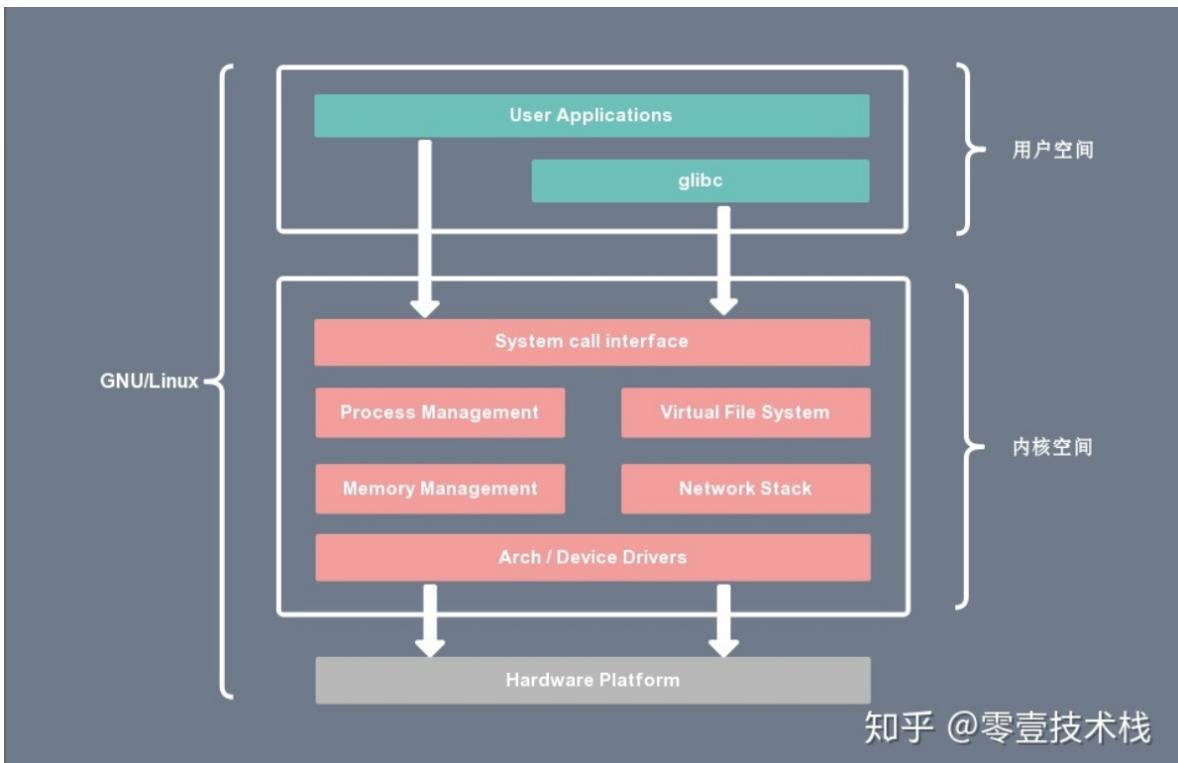
# Linux的内部层级结构

内核态可以执行任意命令，调用系统的一切资源，而用户态只能执行简单的运算，不能直接调用系统资源。用户态必须通过系统接口（System Call），才能向内核发出指令。比如，当用户进程启动一个 bash 时，它会通过 getpid() 对内核的 pid 服务发起系统调用，获取当前用户进程的 ID；当用户进程通过 cat 命令查看主机配置时，它会对内核的文件子系统发起系统调用。



- 内核空间可以访问所有的 CPU 指令和所有的内存空间、I/O 空间和硬件设备。
- 用户空间只能访问受限的资源，如果需要特殊权限，可以通过系统调用获取相应的资源。
- 用户空间允许页面中断，而内核空间则不允许。
- 内核空间和用户空间是针对线性地址空间的。
- x86 CPU 中用户空间是 0 - 3G 的地址范围，内核空间是 3G - 4G 的地址范围。x86\_64 CPU 用户空间地址范围为 0x0000000000000000 - 0x00007ffffffffff，内核地址空间为 0xffff880000000000 - 最大地址。
- **所有内核进程（线程）共用一个地址空间，而用户进程都有各自的地址空间。**

有了用户空间和内核空间的划分后，Linux 内部层级结构可以分为三部分，从最底层到最上层依次是硬件、内核空间和用户空间，如下图所示：

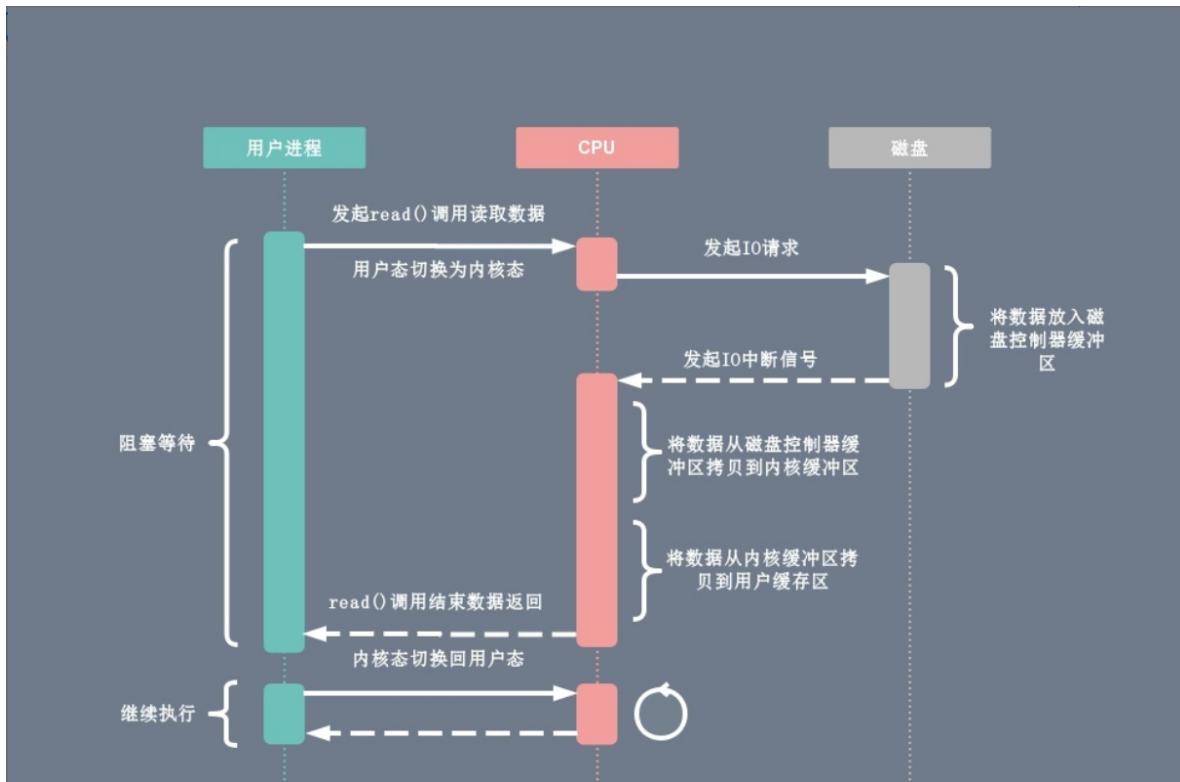


## Linux I/O 读写方式

Linux 提供了轮询、I/O 中断以及 DMA 传输这 3 种磁盘与主存之间的数据传输机制。其中轮询方式是基于死循环对 I/O 端口进行不断检测。I/O 中断方式是指当数据到达时，磁盘主动向 CPU 发起中断请求，由 CPU 自身负责数据的传输过程。DMA 传输则在 I/O 中断的基础上引入了 DMA 磁盘控制器，由 DMA 磁盘控制器负责数据的传输，降低了 I/O 中断操作对 CPU 资源的大量消耗。

## I/O中断原理

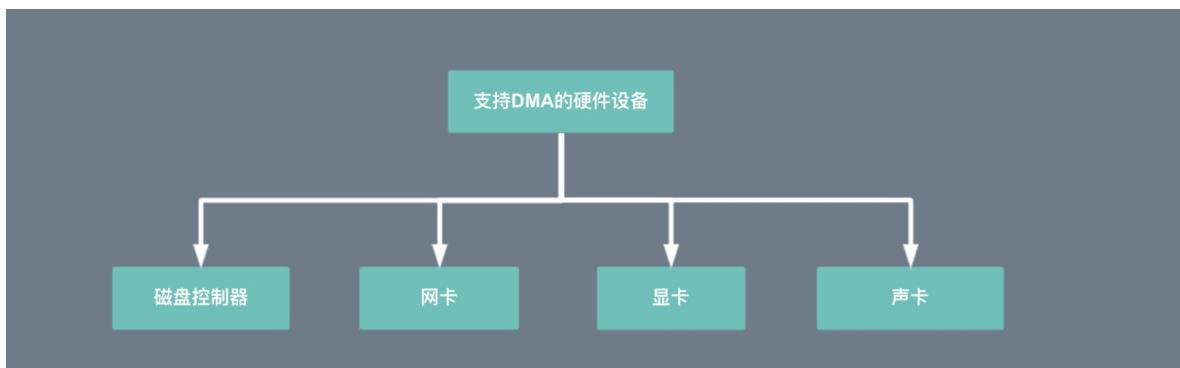
在 DMA 技术出现之前，应用程序与磁盘之间的 I/O 操作都是通过 CPU 的中断完成的。每次用户进程读取磁盘数据时，都需要 CPU 中断，然后发起 I/O 请求等待数据读取和拷贝完成，每次的 I/O 中断都导致 CPU 的上下文切换。



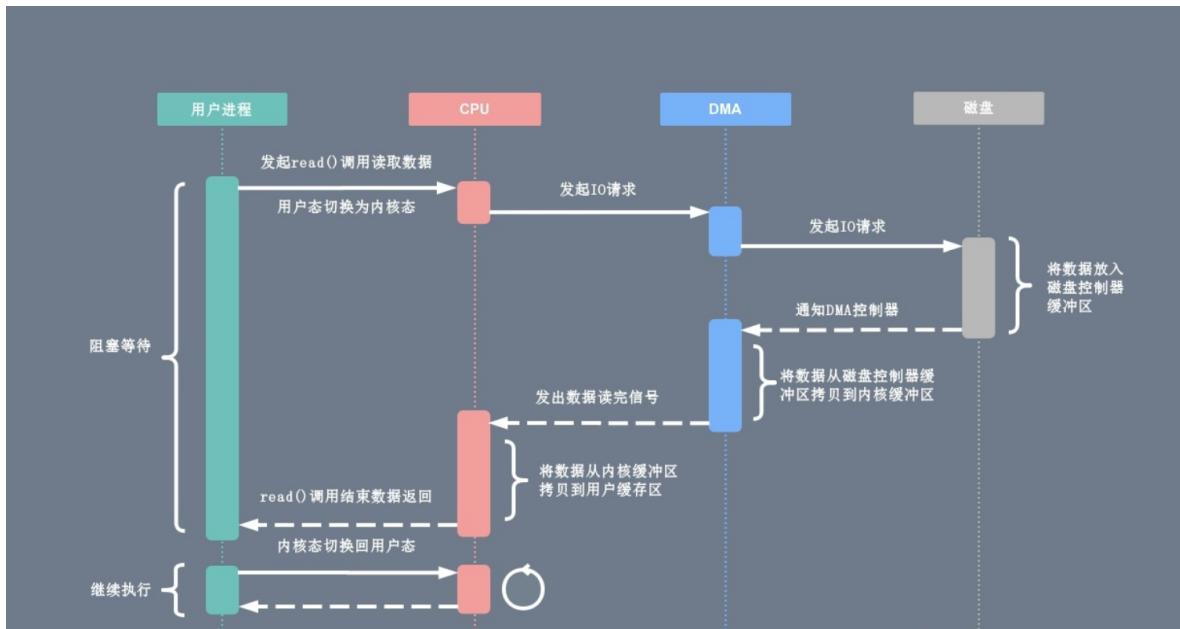
1. 用户进程向 CPU 发起 `read` 系统调用读取数据，由用户态切换为内核态，然后一直阻塞等待数据的返回。
2. CPU 在接收到指令以后对磁盘发起 I/O 请求，将磁盘数据先放入磁盘控制器缓冲区。
3. 数据准备完成以后，磁盘向 CPU 发起 I/O 中断。
4. CPU 收到 I/O 中断以后将磁盘缓冲区中的数据拷贝到内核缓冲区，然后再从内核缓冲区拷贝到用户缓冲区。
5. 用户进程由内核态切换回用户态，解除阻塞状态，然后等待 CPU 的下一个执行时间钟。

## DMA 传输原理

DMA 的全称叫**直接内存存取 (Direct Memory Access)**，是一种允许外围设备（硬件子系统）直接访问系统主内存的机制。也就是说，基于 DMA 访问方式，系统主内存于硬盘或网卡之间的数据传输可以绕开 CPU 的全程调度。目前大多数的硬件设备，包括磁盘控制器、网卡、显卡以及声卡等都支持 DMA 技术。



整个数据传输操作在一个 DMA 控制器的控制下进行的。CPU 除了在数据传输开始和结束时做一点处理外（开始和结束时候要做中断处理），在传输过程中 CPU 可以继续进行其他的工作。这样在大部分时间里，CPU 计算和 I/O 操作都处于并行操作，使整个计算机系统的效率大大提高。



有了 DMA 磁盘控制器接管数据读写请求以后，CPU 从繁重的 I/O 操作中解脱，数据读取操作的流程如下：

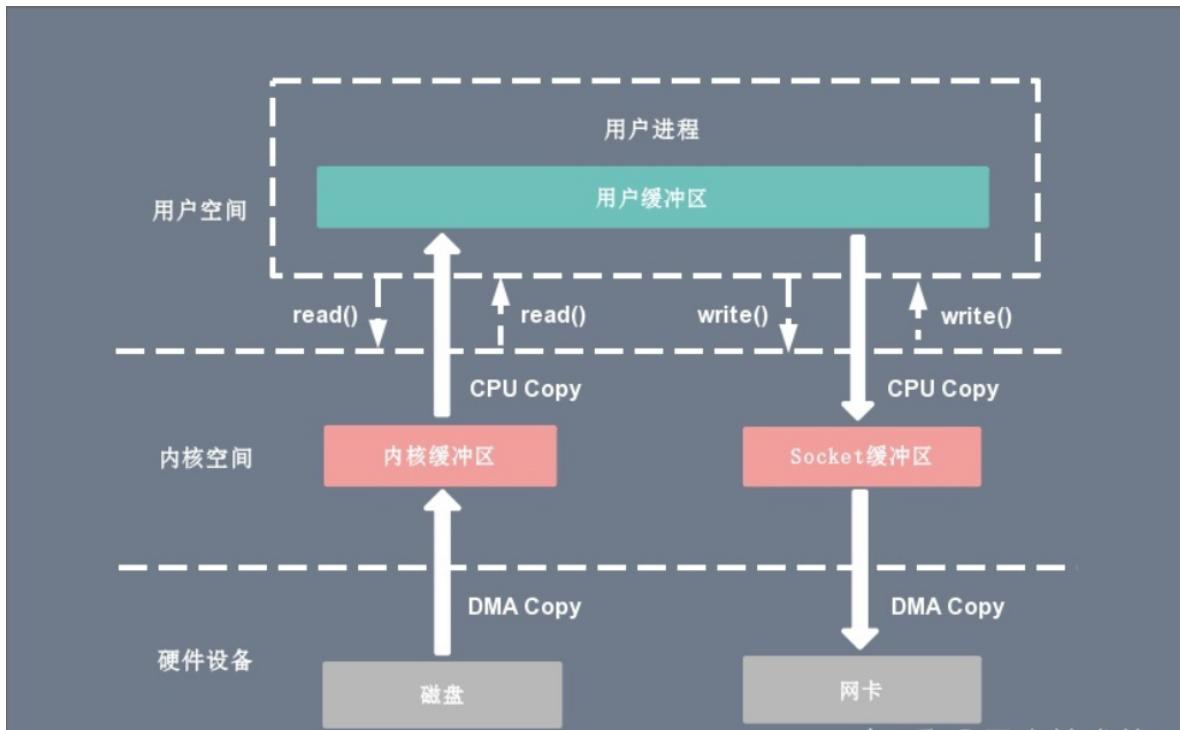
1. 用户进程向 CPU 发起 read 系统调用读取数据，由用户态切换为内核态，然后一直阻塞等待数据的返回。
2. CPU 在接收到指令以后对 DMA 磁盘控制器发起调度指令。
3. DMA 磁盘控制器对磁盘发起 I/O 请求，将磁盘数据先放入磁盘控制器缓冲区，CPU 全程不参与此过程。
4. 数据读取完成后，DMA 磁盘控制器会接受到磁盘的通知，将数据从磁盘控制器缓冲区拷贝到内核缓冲区。
5. DMA 磁盘控制器向 CPU 发出数据读完的信号，由 CPU 负责将数据从内核缓冲区拷贝到用户缓冲区。
6. 用户进程由内核态切换回用户态，解除阻塞状态，然后等待 CPU 的下一个执行时间钟。

## 传统I/O方式

为了更好的理解零拷贝解决的问题，我们首先了解一下传统 I/O 方式存在的问题。在 Linux 系统中，传统的访问方式是通过 write() 和 read() 两个系统调用实现的，通过 read() 函数读取文件到到缓存区中，然后通过 write() 方法把缓存中的数据输出到网络端口，伪代码如下：

```
read(file_fd, tmp_buf, len);
write(socket_fd, tmp_buf, len);
```

下图分别对应传统 I/O 操作的数据读写流程，整个过程涉及 2 次 CPU 拷贝、2 次 DMA 拷贝总共 4 次拷贝，以及 4 次上下文切换，下面简单地阐述一下相关的概念。



- 上下文切换：当用户程序向内核发起系统调用时，CPU 将用户进程从用户态切换到内核态；当系统调用返回时，CPU 将用户进程从内核态切换回用户态。
- CPU拷贝：由 CPU 直接处理数据的传送，**数据拷贝时会一直占用 CPU 的资源**。
- DMA拷贝：由 CPU 向DMA磁盘控制器下达指令，让 DMA 控制器来处理数据的传送，数据传送完毕再把信息反馈给 CPU，从而减轻了 CPU 资源的占有率。

## 传统读操作

当应用程序执行 `read` 系统调用读取一块数据的时候，如果这块数据已经存在于用户进程的页内存中，就直接从内存中读取数据；如果数据不存在，则先将数据从磁盘加载数据到内核空间的读缓存（`read buffer`）中，再从读缓存拷贝到用户进程的页内存中。

```
read(file_fd, tmp_buf, len);
```

基于传统的 I/O 读取方式，`read` 系统调用会触发 2 次上下文切换，1 次 DMA 拷贝和 1 次 CPU 拷贝，发起数据读取的流程如下：

1. 用户进程通过 `read()` 函数向内核（kernel）发起系统调用，上下文从用户态（user space）切换为内核态（kernel space）。
2. CPU 利用 DMA 控制器将数据从主存或硬盘拷贝到内核空间（kernel space）的读缓冲区（`read buffer`）。
3. CPU 将读缓冲区（`read buffer`）中的数据拷贝到用户空间（user space）的用户缓冲区（user buffer）。
4. 上下文从内核态（kernel space）切换回用户态（user space），`read` 调用执行返回。

## 传统写操作

当应用程序准备好数据，执行 `write` 系统调用发送网络数据时，先将数据从用户空间的页缓存拷贝到内核空间的网络缓冲区（socket buffer）中，然后再将写缓存中的数据拷贝到网卡设备完成数据发送。

```
write(socket_fd, tmp_buf, len);
```

基于传统的 I/O 写入方式，`write()` 系统调用会触发 2 次上下文切换，1 次 CPU 拷贝和 1 次 DMA 拷贝，用户程序发送网络数据的流程如下

1. 用户进程通过 write() 函数向内核 (kernel) 发起系统调用，上下文从用户态 (user space) 切换为内核态 (kernel space)。
2. CPU 将用户缓冲区 (user buffer) 中的数据拷贝到内核空间 (kernel space) 的网络缓冲区 (socket buffer)。
3. CPU 利用 DMA 控制器将数据从网络缓冲区 (socket buffer) 拷贝到网卡进行数据传输。
4. 上下文从内核态 (kernel space) 切换回用户态 (user space)，write 系统调用执行返回。

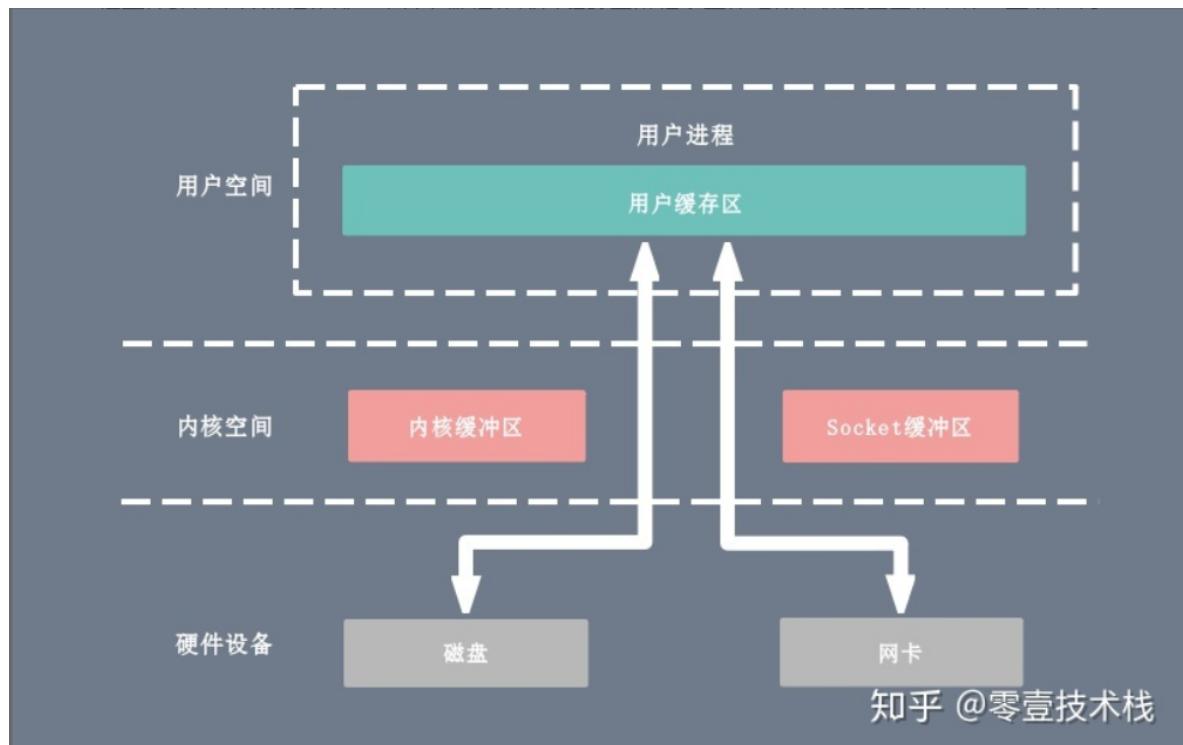
## 零拷贝方式

在 Linux 中零拷贝技术主要有 3 个实现思路：用户态直接 I/O、减少数据拷贝次数以及写时复制技术。

- 用户态直接 I/O：应用程序可以直接访问硬件存储，操作系统内核只是辅助数据传输。这种方式依旧存在用户空间和内核空间的上下文切换，硬件上的数据直接拷贝至了用户空间，不经过内核空间。因此，直接 I/O 不存在内核空间缓冲区和用户空间缓冲区之间的数据拷贝。
- 减少数据拷贝次数：在数据传输过程中，避免数据在用户空间缓冲区和系统内核空间缓冲区之间的 CPU 拷贝，以及数据在系统内核空间内的 CPU 拷贝，这也是当前主流零拷贝技术的实现思路。
- 写时复制技术：写时复制指的是当多个进程共享同一块数据时，如果其中一个进程需要对这份数据进行修改，那么将其拷贝到自己的进程地址空间中，如果只是数据读取操作则不需要进行拷贝操作。

## 用户态直接 I/O

用户态直接 I/O 使得应用进程或运行在用户态 (user space) 下的库函数直接访问硬件设备，数据直接跨过内核进行传输，内核在数据传输过程除了进行必要的虚拟存储配置工作之外，不参与任何其他工作，这种方式能够直接绕过内核，极大提高了性能。



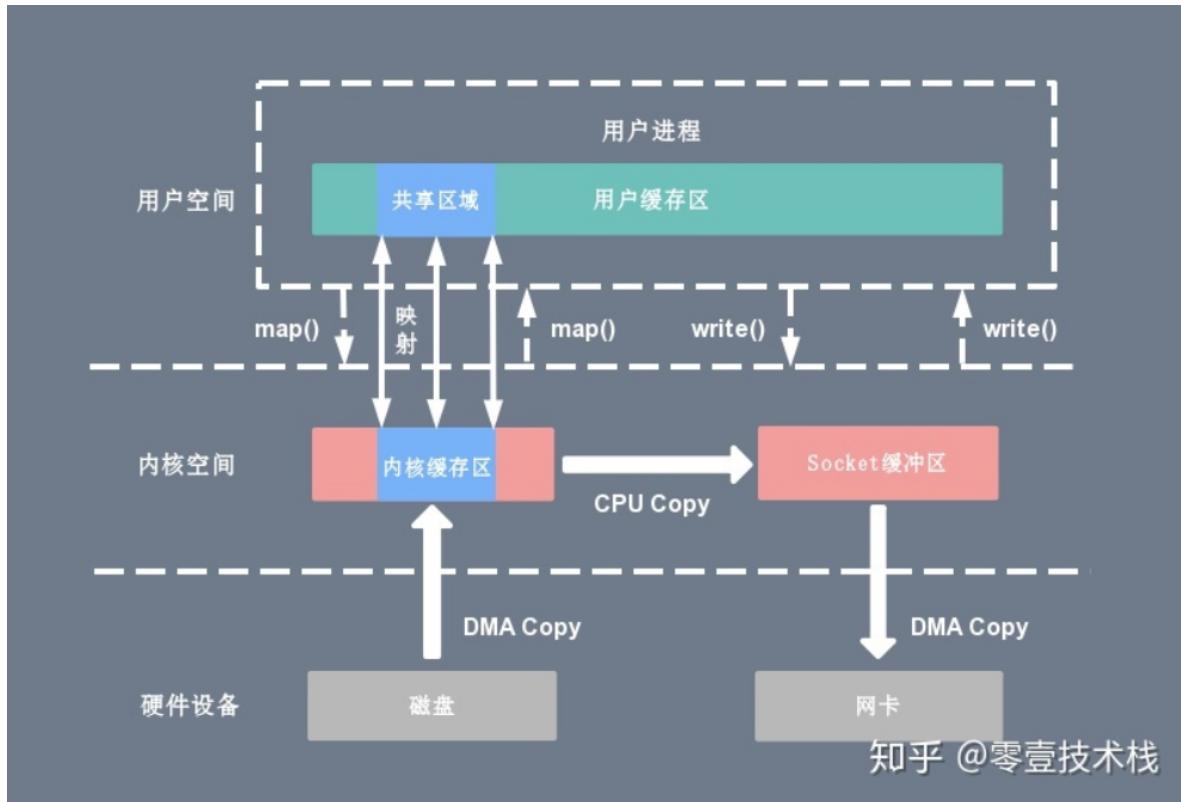
用户态直接 I/O 只能适用于不需要内核缓冲区处理的应用程序，这些应用程序通常在进程地址空间有自己的数据缓存机制，称为自缓存应用程序，如数据库管理系统就是一个代表。其次，这种零拷贝机制会直接操作磁盘 I/O，由于 CPU 和磁盘 I/O 之间的执行时间差距，会造成大量资源的浪费，解决方案是配合异步 I/O 使用。

## mmap + write

一种零拷贝方式是使用 mmap + write 代替原来的 read + write 方式，减少了 1 次 CPU 拷贝操作。mmap 是 Linux 提供的一种内存映射文件方法，即将一个进程的地址空间中的一段虚拟地址映射到磁盘文件地址，mmap + write 的伪代码如下：

```
tmp_buf = mmap(file_fd, len);
write(socket_fd, tmp_buf, len);
```

使用 mmap 的目的是将内核中读缓冲区 (read buffer) 的地址与用户空间的缓冲区 (user buffer) 进行映射，从而实现内核缓冲区与应用程序内存的共享，省去了将数据从内核读缓冲区 (read buffer) 拷贝到用户缓冲区 (user buffer) 的过程，然而内核读缓冲区 (read buffer) 仍需将数据到内核写缓冲区 (socket buffer)，大致的流程如下图所示：



基于 mmap + write 系统调用的零拷贝方式，整个拷贝过程会发生 4 次上下文切换，1 次 CPU 拷贝和 2 次 DMA 拷贝，用户程序读写数据的流程如下：

1. 用户进程通过 mmap() 函数向内核 (kernel) 发起系统调用，上下文从用户态 (user space) 切换为内核态 (kernel space)。
2. 将用户进程的内核空间的读缓冲区 (read buffer) 与用户空间的缓存区 (user buffer) 进行内存地址映射。
3. CPU 利用 DMA 控制器将数据从主存或硬盘拷贝到内核空间 (kernel space) 的读缓冲区 (read buffer)。
4. 上下文从内核态 (kernel space) 切换回用户态 (user space)，mmap 系统调用执行返回。
5. 用户进程通过 write() 函数向内核 (kernel) 发起系统调用，上下文从用户态 (user space) 切换为内核态 (kernel space)。
6. CPU 将读缓冲区 (read buffer) (因为内核读缓冲区和用户缓冲区映射了，是同一个内存地址，也就是共享) 中的数据拷贝到的网络缓冲区 (socket buffer)。
7. CPU 利用 DMA 控制器将数据从网络缓冲区 (socket buffer) 拷贝到网卡进行数据传输。
8. 上下文从内核态 (kernel space) 切换回用户态 (user space)，write 系统调用执行返回。

mmap 主要的用处是提高 I/O 性能，特别是针对大文件。对于小文件，内存映射文件反而会导致碎片空间的浪费，因为内存映射总是要对齐页边界，最小单位是 4 KB，一个 5 KB 的文件将会映射占用 8 KB 内存，也就会浪费 3 KB 内存。

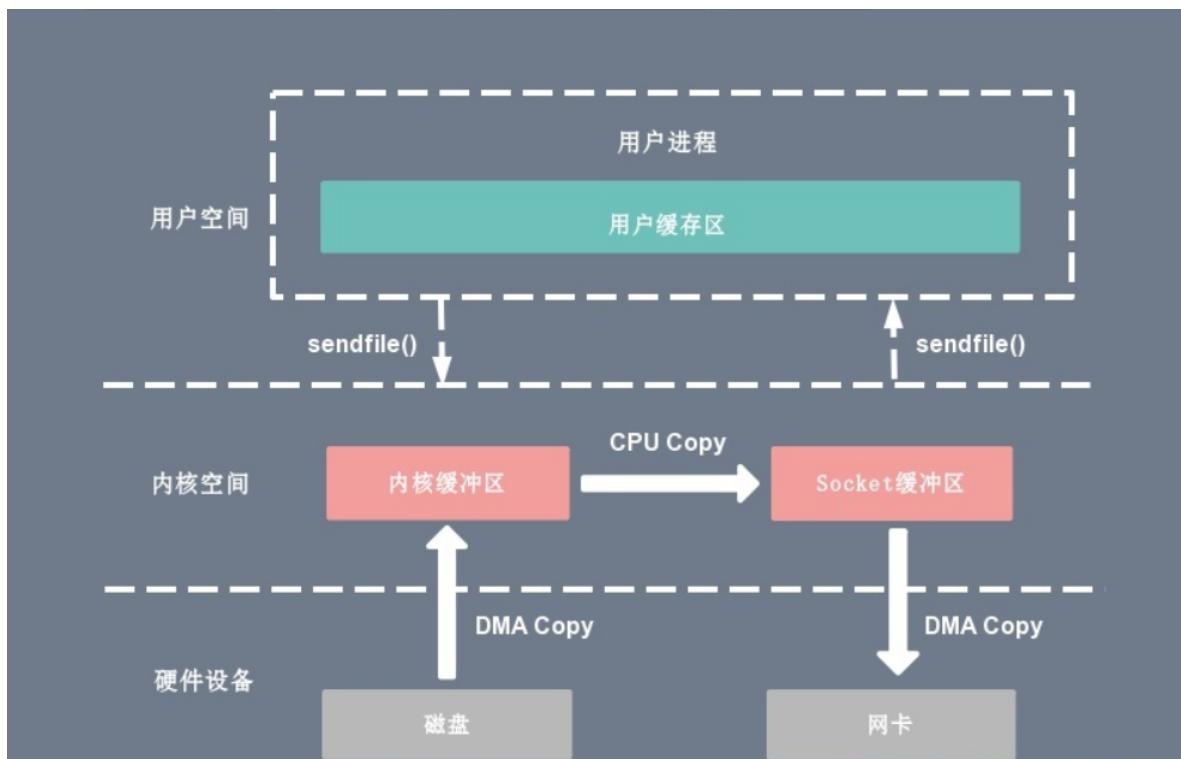
mmap 的拷贝虽然减少了 1 次拷贝，提升了效率，但也存在一些隐藏的问题。当 mmap 一个文件时，如果这个文件被另一个进程所截获，那么 write 系统调用会因为访问非法地址被 SIGBUS 信号终止，SIGBUS 默认会杀死进程并产生一个 coredump，服务器可能因此被终止。

## sendfile

sendfile 系统调用在 Linux 内核版本 2.1 中被引入，目的是简化通过网络在两个通道之间进行的数据传输过程。sendfile 系统调用的引入，不仅减少了 CPU 拷贝的次数，还减少了上下文切换的次数，它的伪代码如下：

```
sendfile(socket_fd, file_fd, len);
```

通过 sendfile 系统调用，数据可以直接在内核空间内部进行 I/O 传输，从而省去了数据在用户空间和内核空间之间的来回拷贝。与 mmap 内存映射方式不同的是，sendfile 调用中 I/O 数据对用户空间是完全不可见的。也就是说，这是一次完全意义上的**数据传输过程**。



基于 sendfile 系统调用的零拷贝方式，整个拷贝过程会发生 2 次上下文切换，1 次 CPU 拷贝和 2 次 DMA 拷贝，用户程序读写数据的流程如下：

1. 用户进程通过 sendfile() 函数向内核 (kernel) 发起系统调用，上下文从用户态 (user space) 切换为内核态 (kernel space)。
2. CPU 利用 DMA 控制器将数据从主存或硬盘拷贝到内核空间 (kernel space) 的读缓冲区 (read buffer)。
3. CPU 将读缓冲区 (read buffer) 中的数据拷贝到的网络缓冲区 (socket buffer)。
4. CPU 利用 DMA 控制器将数据从网络缓冲区 (socket buffer) 拷贝到网卡进行数据传输。
5. 上下文从内核态 (kernel space) 切换回用户态 (user space)，sendfile 系统调用执行返回。

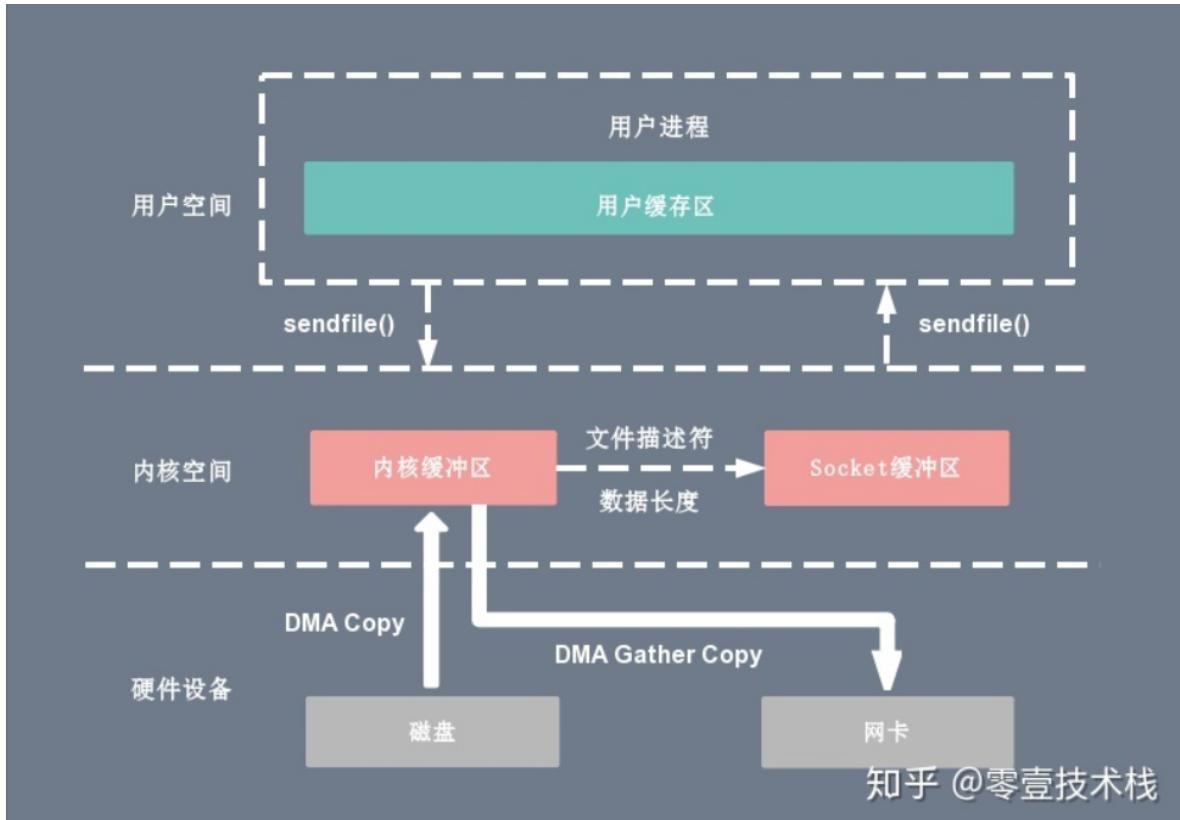
相比于 mmap 内存映射的方式，sendfile 少了 2 次上下文切换，但是仍然有 1 次 CPU 拷贝操作。sendfile 存在的问题是用户程序不能对数据进行修改，而只是单纯地完成了一次数据传输过程（从磁盘读出后没修改，直接给网卡了）。

## sendfile + DMA gather copy

Linux 2.4 版本的内核对 sendfile 系统调用进行修改，为 DMA 拷贝引入了 gather 操作。它将内核空间 (kernel space) 的读缓冲区 (read buffer) 中对应的数据描述信息 (内存地址、地址偏移量) 记录到相应的网络缓冲区 (socket buffer) 中，由 DMA 根据内存地址、地址偏移量将数据批量地从读缓冲区 (read buffer) 拷贝到网卡设备中，这样就省去了内核空间中仅剩的 1 次 CPU 拷贝操作，sendfile 的伪代码如下：

```
sendfile(socket_fd, file_fd, len);
```

在硬件的支持下，sendfile 拷贝方式不再从内核缓冲区的数据拷贝到 socket 缓冲区，取而代之的仅仅是缓冲区文件描述符和数据长度的拷贝，这样 DMA 引擎直接利用 gather 操作将页缓存中数据打包发送到网络中即可，本质就是和虚拟内存映射的思路类似。



基于 sendfile + DMA gather copy 系统调用的零拷贝方式，整个拷贝过程会发生 2 次上下文切换、0 次 CPU 拷贝以及 2 次 DMA 拷贝，用户程序读写数据的流程如下：

1. 用户进程通过 sendfile() 函数向内核 (kernel) 发起系统调用，上下文从用户态 (user space) 切换为内核态 (kernel space)。
2. CPU 利用 DMA 控制器将数据从主存或硬盘拷贝到内核空间 (kernel space) 的读缓冲区 (read buffer)。
3. CPU 把读缓冲区 (read buffer) 的文件描述符 (file descriptor) 和数据长度拷贝到网络缓冲区 (socket buffer)。
4. 基于已拷贝的文件描述符 (file descriptor) 和数据长度，CPU 利用 DMA 控制器的 gather/scatter 操作直接批量地将数据从内核的读缓冲区 (read buffer) 拷贝到网卡进行数据传输。
5. 上下文从内核态 (kernel space) 切换回用户态 (user space)，sendfile 系统调用执行返回。

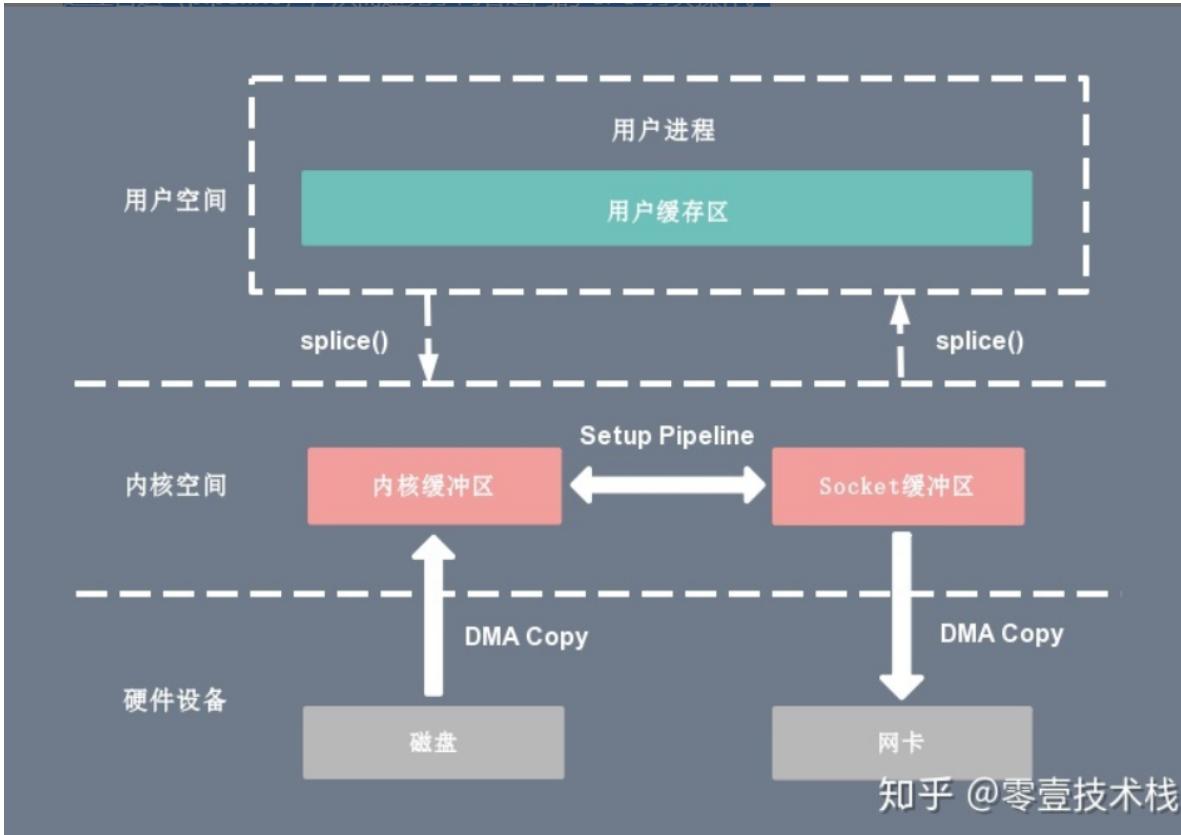
sendfile + DMA gather copy 拷贝方式同样存在用户程序不能对数据进行修改的问题，而且本身需要硬件的支持，它只适用于将数据从文件拷贝到 socket 套接字上的传输过程。

## splice

sendfile 只适用于将数据从文件拷贝到 socket 套接字上，同时需要硬件的支持，这也限定了它的使用范围。Linux 在 2.6.17 版本引入 splice 系统调用，不仅不需要硬件支持，还实现了两个文件描述符之间的数据零拷贝。splice 的伪代码如下：

```
splice(fd_in, off_in, fd_out, off_out, len, flags);
```

splice 系统调用可以在内核空间的读缓冲区 (read buffer) 和网络缓冲区 (socket buffer) 之间建立管道 (pipeline)，从而避免了两者之间的 CPU 拷贝操作。



基于 splice 系统调用的零拷贝方式，整个拷贝过程会发生 2 次上下文切换，0 次 CPU 拷贝以及 2 次 DMA 拷贝，用户程序读写数据的流程如下：

1. 用户进程通过 splice() 函数向内核 (kernel) 发起系统调用，上下文从用户态 (user space) 切换为内核态 (kernel space)。
2. CPU 利用 DMA 控制器将数据从主存或硬盘拷贝到内核空间 (kernel space) 的读缓冲区 (read buffer)。
3. CPU 在内核空间的读缓冲区 (read buffer) 和网络缓冲区 (socket buffer) 之间建立管道 (pipeline)。
4. CPU 利用 DMA 控制器将数据从网络缓冲区 (socket buffer) 拷贝到网卡进行数据传输。
5. 上下文从内核态 (kernel space) 切换回用户态 (user space)，splice 系统调用执行返回。

splice 拷贝方式也同样存在用户程序不能对数据进行修改的问题。除此之外，它使用了 Linux 的管道缓冲机制，可以用于任意两个文件描述符中传输数据，但是它的两个文件描述符参数中有一个必须是管道设备。

## 写时复制

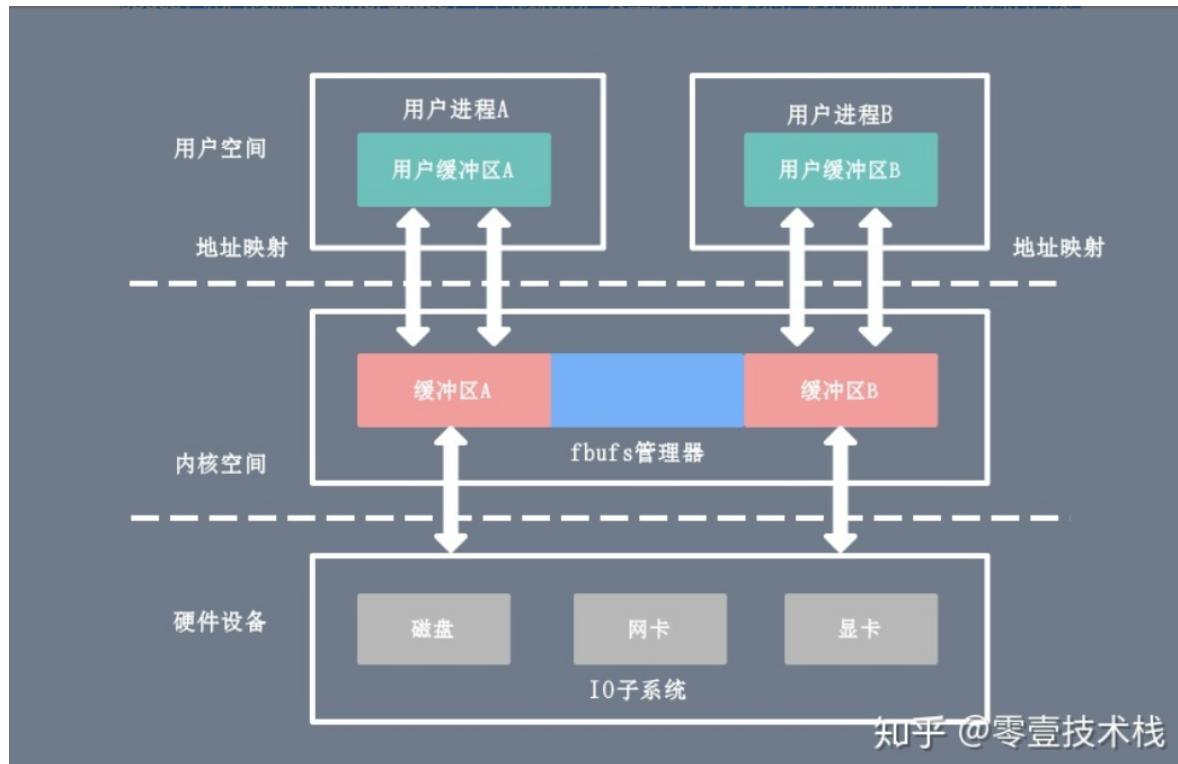
在某些情况下，内核缓冲区可能被多个进程所共享，如果某个进程想要这个共享区进行 write 操作，由于 write 不提供任何的锁操作，那么就会对共享区中的数据造成破坏，写时复制的引入就是 Linux 用来保护数据的。

写时复制指的是当多个进程共享同一块数据时，如果其中一个进程需要对这份数据进行修改，那么就需要将其拷贝到自己的进程地址空间中。这样做并不影响其他进程对这块数据的操作，每个进程要修改的时候才会进行拷贝，所以叫写时拷贝。这种方法在某种程度上能够降低系统开销，如果某个进程永远不会对所访问的数据进行更改，那么也就永远不需要拷贝。

## 缓冲区共享

缓冲区共享方式完全改写了传统的 I/O 操作，因为传统 I/O 接口都是基于数据拷贝进行的，要避免拷贝就得去掉原先的那套接口并重新改写，所以这种方法是比较全面的零拷贝技术，目前比较成熟的一个方案是在 Solaris 上实现的 fbuf (Fast Buffer，快速缓冲区)。

fbuf 的思想是每个进程都维护着一个缓冲区池，这个缓冲区池能被同时映射到用户空间 (user space) 和内核态 (kernel space)，内核和用户共享这个缓冲区池，这样就避免了一系列的拷贝操作。



缓冲区共享的难度在于管理共享缓冲区池需要应用程序、网络软件以及设备驱动程序之间的紧密合作，而且如何改写 API 目前还处于试验阶段并不成熟。

## Linux零拷贝对比

无论是传统 I/O 拷贝方式还是引入零拷贝的方式，2 次 DMA Copy 是都少不了的，因为两次 DMA 都是依赖硬件完成的。下面从 CPU 拷贝次数、DMA 拷贝次数以及系统调用几个方面总结一下上述几种 I/O 拷贝方式的差别。

拷贝方式	CPU拷贝	DMA拷贝	系统调用	上下文切换
传统方式 (read + write)	2	2	read / write	4
内存映射 (mmap + write)	1	2	mmap / write	4
sendfile	1	2	sendfile	2
sendfile + DMA gather copy	0	2	sendfile	2
splice	0	2	splice	2

## Java NIO零拷贝实现

在 Java NIO 中的通道 (Channel) 就相当于操作系统的内核空间 (kernel space) 的缓冲区，而缓冲区 (Buffer) 对应的相当于操作系统的用户空间 (user space) 中的用户缓冲区 (user buffer)。

- 通道 (Channel) 是全双工的 (双向传输)，它既可能是读缓冲区 (read buffer)，也可能是网络缓冲区 (socket buffer)。
- 缓冲区 (Buffer) 分为堆内存 (HeapBuffer) 和堆外内存 (DirectBuffer)，这是通过 malloc() 分配出来的用户态内存。

堆外内存 (DirectBuffer) 在使用后需要应用程序手动回收，而堆内存 (HeapBuffer) 的数据在 GC 时可能会被自动回收。因此，在使用 HeapBuffer 读写数据时，为了避免缓冲区数据因为 GC 而丢失，NIO 会先把 HeapBuffer 内部的数据拷贝到一个临时的 DirectBuffer 中的本地内存 (native memory)，这个拷贝涉及到 sun.misc.Unsafe.copyMemory() 的调用，背后的实现原理与 memcpy() 类似。最后，将临时生成的 DirectBuffer 内部的数据的内存地址传给 I/O 调用函数，这样就避免了再去访问 Java 对象处理 I/O 读写。

## MappedByteBuffer

MappedByteBuffer 是 NIO 基于内存映射 (mmap) 这种零拷贝方式的提供的一种实现，它继承自 ByteBuffer。FileChannel 定义了一个 map() 方法，它可以把一个文件从 position 位置开始的 size 大小的区域映射为内存映像文件。抽象方法 map() 方法在 FileChannel 中的定义如下：

```
public abstract MappedByteBuffer map(MapMode mode, long position, long size)
    throws IOException;
```

- mode：限定内存映射区域 (MappedByteBuffer) 对内存映像文件的访问模式，包括只可读 (READ\_ONLY)、可读可写 (READ\_WRITE) 和写时拷贝 (PRIVATE) 三种模式。
- position：文件映射的起始地址，对应内存映射区域 (MappedByteBuffer) 的首地址。
- size：文件映射的字节长度，从 position 往后的字节数，对应内存映射区域 (MappedByteBuffer) 的大小。

MappedByteBuffer 相比 ByteBuffer 新增了 fore()、load() 和 isLoad() 三个重要的方法：

- fore()：对于处于 READ\_WRITE 模式下的缓冲区，把对缓冲区内容的修改强制刷新到本地文件。
- load()：将缓冲区的内容载入物理内存中，并返回这个缓冲区的引用。
- isLoaded()：如果缓冲区的内容在物理内存中，则返回 true，否则返回 false。

下面给出一个利用 MappedByteBuffer 对文件进行读写的使用示例：

```
private final static String CONTENT = "Zero copy implemented by
MappedByteBuffer";
private final static String FILE_NAME = "/mmap.txt";
private final static String CHARSET = "UTF-8";
```

- 写文件数据：打开文件通道 fileChannel 并提供读权限、写权限和数据清空权限，通过 fileChannel 映射到一个可写的内存缓冲区 mappedByteBuffer，将目标数据写入 mappedByteBuffer，通过 force() 方法把缓冲区更改的内容强制写入本地文件。

```
@Test
public void writeToFileByMappedByteBuffer() {
    Path path = Paths.get(getClass().getResource(FILE_NAME).getPath());
    byte[] bytes = CONTENT.getBytes(Charset.forName(CHARSET));
    try (FileChannel filechannel = FileChannel.open(path,
        StandardOpenOption.READ,
        StandardOpenOption.WRITE, StandardOpenOption.TRUNCATE_EXISTING)) {
```

```

        MappedByteBuffer mappedByteBuffer = fileChannel.map(READ_WRITE, 0,
bytes.length);
        if (mappedByteBuffer != null) {
            mappedByteBuffer.put(bytes);
            mappedByteBuffer.force();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

- 读文件数据：打开文件通道 fileChannel 并提供只读权限，通过 fileChannel 映射到一个只可读的内存缓冲区 mappedByteBuffer，读取 mappedByteBuffer 中的字节数组即可得到文件数据。

```

@Test
public void readFromFileByMappedByteBuffer() {
    Path path = Paths.get(getClass().getResource(FILE_NAME).getPath());
    int length = CONTENT.getBytes(Charset.forName(CHARSET)).length;
    try (FileChannel fileChannel = FileChannel.open(path,
StandardOpenOption.READ)) {
        MappedByteBuffer mappedByteBuffer = fileChannel.map(READ_ONLY, 0,
length);
        if (mappedByteBuffer != null) {
            byte[] bytes = new byte[length];
            mappedByteBuffer.get(bytes);
            String content = new String(bytes, StandardCharsets.UTF_8);
            assertEquals(content, "Zero copy implemented by MappedByteBuffer");
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

下面介绍 map() 方法的底层实现原理。map() 方法是 java.nio.channels.FileChannel 的抽象方法，由子类 sun.nio.ch.FileChannelImpl.java 实现，下面是和内存映射相关的核心代码：

```

public MappedByteBuffer map(MapMode mode, long position, long size) throws
IOException {
    int pagePosition = (int)(position % allocationGranularity);
    long mapPosition = position - pagePosition;
    long mapSize = size + pagePosition;
    try {
        addr = map0(imode, mapPosition, mapSize);
    } catch (OutOfMemoryError x) {
        System.gc();
        try {
            Thread.sleep(100);
        } catch (InterruptedException y) {
            Thread.currentThread().interrupt();
        }
        try {
            addr = map0(imode, mapPosition, mapSize);
        } catch (outofMemoryError y) {
            throw new IOException("Map failed", y);
        }
    }
}

```

```

int isize = (int)size;
Unmapper um = new Unmapper(addr, mapsize, isize, mfd);
if ((!writable) || (imode == MAP_RO)) {
    return Util.newMappedByteBuffer(isize, addr + pagePosition, mfd, um);
} else {
    return Util.newMappedByteBuffer(isize, addr + pagePosition, mfd, um);
}
}

```

map() 方法通过本地方方法 map0() 为文件分配一块虚拟内存，作为它的内存映射区域，然后返回这块内存映射区域的起始地址。

1. 文件映射需要在 Java 堆中创建一个 MappedByteBuffer 的实例。如果第一次文件映射导致 OOM，则手动触发垃圾回收，休眠 100ms 后再尝试映射，如果失败则抛出异常。
2. 通过 Util 的 newMappedByteBuffer（可读可写）方法或者 newMappedByteBufferR（仅读）方法方法反射创建一个 DirectByteBuffer 实例，其中 DirectByteBuffer 是 MappedByteBuffer 的子类。

map() 方法返回的是内存映射区域的起始地址，通过（起始地址 + 偏移量）就可以获取指定内存的数据。这样一定程度上替代了 read() 或 write() 方法，底层直接采用 sun.misc.Unsafe 类的 getByte() 和 putByte() 方法对数据进行读写。

```

private native long map0(int prot, long position, long mapsize) throws
IOException;

```

上面是本地方方法（native method）map0 的定义，它通过 JNI（Java Native Interface）调用底层 C 的实现，这个 native 函数（Java\_sun\_nio\_ch\_FileChannelImpl\_map0）的实现位于 JDK 源码包下的 native/sun/nio/ch/FileChannelImpl.c 这个源文件里面。

```

JNIEXPORT jlong JNICALL
Java_sun_nio_ch_FileChannelImpl_map0(JNIEnv *env, jobject this,
                                         jint prot, jlong off, jlong len)
{
    void *mapAddress = 0;
    jobject fdo = (*env)->GetObjectField(env, this, chan_fd);
    jint fd = fdval(env, fdo);
    int protections = 0;
    int flags = 0;

    if (prot == sun_nio_ch_FileChannelImpl_MAP_RO) {
        protections = PROT_READ;
        flags = MAP_SHARED;
    } else if (prot == sun_nio_ch_FileChannelImpl_MAP_RW) {
        protections = PROT_WRITE | PROT_READ;
        flags = MAP_SHARED;
    } else if (prot == sun_nio_ch_FileChannelImpl_MAP_PV) {
        protections = PROT_WRITE | PROT_READ;
        flags = MAP_PRIVATE;
    }

    mapAddress = mmap64(
        0,                         /* Let OS decide location */
        len,                        /* Number of bytes to map */
        protections,                /* File permissions */
        flags,                      /* Changes are shared */

```

```

        fd,           /* File descriptor of mapped file */
        off);         /* offset into file */

    if (mapAddress == MAP_FAILED) {
        if (errno == ENOMEM) {
            JNU_ThrowOutOfMemoryError(env, "Map failed");
            return IOS_THROWN;
        }
        return handle(env, -1, "Map failed");
    }

    return ((jlong) (unsigned long) mapAddress);
}

```

可以看出 map0() 函数最终是通过 mmap64() 这个函数对 Linux 底层内核发出内存映射的调用，mmap64() 函数的原型如下：

```

#include <sys/mman.h>

void *mmap64(void *addr, size_t len, int prot, int flags, int fd, off64_t
offset);

```

下面详细介绍一下 mmap64() 函数各个参数的含义以及参数可选值：

- addr：文件在用户进程空间的内存映射区中的起始地址，是一个建议的参数，通常可设置为 0 或 NULL，此时由内核去决定真实的起始地址。当 flags 为 MAP\_FIXED 时，addr 就是一个必选的参数，即需要提供一个存在的地址。
- len：文件需要进行内存映射的字节长度
- prot：控制用户进程对内存映射区的访问权限
  - PROT\_READ：读权限
  - PROT\_WRITE：写权限
  - PROT\_EXEC：执行权限
  - PROT\_NONE：无权限
- flags：控制内存映射区的修改是否被多个进程共享
  - MAP\_PRIVATE：对内存映射区数据的修改不会反映到真正的文件，数据修改发生时采用写时复制机制
  - MAP\_SHARED：对内存映射区的修改会同步到真正的文件，修改对共享此内存映射区的进程是可见的
- MAP\_FIXED：不建议使用，这种模式下 addr 参数指定的必须的提供一个存在的 addr 参数
- fd：文件描述符。每次 map 操作会导致文件的引用计数加 1，每次 unmap 操作或者结束进程会导致引用计数减 1
- offset：文件偏移量。进行映射的文件位置，从文件起始地址向后的位移量

下面总结一下 MappedByteBuffer 的特点和不足之处：

- MappedByteBuffer 使用的是堆外的虚拟内存，因此分配 (map) 的内存大小不受 JVM 的 -Xmx 参数限制，但是也是有大小限制的。
- 如果当文件超出 Integer.MAX\_VALUE 字节限制时，可以通过 position 参数重新 map 文件后面的内容。
- MappedByteBuffer 在处理大文件时性能的确很高，但也存内存占用、文件关闭不确定等问题，被其打开的文件只有在垃圾回收的才会被关闭，而且这个时间点是不确定的。
- MappedByteBuffer 提供了文件映射内存的 mmap() 方法，也提供了释放映射内存的 unmap() 方法。然而 unmap() 是 FileChannelImpl 中的私有方法，无法直接显示调用。因此，用户程序需要通过 Java 反射的调用 sun.misc.Cleaner 类的 clean() 方法手动释放映射占用的内存区域。

```

public static void clean(final Object buffer) throws Exception {
    AccessController.doPrivileged((PrivilegedAction<Void>) () -> {
        try {
            Method getCleanerMethod = buffer.getClass().getMethod("cleaner", new
Class[0]);
            getCleanerMethod.setAccessible(true);
            Cleaner cleaner = (Cleaner) getCleanerMethod.invoke(buffer, new
Object[0]);
            cleaner.clean();
        } catch(Exception e) {
            e.printStackTrace();
        }
    });
}

```

## DirectByteBuffer

DirectByteBuffer 的对象引用位于 Java 内存模型的堆里面，JVM 可以对 DirectByteBuffer 的对象进行内存分配和回收管理，一般使用 DirectByteBuffer 的静态方法 allocateDirect() 创建 DirectByteBuffer 实例并分配内存。

```

public static ByteBuffer allocateDirect(int capacity) {
    return new DirectByteBuffer(capacity);
}

```

DirectByteBuffer 内部的字节缓冲区位于堆外的（用户态）直接内存，它是通过 Unsafe 的本地方法 allocateMemory() 进行内存分配，底层调用的是操作系统的 malloc() 函数。

```

DirectByteBuffer(int cap) {
    super(-1, 0, cap, cap);
    boolean pa = VM.isDirectMemoryPageAligned();
    int ps = Bits.pageSize();
    long size = Math.max(1L, (long)cap + (pa ? ps : 0));
    Bits.reserveMemory(size, cap);

    long base = 0;
    try {
        base = unsafe.allocateMemory(size);
    } catch (OutOfMemoryError x) {
        Bits.unreserveMemory(size, cap);
        throw x;
    }
    unsafe.setMemory(base, size, (byte) 0);
    if (pa && (base % ps != 0)) {
        address = base + ps - (base & (ps - 1));
    } else {
        address = base;
    }
    cleaner = Cleaner.create(this, new Deallocator(base, size, cap));
    att = null;
}

```

除此之外，初始化 DirectByteBuffer 时还会创建一个 Deallocator 线程，并通过 Cleaner 的 freeMemory() 方法来对直接内存进行回收操作，freeMemory() 底层调用的是操作系统的 free() 函数。

```

private static class Deallocator implements Runnable {
    private static Unsafe unsafe = Unsafe.getUnsafe();

    private long address;
    private long size;
    private int capacity;

    private Deallocator(long address, long size, int capacity) {
        assert (address != 0);
        this.address = address;
        this.size = size;
        this.capacity = capacity;
    }

    public void run() {
        if (address == 0) {
            return;
        }
        unsafe.freeMemory(address);
        address = 0;
        Bits.unreserveMemory(size, capacity);
    }
}

```

由于使用 DirectByteBuffer 分配的是系统本地的内存，不在 JVM 的管控范围之内，因此直接内存的回收和堆内存的回收不同，直接内存如果使用不当，很容易造成 OutOfMemoryError。

说了这么多，那么 DirectByteBuffer 和零拷贝有什么关系？前面有提到在 MappedByteBuffer 进行内存映射时，它的 map() 方法会通过 Util.newMappedByteBuffer() 来创建一个缓冲区实例，初始化的代码如下：

```

static MappedByteBuffer newMappedByteBuffer(int size, long addr, FileDescriptor
fd,
                                            Runnable unmapper) {
    MappedByteBuffer dbb;
    if (directByteBufferConstructor == null)
        initDBBConstructor();
    try {
        dbb = (MappedByteBuffer)directByteBufferConstructor.newInstance(
            new Object[] { new Integer(size), new Long(addr), fd, unmapper });
    } catch (InstantiationException | IllegalAccessException |
InvocationTargetException e) {
        throw new InternalError(e);
    }
    return dbb;
}

private static void initDBBConstructor() {
    AccessController.doPrivileged(new PrivilegedAction<Void>() {
        public void run() {
            try {
                Class<?> cl = Class.forName("java.nio.DirectByteBufferR");
                Constructor<?> ctor = cl.getDeclaredConstructor(
                    new Class<?>[] { int.class, Long.class,
FileDescriptor.class,
                                            Runnable.class });
                ctor.setAccessible(true);
            }
        }
    });
}

```

```

        directByteBufferRConstructor = ctor;
    } catch (ClassNotFoundException | NoSuchMethodException |
              IllegalArgumentException | ClassCastException x) {
        throw new InternalError(x);
    }
    return null;
});
}

```

DirectByteBuffer 是 MappedByteBuffer 的具体实现类。实际上，Util.newMappedByteBuffer() 方法通过反射机制获取 DirectByteBuffer 的构造器，然后创建一个 DirectByteBuffer 的实例，对应的是一个单独用于内存映射的构造方法：

```

protected DirectByteBuffer(int cap, long addr, FileDescriptor fd, Runnable
unmapper) {
    super(-1, 0, cap, cap, fd);
    address = addr;
    cleaner = Cleaner.create(this, unmapper);
    att = null;
}

```

因此，除了允许分配操作系统的直接内存以外，DirectByteBuffer 本身也具有文件内存映射的功能，这里不做过多说明。我们需要关注的是，DirectByteBuffer 在 MappedByteBuffer 的基础上提供了内存映像文件的随机读取 get() 和写入 write() 的操作。

- 内存映像文件的随机读操作

```

public byte get() {
    return ((unsafe.getByte(ix(nextGetIndex())));
}

public byte get(int i) {
    return ((unsafe.getByte(ix(checkIndex(i)))));
}

```

- 内存映像文件的随机写操作

```

public ByteBuffer put(byte x) {
    unsafe.putByte(ix(nextPutIndex()), ((x)));
    return this;
}

public ByteBuffer put(int i, byte x) {
    unsafe.putByte(ix(checkIndex(i)), ((x)));
    return this;
}

```

内存映像文件的随机读写都是借助 ix() 方法实现定位的， ix() 方法通过内存映射空间的内存首地址 (address) 和给定偏移量 i 计算出指针地址，然后由 unsafe 类的 get() 和 put() 方法和对指针指向的数据进行读取或写入。

```

private long ix(int i) {
    return address + ((long)i << 0);
}

```

## FileChannel

FileChannel 是一个用于文件读写、映射和操作的通道，同时它在并发环境下是线程安全的，基于 FileInputStream、 FileOutputStream 或者 RandomAccessFile 的 getChannel() 方法可以创建并打开一个文件通道。FileChannel 定义了 transferFrom() 和 transferTo() 两个抽象方法，它通过在通道和通道之间建立连接实现数据传输的。

- transferTo(): 通过 FileChannel 把文件里面的源数据写入一个 WritableByteChannel 的目的通道。

```
public abstract long transferTo(long position, long count, WritableByteChannel target)
    throws IOException;
```

- transferFrom(): 把一个源通道 ReadableByteChannel 中的数据读取到当前 FileChannel 的文件里面。

```
public abstract long transferFrom(ReadableByteChannel src, long position, long count)
    throws IOException;
```

下面给出 FileChannel 利用 transferTo() 和 transferFrom() 方法进行数据传输的使用示例：

```
private static final String CONTENT = "Zero copy implemented by FileChannel";
private static final String SOURCE_FILE = "/source.txt";
private static final String TARGET_FILE = "/target.txt";
private static final String CHARSET = "UTF-8";
```

首先在类加载根路径下创建 source.txt 和 target.txt 两个文件，对源文件 source.txt 文件写入初始化数据。

```
@Before
public void setup() {
    Path source = Paths.get(getClassPath(SOURCE_FILE));
    byte[] bytes = CONTENT.getBytes(Charset.forName(CHARSET));
    try (FileChannel fromChannel = FileChannel.open(source,
StandardOpenOption.READ,
        StandardOpenOption.WRITE, StandardOpenOption.TRUNCATE_EXISTING)) {
        fromChannel.write(ByteBuffer.wrap(bytes));
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

对于 transferTo() 方法而言，目的通道 toChannel 可以是任意的单向字节写通道 WritableByteChannel；而对于 transferFrom() 方法而言，源通道 fromChannel 可以是任意的单向字节读通道 ReadableByteChannel。其中，FileChannel、SocketChannel 和 DatagramChannel 等通道实现了 WritableByteChannel 和 ReadableByteChannel 接口，都是同时支持读写的双向通道。为了方便测试，下面给出基于 FileChannel 完成 channel-to-channel 的数据传输示例。

- 通过 transferTo() 将 fromChannel 中的数据拷贝到 toChannel

```

    @Test
    public void transferTo() throws Exception {
        try (FileChannel fromChannel = new RandomAccessFile(
                getClassPath(SOURCE_FILE), "rw").getChannel();
            FileChannel toChannel = new RandomAccessFile(
                getClassPath(TARGET_FILE), "rw").getChannel()) {
            long position = 0L;
            long offset = fromChannel.size();
            fromChannel.transferTo(position, offset, toChannel);
        }
    }
}

```

- 通过 transferFrom() 将 fromChannel 中的数据拷贝到 toChannel

```

    @Test
    public void transferFrom() throws Exception {
        try (FileChannel fromChannel = new RandomAccessFile(
                getClassPath(SOURCE_FILE), "rw").getChannel();
            FileChannel toChannel = new RandomAccessFile(
                getClassPath(TARGET_FILE), "rw").getChannel()) {
            long position = 0L;
            long offset = fromChannel.size();
            toChannel.transferFrom(fromChannel, position, offset);
        }
    }
}

```

下面介绍 transferTo() 和 transferFrom() 方法的底层实现原理，这两个方法也是 java.nio.channels.FileChannel 的抽象方法，由子类 sun.nio.ch.FileChannelImpl.java 实现。transferTo() 和 transferFrom() 底层都是基于 sendfile 实现数据传输的，其中 FileChannelImpl.java 定义了 3 个常量，用于标示当前操作系统的内核是否支持 sendfile 以及 sendfile 的相关特性。

```

private static volatile boolean transferSupported = true;
private static volatile boolean pipeSupported = true;
private static volatile boolean fileSupported = true;

```

- transferSupported：用于标记当前的系统内核是否支持 sendfile() 调用，默认为 true。
- pipeSupported：用于标记当前的系统内核是否支持文件描述符 (fd) 基于管道 (pipe) 的 sendfile() 调用，默认为 true。
- fileSupported：用于标记当前的系统内核是否支持文件描述符 (fd) 基于文件 (file) 的 sendfile() 调用，默认为 true。

下面以 transferTo() 的源码实现为例。FileChannelImpl 首先执行 transferToDirectly() 方法，以 sendfile 的零拷贝方式尝试数据拷贝。如果系统内核不支持 sendfile，进一步执行 transferToTrustedChannel() 方法，以 mmap 的零拷贝方式进行内存映射，这种情况下目的通道必须是 FileChannelImpl 或者 SelChImpl 类型。如果以上两步都失败了，则执行 transferToArbitraryChannel() 方法，基于传统的 I/O 方式完成读写，具体步骤是初始化一个临时的 DirectBuffer，将源通道 FileChannel 的数据读取到 DirectBuffer，再写入目的通道 WritableByteChannel 里面。

```

public long transferTo(long position, long count, WritableByteChannel target)
    throws IOException {
    // 计算文件的大小
    long sz = size();
    // 校验起始位置
}

```

```

    if (position > sz)
        return 0;
    int icount = (int) Math.min(count, Integer.MAX_VALUE);
    // 校验偏移量
    if ((sz - position) < icount)
        icount = (int)(sz - position);

    long n;

    if ((n = transferToDirectly(position, icount, target)) >= 0)
        return n;

    if ((n = transferToTrustedChannel(position, icount, target)) >= 0)
        return n;

    return transferToArbitraryChannel(position, icount, target);
}

```

接下来重点分析一下 transferToDirectly() 方法的实现，也就是 transferTo() 通过 sendfile 实现零拷贝的精髓所在。可以看到，transferToDirectlyInternal() 方法先获取到目的通道 WritableByteChannel 的文件描述符 targetFD，获取同步锁然后执行 transferToDirectlyInternal() 方法。

```

private long transferToDirectly(long position, int icount, WritableByteChannel
target)
    throws IOException {
// 省略从target获取targetFD的过程
if (!nd.transferToDirectlyNeedsPositionLock()) {
    synchronized (positionLock) {
        long pos = position();
        try {
            return transferToDirectlyInternal(position, icount,
                target, targetFD);
        } finally {
            position(pos);
        }
    }
} else {
    return transferToDirectlyInternal(position, icount, target, targetFD);
}
}

```

最终由 transferToDirectlyInternal() 调用本地方法 transferTo0()，尝试以 sendfile 的方式进行数据传输。如果系统内核完全不支持 sendfile，比如 Windows 操作系统，则返回 UNSUPPORTED 并把 transferSupported 标识为 false。如果系统内核不支持 sendfile 的一些特性，比如说低版本的 Linux 内核不支持 DMA gather copy 操作，则返回 UNSUPPORTED\_CASE 并把 pipeSupported 或者 fileSupported 标识为 false。

```

private long transferToDirectlyInternal(long position, int icount,
                                         WritableByteChannel target,
                                         FileDescriptor targetFD) throws
IOException {
    assert !nd.transferToDirectlyNeedsPositionLock() ||
        Thread.holdsLock(positionLock);

    long n = -1;
    int ti = -1;
}

```

```

try {
    begin();
    ti = threads.add();
    if (!isOpen())
        return -1;
    do {
        n = transferTo0(fd, position, iCount, targetFD);
    } while ((n == IOStatus.INTERRUPTED) && isOpen());
    if (n == IOStatus.UNSUPPORTED_CASE) {
        if (target instanceof SinkChannelImpl)
            pipeSupported = false;
        if (target instanceof FileChannelImpl)
            fileSupported = false;
        return IOStatus.UNSUPPORTED_CASE;
    }
    if (n == IOStatus.UNSUPPORTED) {
        transferSupported = false;
        return IOStatus.UNSUPPORTED;
    }
    return IOStatus.normalize(n);
} finally {
    threads.remove(ti);
    end (n > -1);
}
}

```

本地方法 (native method) transferTo0() 通过 JNI (Java Native Interface) 调用底层 C 的函数，这个 native 函数 (Java\_sun\_nio\_ch\_FileChannelImpl\_transferTo0) 同样位于 JDK 源码包下的 native/sun/nio/ch/FileChannelImpl.c 源文件里面。JNI 函数 Java\_sun\_nio\_ch\_FileChannelImpl\_transferTo0() 基于条件编译对不同的系统进行预编译，下面是 JDK 基于 Linux 系统内核对 transferTo() 提供的调用封装。

```

#ifndef __linux__ || defined(__solaris__)
#include <sys/sendfile.h>
#elif defined(_AIX)
#include <sys/socket.h>
#elif defined(_ALLBSD_SOURCE)
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.h>

#define lseek64 lseek
#define mmap64 mmap
#endif

JNIEXPORT jlong JNICALL
Java_sun_nio_ch_FileChannelImpl_transferTo0(JNIEnv *env, jobject this,
                                              jobject srcFDO,
                                              jlong position, jlong count,
                                              jobject dstFDO)
{
    jint srcFD = fdval(env, srcFDO);
    jint dstFD = fdval(env, dstFDO);

#if defined(__linux__)
    off64_t offset = (off64_t)position;
    jlong n = sendfile64(dstFD, srcFD, &offset, (size_t)count);

```

```
    return n;
#elif defined(__solaris__)
    result = sendfile64(dstFD, &sfv, 1, &numBytes);
    return result;
#elif defined(__APPLE__)
    result = sendfile(srcFD, dstFD, position, &numBytes, NULL, 0);
    return result;
#endif
}
```

对 Linux、Solaris 以及 Apple 系统而言，transferTo0() 函数底层会执行 sendfile64 这个系统调用完成零拷贝操作，sendfile64() 函数的原型如下：

```
#include <sys/sendfile.h>

ssize_t sendfile64(int out_fd, int in_fd, off_t *offset, size_t count);
```

下面简单介绍一下 sendfile64() 函数各个参数的含义：

- out\_fd：待写入的文件描述符
- in\_fd：待读取的文件描述符
- offset：指定 in\_fd 对应文件流的读取位置，如果为空，则默认从起始位置开始
- count：指定在文件描述符 in\_fd 和 out\_fd 之间传输的字节数

在 Linux 2.6.3 之前，out\_fd 必须是一个 socket，而从 Linux 2.6.3 以后，out\_fd 可以是任何文件。也就是说，sendfile64() 函数不仅可以进行网络文件传输，还可以对本地文件实现零拷贝操作。

## 其它的零拷贝实现

### Netty零拷贝

Netty 中的零拷贝和上面提到的操作系统层面上的零拷贝不太一样，我们所说的 Netty 零拷贝完全是基于（Java 层面）用户态的，它的更多的是偏向于数据操作优化这样的概念，具体表现在以下几个方面：

- Netty 通过 DefaultFileRegion 类对 java.nio.channels.FileChannel 的 transferTo() 方法进行包装，在文件传输时可以将文件缓冲区的数据直接发送到目的通道（Channel）
- ByteBuf 可以通过 wrap 操作把字节数组、ByteBuf、ByteBuffer 包装成一个 ByteBuf 对象，进而避免了拷贝操作
- ByteBuf 支持 slice 操作，因此可以将 ByteBuf 分解为多个共享同一个存储区域的 ByteBuf，避免了内存的拷贝
- Netty 提供了 CompositeByteBuf 类，它可以将多个 ByteBuf 合并为一个逻辑上的 ByteBuf，避免了各个 ByteBuf 之间的拷贝

其中第 1 条属于操作系统层面的零拷贝操作，后面 3 条只能算用户层面的数据操作优化

### RocketMQ和Kafka对比

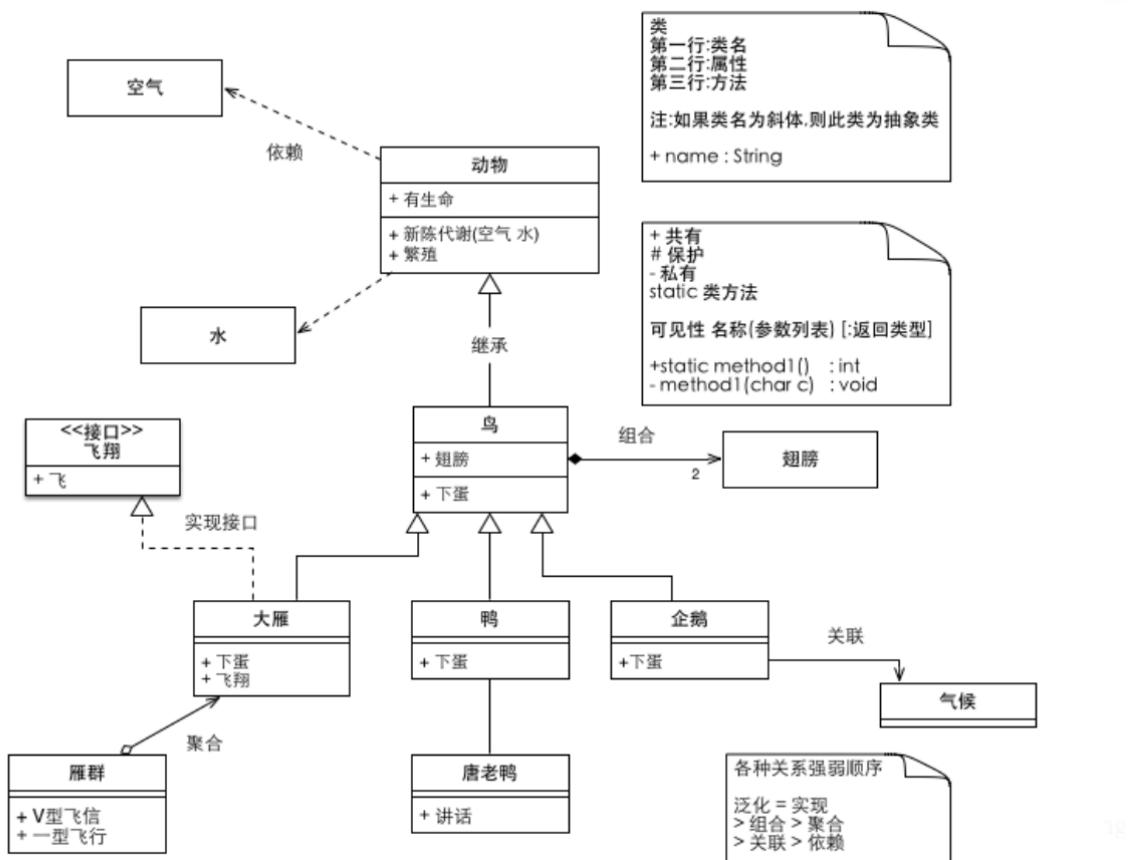
RocketMQ 选择了 mmap + write 这种零拷贝方式，适用于业务级消息这种小块文件的数据持久化和传输；而 Kafka 采用的是 sendfile 这种零拷贝方式，适用于系统日志消息这种高吞吐量的大块文件的数据持久化和传输。但是值得注意的一点是，Kafka 的索引文件使用的是 mmap + write 方式，数据文件使用的是 sendfile 方式。

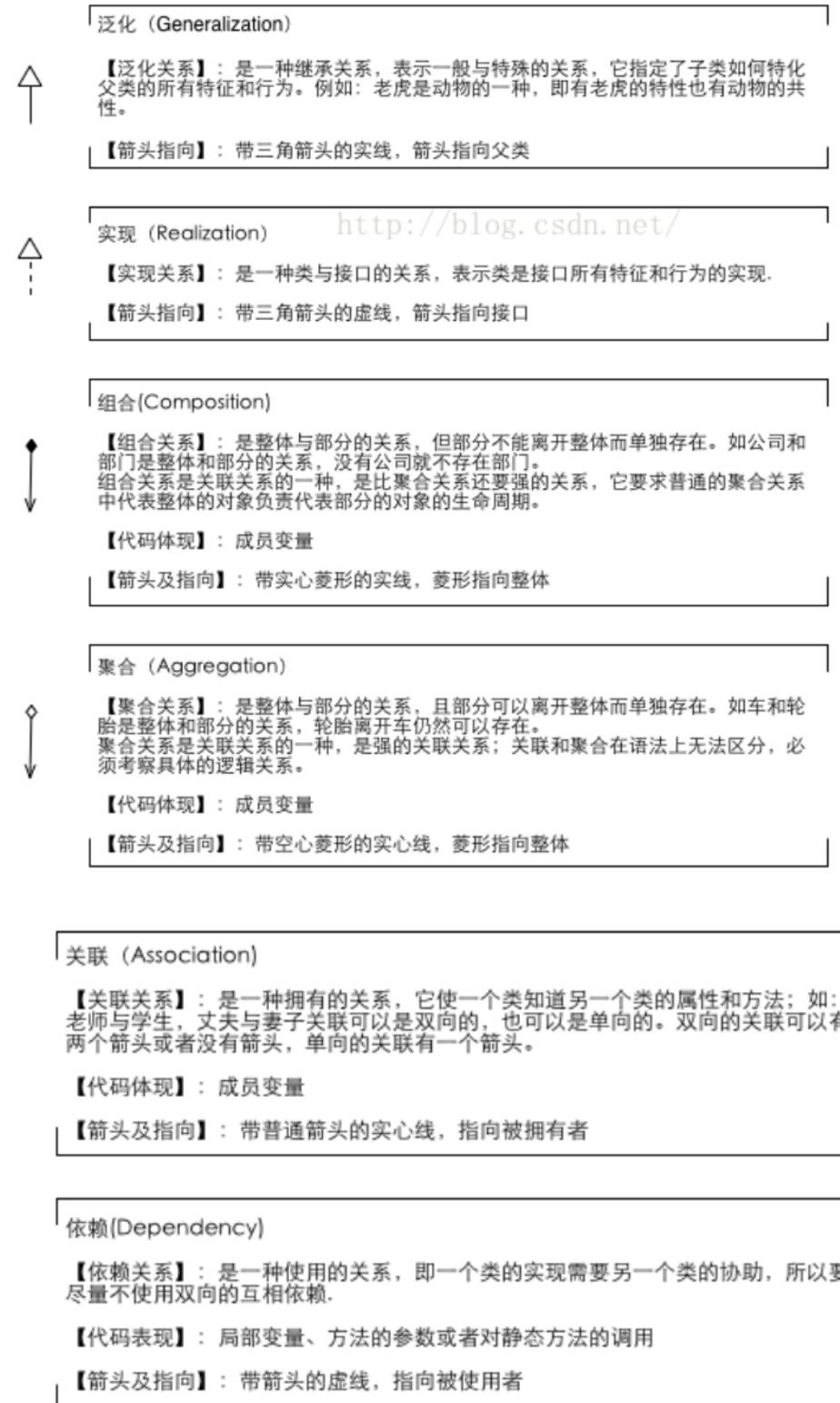
消息队列	零拷贝方式	优点	缺点
RocketMQ	mmap + write	适用于小块文件传输，频繁调用时，效率很高	不能很好的利用 DMA 方式，会比 sendfile 多消耗 CPU，内存安全性控制复杂，需要避免 JVM Crash 问题
Kafka	sendfile	可以利用 DMA 方式，消耗 CPU 较少，大块文件传输效率高，无内存安全性问题	小块文件效率低于 mmap 方式，只能是 BIO 方式传输，不能使用 NIO

## 总结

本文开篇详述了 Linux 操作系统中的物理内存和虚拟内存，内核空间和用户空间的概念以及 Linux 内部的层级结构。在此基础上，进一步分析和对比传统 I/O 方式和零拷贝方式的区别，然后介绍了 Linux 内核提供的几种零拷贝实现，包括内存映射 mmap、sendfile、sendfile + DMA gather copy 以及 splice 几种机制，并从系统调用和拷贝次数层面对它们进行了对比。接下来从源码着手分析了 Java NIO 对零拷贝的实现，主要包括基于内存映射（mmap）方式的 MappedByteBuffer 以及基于 sendfile 方式的 FileChannel。最后在篇末简单的阐述了一下 Netty 中的零拷贝机制，以及 RocketMQ 和 Kafka 两种消息队列在零拷贝实现方式上的区别。

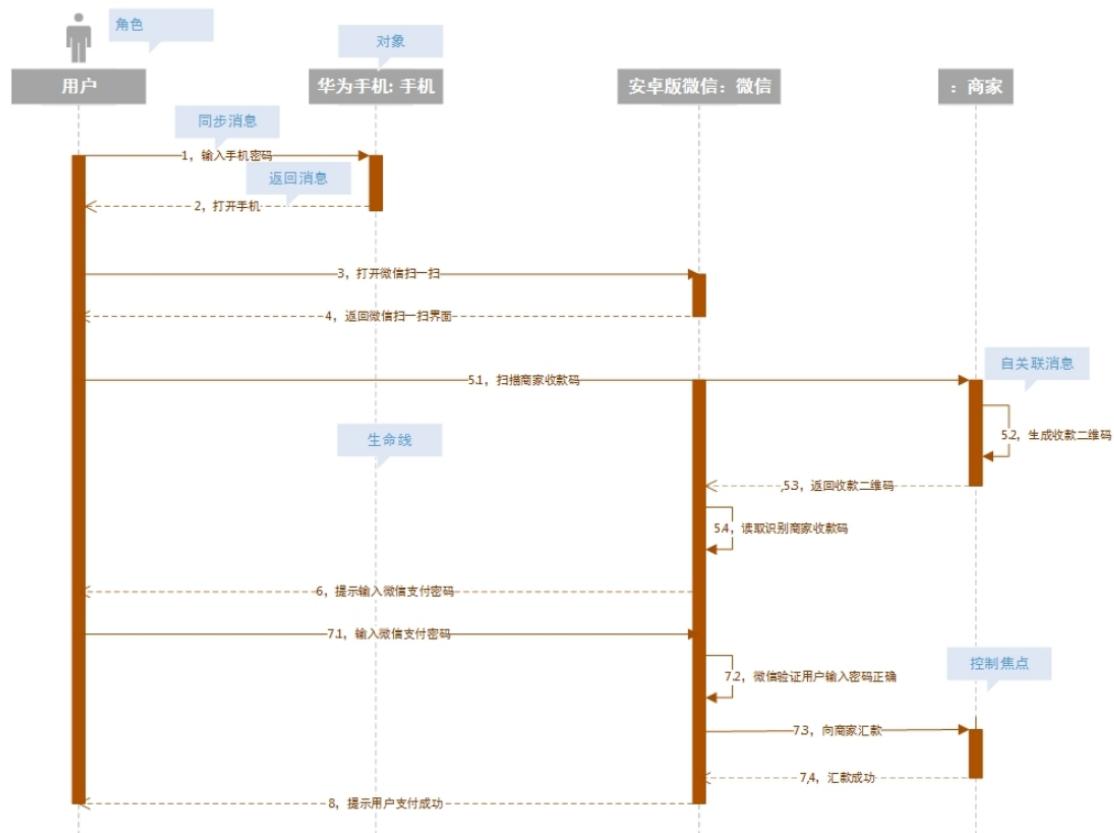
## 类图



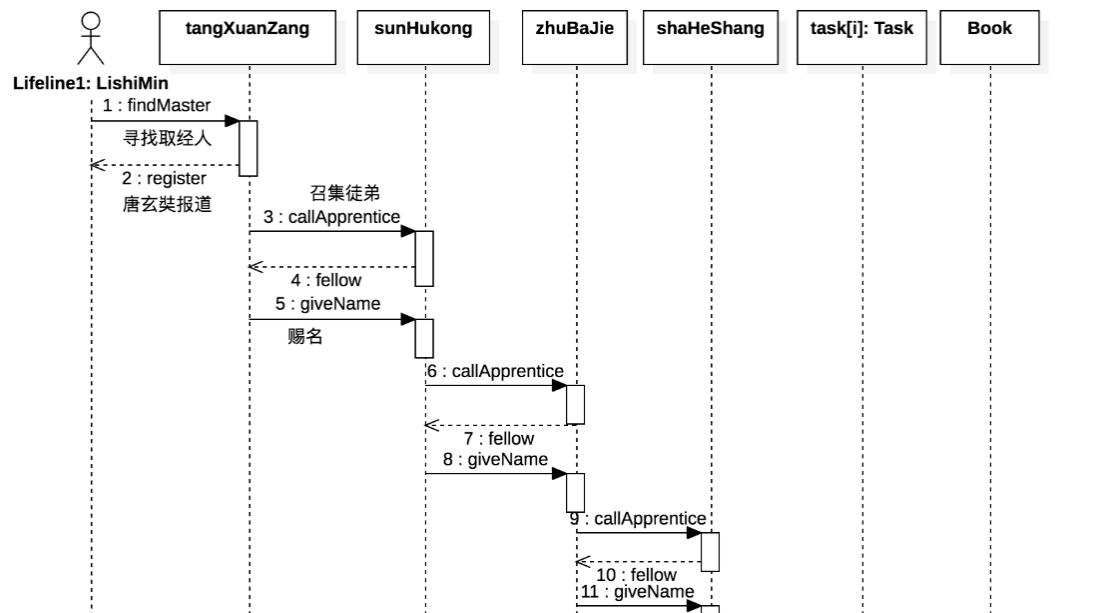


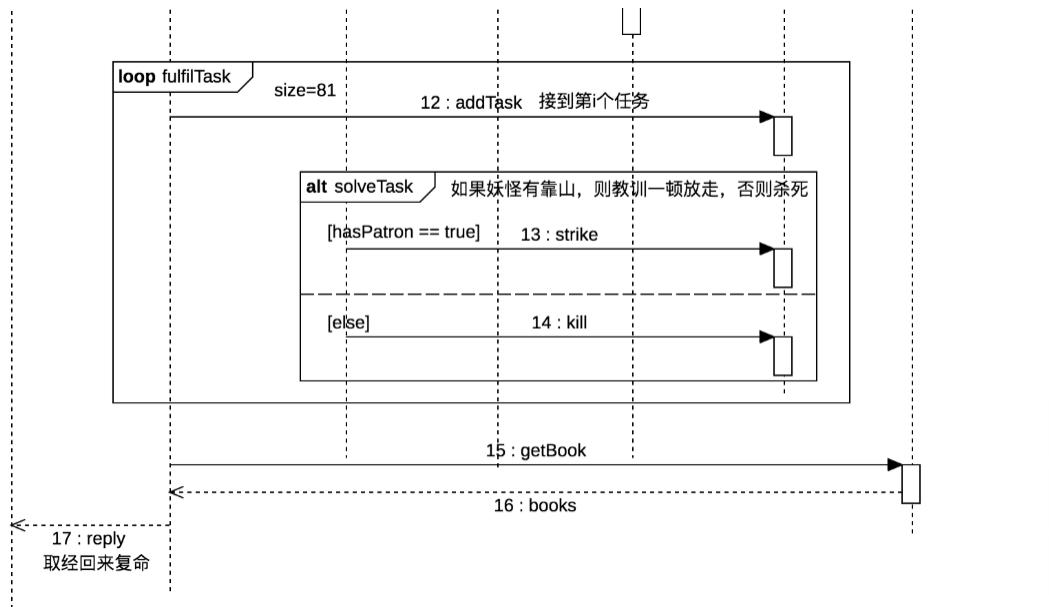
# 时序图

竖直虚线对应空闲状态，竖直方框对对应对象激活状态；时间从上往下执行



以唐僧师徒4人西天取经的超简化版为例，画出取经过程的时序图。





alternative fragment (denoted "alt") 与 if...else 对应

option fragment (denoted "opt") 与 switch 对应

parallel fragment (denoted "par") 表示并发

loop fragment (denoted "loop") 与 for、while 或者 foreach 对应

break fragment ("break") 与 for ... break 或 while...break 对应

critical fragment ("critical") 并发执行时访问临界资源

assert fragment ("assert") 断言，执行动作之前先进行判断，符合判定条件才继续执行

strict fragment ("strict") 强有序，几个动作的执行严格遵循一定的顺序（不在同一条生命线也要遵循有序性）

seq fragment ("seq") 弱有序，不在同一条生命线上的操作的顺序任意，在同一条生命线上的操作严格有序

ignore fragment ("ignore{item1, item2, ...}") 忽略指定操作

consider fragment ("consider{item1, item2, ...}") 除指定操作外，忽略其他操作

neg fragment ("neg") 当系统失败（超时或宕机）时的操作为 negative

ps: 其实 6, 9 都应该是唐僧召集徒弟

# IDEA

# Linux

## 常用命令

ps -ef | grep logstash 查看进程信息

netstat -tunlp | grep nginx 查看进程端口占用

fuser -v -n tcp 9230 找到端口文件等的占用者

wc -l file 查看文件行数(没有-l就显示文件行数，单词数，字节数，文件名)

free -g 看内存

df -h 看磁盘空间

top 命令展示进程使用信息。直接输入 d，在输入数字，表示刷新间隔时间；直接输入 u，输入用户名，表示只展示当前用户名下的进程；直接空格或者回车键，立刻刷新一次；直接键入大写 P，按照 CPU 使用排序；top -p [pid] 直接监视某进程；q 表示退出

## 数据存储方式

---

所谓的大端模式（Big-endian），是指数据的高字节，保存在内存的低地址中，而数据的低字节，保存在内存的高地址中，这样的存储模式有点儿类似于把数据当作字符串顺序处理：地址由小向大增加，而数据从高位往低位放；

所谓小端模式（Little-endian），是指数据的高字节保存在内存的高地址中，而数据的低字节保存在内存的低地址中，这种存储模式将地址的高低和数据位权有效结合起来，高地址部分权值高，低地址部分权值低，和我们的逻辑方法一致；

例如，16bit宽的数0x1234在Little-endian模式CPU内存中的存放方式（假设从地址0x4000开始存放）为：

内存地址	0x4000	0x4001
存放内容	0x34	0x12

而在Big-endian模式CPU内存中的存放方式则为：

内存地址	0x4000	0x4001
存放内容	0x12	0x34

## 计算机网络基础

---

ip

---

ipv4

ipv6

TCP

UDP

HTTP

HTTPS

## 设计模式

---

简述

---

创建型：用于创建对象，为设计类实例化新对象提供指南

结构型：用于处理类或对象的组合，对类如何设计以形成更大的结构提供指南

行为型：用于描述类或对象的交互以及职责的分配，对类之间交互以及分配职责的方式提供指南

创建型模式：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式

结构型模式：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式

行为型模式：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式

设计原则名称	定义	使用频率
单一职责原则 (Single Responsibility Principle, SRP)	一个类只负责一个功能领域中的相应职责	★★★ ★☆
开闭原则 (Open-Closed Principle, OCP)	软件实体应对扩展开放，而对修改关闭	★★★ ★★
里氏代换原则 (Liskov Substitution Principle, LSP)	所有引用基类对象的地方能够透明地使用其子类的对象	★★★ ★★
依赖倒转原则 (Dependence Inversion Principle, DIP)	抽象不应该依赖于细节，细节应该依赖于抽象	★★★ ★★
接口隔离原则 (Interface Segregation Principle, ISP)	使用多个专门的接口，而不使用单一的总接口	★★★ ☆☆
合成复用原则 (Composite Reuse Principle, CRP)	尽量使用对象组合，而不是继承来达到复用的目的	★★★ ★☆
迪米特法则 (Law of Demeter, LoD)	一个软件实体应当尽可能少地与其他实体发生相互作用	★★★ ☆☆

## 建造者模式

## 命令模式

## 单例模式

## 装饰者模式

## 观察者模式

## 代理模式

## 策略模式

## 工厂方法模式

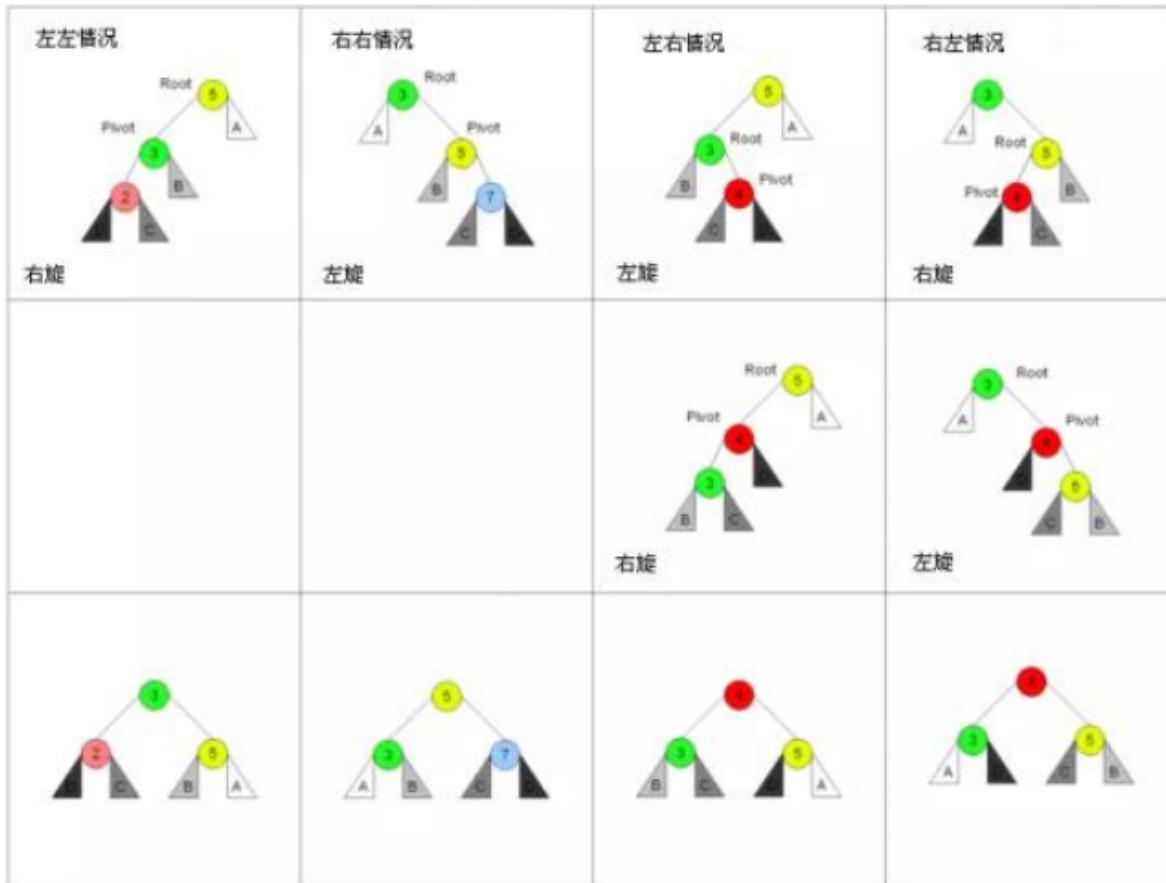
## 简单工厂模式

## 抽象工厂模式

# 数据结构

## 二叉树

### 平衡二叉树



四种不平衡的情况：

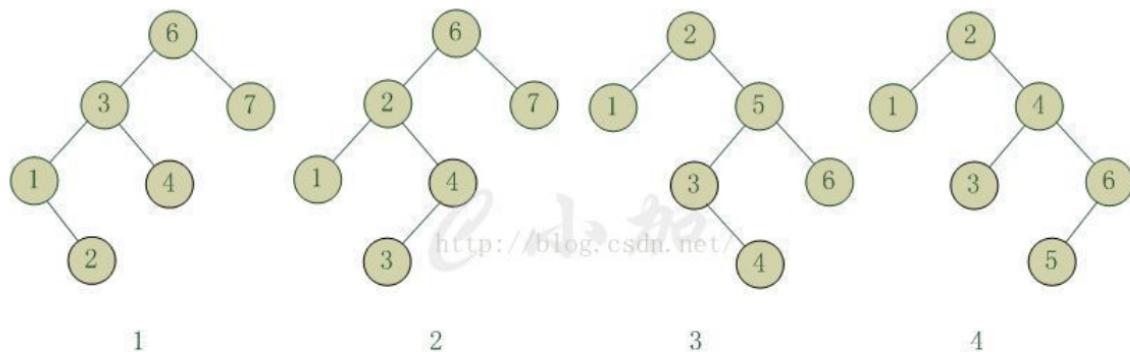


图2 四种不平衡的情况

ps 其实就是对于代码里的情况。拿2说，当前不平衡节点6的左孩子节点2，如果2的左子树比右子树低，就要先左旋后右旋。（左旋以2为圆心，右旋以6圆心）

## 红黑树

## B树

# B+树

## 常见算法

### 基本算法总结

排序方法	平均复杂度	最坏复杂度	最好复杂度	辅助空间	稳定性
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
直接插入排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	稳定
快速排序	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(1)$	不稳定
希尔排序	$O(n \log n) \sim O(n^2)$	$O(n^{1.3})$	$O(n^2)$	$O(\log n) \sim O(n)$	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	不稳定

顺序查找	$O(n)$	无序或有序队列	按顺序比较每个元素，直到找到关键字为止
二分查找（折半查找）	$O(\log n)$	有序数组	查找过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜索过程结束；如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。 如果在某一步骤数组为空，则代表找不到。
二叉排序树查找	$O(\log n)$	二叉排序树	在二叉查找树b中查找x的过程为： 1. 若b是空树，则搜索失败 2. 若x等于b的根节点的数据域之值，则查找成功； 3. 若x小于b的根节点的数据域之值，则搜索左子树 4. 查找右子树。
哈希表法（散列表）	$O(1)$	先创建哈希表（散列表）	根据键值方式(Key value)进行查找，通过散列函数，定位数据元素。
分块查找	$O(\log n)$	无序或有序队列	将n个数据元素“按块有序”划分为m块 ( $m \leq n$ )。 每一块中的结点不必有序，但块与块之间必须“按块有序”；即第1块中任一元素的关键字都必须小于第2块中任一元素的关键字；而第2块中任一元素又都必须小于第3块中的任一元素，……。然后使用二分查找及顺序查找。

## Top K问题

### 快速排序

### 归并排序

### 层序遍历

# 滑动窗口

---

## 动态规划

---

## Bit-Map

---

## 字典树

---

# 分布式及WEB相关

---

## 幂等性

---

### 概念

HTTP/1.1中对幂等性的定义是：一次和多次请求某一个资源对于资源本身应该具有同样的结果（网络超时等问题除外）。也就是说，**其任意多次执行对资源本身所产生的影响均与一次执行的影响相同**。其实幂等性是系统服务对外一种承诺（而不是实现），承诺只要调用接口成功，外部多次调用对系统的影响是一致的。声明为幂等的服务会认为外部调用失败是常态，并且失败之后必然会有重试。

### 为何需要幂等性

原因很简单，在系统调用没有达到期望的结果后，会重试。那重试就会面临问题，重试之后不能给业务逻辑带来影响，例如创建订单，第一次调用超时了，但是调用的系统不知道超时了是成功了还是失败了，然后他就重试，但是实际上第一次调用订单创建是成功的，这时候重试了，显然不能再创建订单了。

### 什么情况使用幂等性

以SQL为例，有下面三种场景，只有第三种场景需要开发人员使用其他策略保证幂等性：

1. `SELECT col1 FROM tab1 WHERE col2=2`，无论执行多少次都不会改变状态，是天然的幂等。
2. `UPDATE tab1 SET col1=1 WHERE col2=2`，无论执行成功多少次状态都是一致的，因此也是幂等操作。
3. `UPDATE tab1 SET col1=col1+1 WHERE col2=2`，每次执行的结果都会发生变化，这种不是幂等的。

### 如何保证幂等性

幂等需要通过**唯一的业务单号**来保证。也就是说相同的业务单号，认为是同一笔业务。使用这个唯一的业务单号来确保，后面多次的相同的业务单号的处理逻辑和执行效果是一致的。下面以支付为例，在不考虑并发的情况下，实现幂等很简单：①先查询一下订单是否已经支付过，②如果已经支付过，则返回支付成功；如果没有支付，进行支付流程，修改订单状态为‘已支付’。

### 防重复提交

上述的保证幂等方案是分成两步的，第②步依赖第①步的查询结果，无法保证原子性的。在高并发下就会出现下面的情况：第二次请求在第一次请求第②步订单状态还没有修改为‘已支付状态’的情况下到来。既然得出了这个结论，余下的问题也就变得简单：把查询和变更状态操作加锁，将并行操作改为串行操作。

### 乐观锁

如果只是更新已有的数据，没有必要对业务进行加锁，设计表结构时使用乐观锁，一般通过version来做乐观锁，这样既能保证执行效率，又能保证幂等。例如：`UPDATE tab1 SET col1=1,version=version+1 WHERE version=#version#` 不过，乐观锁存在失效的情况，就是常说的ABA问题，不过如果version版本一直是自增的就不会出现ABA的情况。（从网上找了一张图片很能说明乐观锁，引用过来，出自Mybatis对乐观锁的支持）

## 防重表

使用订单号orderNo做为去重表的唯一索引，每次请求都根据订单号向去重表中插入一条数据。第一次请求查询订单支付状态，当然订单没有支付，进行支付操作，无论成功与否，执行完后更新订单状态为成功或失败，删除去重表中的数据。后续的订单因为表中唯一索引而插入失败，则返回操作失败，直到第一次的请求完成（成功或失败）。**可以看出防重表作用是加锁的功能。**

## 分布式锁

这里使用的防重表可以使用分布式锁代替，比如Redis。订单发起支付请求，支付系统会去Redis缓存中查询是否存在该订单号的Key，如果不存在，则向Redis增加Key为订单号。查询订单支付已经支付，如果没有则进行支付，支付完成后删除该订单号的Key。通过Redis做到了分布式锁，只有这次订单订单支付请求完成，下次请求才能进来。相比去重表，将放并发做到了缓存中，较为高效。思路相同，**同一时间只能完成一次支付请求。**

## token令牌

这种方式分成两个阶段：申请token阶段和支付阶段。第一阶段，在进入到提交订单页面之前，需要订单系统根据用户信息向支付系统发起一次申请token的请求，支付系统将token保存到Redis缓存中，为第二阶段支付使用。第二阶段，订单系统拿着申请到的token发起支付请求，支付系统会检查Redis中是否存在该token，如果存在，表示第一次发起支付请求，删除缓存中token后开始支付逻辑处理；如果缓存中不存在，表示非法请求。实际上这里的token是一个信物，支付系统根据token确认，你是你妈的孩子。不足是需要系统间交互两次，流程较上述方法复杂。

## 支付缓冲区

把订单的支付请求都快速地接下来，一个快速接单的缓冲管道。后续使用异步任务处理管道中的数据，过滤掉重复的待支付订单。优点是同步转异步，高吞吐。不足是不能及时地返回支付结果，需要后续监听支付结果的异步返回。

# 分布式一致性

---

## 分类

强一致：当更新操作完成之后，任何多个后续进程或者线程的访问都会返回最新的更新过的值。这种是对用户最友好的，就是用户上一次写什么，下一次就保证能读到什么。根据CAP理论，这种实现需要牺牲可用性。

弱一致：系统并不保证续进程或者线程的访问都会返回最新的更新过的值。系统在数据写入成功之后，不承诺立即可以读到最新写入的值，也不会具体的承诺多久之后可以读到。

最终一致：弱一致性的特定形式。系统保证在没有后续更新的前提下，系统最终返回上一次更新操作的值。

## CAP理论

CAP是Consistency、Availability和Partition-tolerance的缩写。分别是指：

1.一致性 (Consistency)：每次读操作都能保证返回的是最新数据，在分布式系统中，如果能针对一个数据项的更新执行成功后，所有的用户都可以读到其最新的值，这样的系统就被认为具有严格的一致性。

2.可用性 (Availability)：任何一个没有发生故障的节点，会在合理的时间内返回一个正常的结果，也就是对于用户的每一个请求总是能够在有限的时间内返回结果；

3.分区容忍性 (Partition-tolerance)：当节点间出现网络分区（不同节点处于不同的子网络，子网络之间应该是联通的，但是子网络之间无法联通了，也就是被切分成了孤立的集群网络），照样可以提供满足一致性和可用性的服务，除非整个网络环境都发生了故障。

**CAP理论指出：CAP三者只能取其二，不可兼得。**

首先，如果我们要使网络分区不存在，就必须将系统部署在单个节点上，因为网络总是会出现故障，分区总是存在的，所以当部署在单节点上，可以同时保证CP，但是这时候，就没什么意义了，这都不是分布式了，同时单点故障可能会发生，就不会保证A可用性。

**所以我们必须明确一点：对于分布式系统而言，分区容错性是必须要满足的，因为分区的出现时必然，也是必须要解决的问题。所以，P必须要保证，那么我们就要在C和A之间做权衡。**

- 有两个或以上节点时，当网络分区发生时，集群中两个节点不能相互通信。此时如果保证数据的一致性C，那么必然会有**一个节点被标记为不可用的状态，违反了可用性A的要求，只能保证CP**。
- 反之，如果保证可用性A，即**两个节点可以继续各自处理请求，那么由于网络不通不能同步数据，必然又会导致数据的不一致**，只能保证AP。

## BASE理论

在上边，我们谈到，因为P总是存在的，放弃不了。另外，可用性、一致性也是我们一般系统必须要满足的，如何在可用性和一致性进行权衡，所以就出现了各种一致性的理论与算法

1.基本可用 (Basically Available)：基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性——但请注意，这绝不等价于系统不可用。如

- 响应时间上的损失：正常情况下，一个在线搜索引擎需要在0.5秒之内返回给用户相应的查询结果，但由于出现故障（比如系统部分机房发生断电或断网故障），查询结果的响应时间增加到了1~2秒。
- 功能上的损失：正常情况下，在一个电子商务网站上进行购物，消费者几乎能够顺利地完成每一笔订单，但是在一些节日大促购物高峰的时候，由于消费者的购物行为激增，为了保护购物系统的稳定性，部分消费者可能会被引导到一个降级页面。

2.软状态 (Soft State)：软状态也称为弱状态，和硬状态相对，是指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时。

3.最终一致性 (Eventual Consistency)：最终一致性强调的是系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

## 一致性变种

**因果一致性**：如果进程A在更新完某个数据项后通知了进程B，那么进程B之后对该数据项的访问都应该能够获取到进程A更新后的最新值，并且如果进程B要对该数据项进行更新操作的话，务必基于进程A更新后的最新值，即不能发生丢失更新情况。与此同时，与进程A无因果关系的进程C的数据访问则没有这样的限制。

**读己之所写**：进程A更新一个数据项之后，它自己总是能够访问到更新过的最新值，而不会看到旧值。也就是说，对于单个数据获取者来说，其读取到的数据，一定不会比自己上次写入的值旧。因此，读己之所写也可以看作是一种特殊的因果一致性。

**会话一致性**: 将对系统数据的访问过程框定在了一个会话当中: 系统能保证在同一个有效的会话中实现“读己之所写”的一致性, 也就是说, 执行更能操作之后, 客户端能够在同一个会话中始终读取到该数据项的最新值。

## 分布式一致性机制整理

# JVM

## 运行时数据区域

### 程序计数器

是一块较小的内存空间, 可以看作是当前线程所执行的字节码的行号指示器, 在虚拟机的概念模型中, 字节码解释器就是通过改变计数器的值来获得下一条要执行的字节码指令, 分支、循环、跳转、异常、线程恢复等基础功能都需要依赖这个计数器来完成。由于java虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现, 所以在任何一个确定的时间, 一个处理器(多核处理器的一核)都只会执行一条线程里的指令。所以为了线程切换后能恢复到正确的执行位置, 每条线程都需要一个独立的程序计数器, 各个线程之间互不影响。即线程私有。

如果线程正在执行一个java方法, 这个计数器记录的就是当前正在执行的虚拟机字节码指令的地址; 如果正在执行native方法, 则这个计数器的值为空(undefined)。此内存区域是java虚拟机规范中唯一不会发生oom的内存区域。

### Java虚拟机栈

也是线程私有, 且生命周期与线程一致。虚拟机栈描述的是java方法执行的内存模型: 每个方法在执行的同时都会创建一个栈帧用于存储局部变量表, 操作数栈, 动态链接, 方法出口等信息。每一个方法从调用直至执行完成的过程, 就对应一个栈帧在虚拟机栈帧中入栈到出栈的过程。

局部变量表存放**编译期**可知的各种基本数据类型、对象引用类型和returnAddress类型(指向了一条字节码指令的地址)。

其中64位的long和double类型都会占用2个局部变量空间(slot), 其余的数据类型只占一个。局部变量表所需要的内存空间在编译期间完成分配, 当进入一个方法时, 这个方法需要在帧中分配多大的局部变量空间(slot)是完全确定的。在方法运行期间不会改变局部变量表的大小

如果线程的请求的栈深度大于虚拟机所允许的深度, 会抛出StackOverflowError异常; 如果虚拟机栈支持动态扩展(当前大部分虚拟机都支持, 只不过虚拟机规范中也允许固定长度的虚拟机栈), 如果扩展到无法申请足够的内存就会抛出oom

### 本地方法栈

类似于虚拟机栈, 只不过执行的是native方法。

### Java堆

所有线程共享的内存区域, 虚拟机启动时创建。目的是存放对象实例, 几乎所有对象实例都在这里分配内存。(JIT编译及逃逸分析技术逐渐成熟, 栈上分配, 标量替换优化技术使得这一切不是那么的绝对了)

从内存回收角度看，目前收集器都采用分代回收算法，所以堆可以分为新生代、老年代。（虽然永久代叫做非堆，但其实也在堆内存中，jdk8用元空间取代，放到本地内存了）；从内存分配角度看，线程共享的java堆中可能划分出多个线程私有的分配缓冲区（Thread Local Allocation Buffer,TLAB）。无论什么角度，堆里存的都是一样的，都是对象实例。划分的目的只是为了更好的回收或者分配。当堆内存不够分配对象实例，并且堆也无法扩展了（Xms,Xmx控制），就会oom

## 方法区

所有线程共享的内存区域，用于存储虚拟机加载的类信息，常量，静态变量，即时编译器编译后的代码等数据。

方法区并不是永久代，只不过是HotSpot虚拟机团队把GC的收集区域扩展到了方法区，也就是用永久代来实现方法区而已，这样可以利用垃圾收集器来管理这个内存区域，省去为方法区内存管理的编码。如何实现方法区是各个虚拟机各自的细节，有的虚拟机并没有永久代。

使用永久代实现方法区并不好，因为会oom，而且不同虚拟机对某些方法的会有不同表现（`String.intern`方法）。后续将取消永久代而使用本地内存了。jdk 7就已经把字符串常量池移出永久代了，jdk 8则使用元空间实现方法区了。

虚拟机规范，对该内存区域的要求很宽松，甚至可以不实现垃圾回收。因为这个区域的垃圾收集行为很少，主要是对常量池的回收和类型的卸载（卸载条件也很苛刻）。当此处区域内存不足以分配，会oom

## 运行时常量池

是方法区的一部分。class文件除了有类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池，用于存放**编译期**生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。

虚拟机对class文件的每一部分格式都有着严格的要求，每个字节用于存储哪种数据都必须复合规范要求才会被虚拟机认可装载和执行。但是对于运行时常量池却没有要求，由各个虚拟机自己实现，不过，一般来说，虚拟机会保存class文件中描述的符号引用以及翻译出来的直接引用。

相比于class文件的另一个特征是具备动态性。java语言并不要求常量一定只有编译期才能产生，也就是并非预置入class文件常量池的内容才能进入方法区运行时常量池，运行期间也是可以的，比如`String.intern`方法。

由于属于方法区的一部分，所以受到方法区内存的限制，当常量池无法再申请内存就会oom

## 直接内存

direct memory不属于虚拟机运行时数据区的一部分，也不是java虚拟机规范中的内存区域，但是这部分内存也经常被使用到，也会发生oom异常。

jdk nio基于通道和buff的io方式，使用native函数库直接分配堆外内存，然后通过一个存储在java堆中的`DirectByteBuffer`对象作为这块内存的引用进行操作。

受到本机总内存的限制，所以也会oom

# 对象的奥秘

## 对象的创建

虚拟机遇到new指令，首先去检查这个指令的参数能否在常量池定位到一个类的符号引用，并且检查这个类是否已经被加载解析和初始化了，如果没有，则先执行这些操作。类加载完成后将对新生对象分配内存（对象所需内存大小在类加载完成后就完全确定了）。为对象分配内存，意味着把堆内存的一部分划出来。

**指针碰撞**: 假设Java堆中的内存绝对规整，所有用过的内存放在一边，空闲的内存放在另外一边，中间放着一个指针作为分界点的指示器，那所分配的内存仅仅是把指针向空闲的一边挪动一段与对象大小相同距离。

**空闲列表**: java堆上的内存不规整，已使用内存和未使用内存交错在一起，无法使用指针碰撞，只能维护一个空闲列表，记录哪些内存可用，在分配的时候就找出一块足够大的内存区域分配给对象实例，并更新表里的记录。

选择哪种方式取决于Java堆是否规整，而堆是否规整又取决于垃圾收集器是否有压缩整理功能。因此，serial, parNew等带有compact过程的收集器采用的是指针碰撞的方式，而cms收集器则是空闲列表。

对象的创建很频繁，即使是修改一下指针的位置，也是线程不安全的（可能正在给对象A分配内存，指针还没来及修改，对象B又同时使用原来的指针分配内存）。解决方案有两种：1. 使用同步，其实虚拟机确实是使用CAS加上失败重试的方法保证更新操作的原子性；2. 内存分配动作按照线程划分在不同空间中进行，即每个线程在Java堆中预先分配一小块内存（就是本地线程分配缓冲，TLAB）。哪个线程要分配内存就在哪个线程的TLAB上分配，只有TLAB用完并分配新的TLAB时才需要同步锁定。是否使用TLAB可用参数-XX:+/-UseTLAB设定。

内存分配完成后，虚拟机将分配到的内存空间都初始化为零值（不包括对象头），如果使用了TLAB，也可以提前在TLAB中进行。这一步操作保证了对象实例后的实例字段在Java代码中可以不赋初始值就使用，程序能访问到这些字段数据类型对应的初始零值。

接下来虚拟机将要对对象实例进行必要的设置了，比如是哪个类的实例，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。这些信息都被放在了对象的对象头（object header）之中。

以上完成后，在虚拟机的角度上，对象已经创建并分配完成了，但是对于Java来说，才刚刚开始，方法还没执行，一切字段都还是零值。所以一般来说，执行完new指令后就会执行方法，把对象的字段设置需要的值（就是构造方法啦）。new和反射都会调用到，但是反序列化（readObject）和克隆不会。反序列化会使用反射创建对象（使用构造器），但是反序列化中的构造方法描述却不是序列化对象的构造方法，而是父类的，所以反序列化会创建对象并执行父类的构造方法，前提是父类有构造方法并且没有实现序列化接口，否则最终构造方法是父类的父类的构造方法，也就是Object的构造方法（具体原因有待源码的探求）。ps：也就是说反序列化里反射的构造方法不是当前对象的，而是父类中没实现序列化接口的那个父类的构造方法。

## 对象的内存布局

对象的存储布局分为三个部分：对象头，实例数据，填充对齐。

**对象头**包括两部分：一部分是用于存储对象自身运行时数据的mark word; 另一部分是类型指针，即对象指向他的类元数据的指针，虚拟机通过这个指针来确定这个对象属于哪个类的实例。当然，也不是所有的虚拟机都在对象头中有类型指针，也就是获取对象的类元数据不一定非要通过对象本身（比如下面的句柄池）。对于数组对象，对象头中还会有一块用于保存数组长度的数据，因此虚拟机可以从普通对象的元数据就获取到数组的大小及对象的大小，但是从数组的元数据中却无法获取数组的长度大小。

**实例数据**是对象真正存储的有效信息，也是在程序代码中所定义的各种类型的字段内容。无论是从父类继承下来的还是在子类中定义的，都需要记录下来。这部分的存储数据会受到虚拟机分配策略及字段在Java源码中定义顺序的影响。Hotspot虚拟机默认的分配策略是longs/doubles、ints、shorts/chars、bytes/booleans、oops（ordinary object pointers）。从默认的分配策略可以看出，相同宽度的字段总是被分配到一起。在满足这个的前提下，在父类中定义的变量会出现在子类之前。如果compactFields参数为true（默认为true），那么子类中较窄的变量也可能插入到父类变量的空隙之中。填充对齐只是为了保证对象的大小为8字节的整数倍。

## 对象的访问定位

建立对象是为了使用对象，我们的Java程序需要通过栈上的reference数据来操作堆上的具体对象。reference类型在Java虚拟机规范中只规定了一个指向对象的引用，并没有定义这个引用的实现方式。目前主流的访问方式是使用句柄和直接指针。

**使用句柄**会在Java堆中划分出一块内存作为句柄池，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**各自的具体地址信息。

**直接指针** reference中存储的直接就是对象地址，对象头中有指向类型数据地址的指针

优缺点：句柄的优势在于对象被移动时（垃圾回收就经常需要移动对象），只需要改变句柄中的实例数据指针，reference本身无需改变；直接指针的优势在于速度更快，节省了一次指针定位的开销。本书的Sun Hotspot使用的是第二种方式。

## 垃圾收集器与内存回收

### 对象已死吗？

引用计数法：给对象中添加一个引用计数器，每当有一个地方引用他，计数值就加1，引用失效时就减1。实现简单，判定效率也高，但是很难解决对象的循环引用问题

可达性分析：通过一系列称为“GC Roots”的对象为起点，从这些节点开始向下搜索，搜索走过的路径成为引用链，当一个对象到GC Roots没有任何引用链时，证明此对象不可用。可以作为GC Roots的对象包括虚拟机栈（栈帧本地变量表）中引用的对象，方法区中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（native方法）中引用的对象

再谈引用：强软弱虚四类引用

回收方法区：永久代（Hotspot实现的方法区）的垃圾回收主要有两部分，一个是废弃的常量，另一个是无用的类。以常量池中字符串的回收为例，假如一个字符串“abc”已经进入常量池，但是当前没有任何一个String引用他，也没有任何其他引用，如果此时发生内存回收，这个“abc”常量就会被移出常量池。常量池中的其他类（接口），方法、字段的符号引用也与此类似。不过对于无用类的回收，就严格很多，必须同时满足以下三个条件，1.该类所有实例已经被回收2.加载该类的类加载器对象已经被回收3.该类对应的class对象没有被任何地方引用，无法在任何地方通过反射访问该类的方法。满足这些，虚拟机就可以进行回收。只是可以，不是必然。

## 垃圾收集算法

### 标记-清除

首先标记出所有要回收的对象，在标记完成后统一回收所有标记的对象。主要不足之处在于标记和清除的效率不高还有就是标记清除之后会有大量不连续的内存碎片，空间碎片太多可能导致分配较大对象时无法找到足够的连续内存而导致垃圾回收。

### 复制算法

将可用内存分为大小一致的两块，每次使用只用一块，当一块用完了，就把还存活的对象移动到另一块，然后再把使用过的内存一次清理掉。这样使得每次只对整个半区进行回收，内存分配也就不用考虑内存碎片的问题了，只要移动堆顶指针，按顺序分配即可。简单高效，但是内存有点浪费。目前新生代采用复制算法，包括一个Eden区和2个survivor区，Eden与survivor默认比例是8:1，也就是整个新生代，只有90%的区域可用，10%被浪费。当回收时，会把Eden区和survivor区存活的对象复制到另一个survivor区中，最后清理掉Eden和刚才用过的survivor区。

### 标记-整理

标记过程与标记-清除一样，但是会让所有可存活对象，都向一端移动，然后清理掉端边界以外的内存。也解决了内存碎片问题。

## 分代收集

根据对象的生命周期不同，将内存分为几块。新生代对象朝生夕灭，适合用复制算法（存活对象少，复制效率也高），老年代存活率高，没有额外内存作为分配担保，适合标记-清除或者标记-整理算法。

## HotSpot的算法实现

### 枚举根节点

可达性分析从GC Roots开始找引用链过程中，系统必须处于一个确保一致性的快照的状态中，也就是不可以让对象的引用关系还一直变化，否则可达性分析的结果就无法保证。所以GC不得不停止所有Java执行线程(native代码可以执行，但是不能和jvm交互)。也就是stop the world

### 安全点

JVM选择用什么方式会影响到GC的实现：如果JVM选择不记录任何这种类型的数据，那么它就无法区分内存里某个位置上的数据到底应该解读为引用类型还是整型还是别的什么。这种条件下，实现出来的GC就会是**“保守式GC”**。在进行GC的时候，JVM开始从一些已知位置（比如说JVM栈）开始扫描内存，扫描的时候每看到一个数字就看看它“像不像是一个指向GC堆中的指针”。这里会涉及上下边界检查（GC堆的上下界是已知的）、对齐检查（通常分配空间的时候会有对齐要求，假如说是4字节对齐，那么不能被4整除的数字就肯定不是指针），之类的。然后递归的这么扫描出去。保守式GC实现简单但是不高效；JVM可以选择在栈上不记录类型信息，而在对象上记录类型信息。这样的话，扫描栈的时候仍然会跟上面说的过程一样，但扫描到GC堆内的对象时因为对象带有足够类型信息了，JVM就能够判断出在该对象内什么位置的数据是引用类型了。这种是**“半保守式GC”**，也称为根上保守，为了支持半保守式GC，运行时需要在对象上带有足够的元数据。如果是JVM的话，这些数据可能在类加载器或者对象模型的模块里计算得到，但不需要JIT编译器的特别支持；**准确式GC**关键就是“类型”，也就是说给定某个位置上的某块数据，要能知道它的准确类型是什么，这样才可以合理地解读数据的含义；GC所关心的含义就是“这块数据是不是指针”。要实现这样的GC，JVM就要能够判断出所有位置上的数据是不是指向GC堆里的引用，包括活动记录（栈+寄存器）里的数据。

从外部记录下类型信息，存成映射表。现在三种主流的高性能JVM实现，HotSpot、JRockit和J9都是这样做的。其中，HotSpot把这样的数据结构叫做OopMap，JRockit里叫做livemap，J9里叫做GC map。Apache Harmony的DRLVM也把它叫GCMAP。

要实现这种功能，需要虚拟机里的解释器和JIT编译器都有相应的支持，由它们来生成足够的元数据提供给GC。

使用这样的映射表一般有两种方式：

- 每次都遍历原始的映射表，循环的一个个偏移量扫描过去；这种用法也叫“解释式”；
- 为每个映射表生成一块定制的扫描代码（想像扫描映射表的循环被展开的样子），以后每次要用映射表就直接执行生成的扫描代码；这种用法也叫“编译式”。

在HotSpot中，对象的类型信息里有记录自己的OopMap，记录了在该类型的对象内什么偏移量上是什么类型的数据。所以从对象开始向外的扫描可以是准确的；这些数据是在类加载过程中计算得到的。

可以把oopMap简单理解成是调试信息。在源代码里面每个变量都是有类型的，但是编译之后的代码就只有变量在栈上的位置了。oopMap就是一个附加的信息，告诉你栈上哪个位置本来是个什么东西。这个信息是在JIT编译时跟机器码一起产生的。因为只有编译器知道源代码跟产生的代码的对应关系。**每个方法可能会有好几个oopMap，就是根据safepoint把一个方法的代码分成几段，每一段都有一个oopMap，作用域自然也仅限于这一段代码。假如这段代码的循环中引用多个对象，肯定就会有多个变量，编译后占据栈上的多个位置。那这段代码的oopMap就会包含多条记录。**

每个被JIT编译过后的方法也会在一些特定的位置记录下OopMap，记录了执行到该方法的某条指令的时候，栈上和寄存器里哪些位置是引用。这样GC在扫描栈的时候就会查询这些OopMap就知道哪里是引用了。这些特定的位置主要在：

- 1、循环的末尾

- 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

这种位置被称为“安全点” (safepoint)

而仍然在字节码解释器中执行的方法则可以通过解释器里的功能自动生成出OopMap出来给GC用。

平时这些OopMap都是压缩了存在内存里的；在GC的时候才按需解压出来使用。

HotSpot是用“解释式”的方式来使用OopMap的，每次都循环变量里面的项来扫描对应的偏移量。

对Java线程中的JNI方法，它们既不是由JVM里的解释器执行的，也不是由JVM的JIT编译器生成的，所以会缺少OopMap信息。那么GC碰到这样的栈帧该如何维持准确性呢？

HotSpot的解决方法是：所有经过JNI调用边界（调用JNI方法传入的参数、从JNI方法传回的返回值）的引用都必须用“句柄” (handle) 包装起来。JNI需要调用Java API的时候也必须自己用句柄包装指针。在这种实现中，JNI方法里写的“jobject”实际上不是直接指向对象的指针，而是先指向一个句柄，通过句柄才能间接访问到对象。这样在扫描到JNI方法的时候就不需要扫描它的栈帧了——只要扫描句柄表就可以得到所有从JNI方法能访问到的GC堆里的对象。

## 安全区域

GC发生时如何让所有线程跑到最近安全点的时候停下？GC的时候会在安全点位置（或者创建对象需要分配内存的地方）设置一个标志，各个线程主动轮询这个标志，发现标志为真就自己中断挂起。**安全区域**指的是在一段代码片段中，引用关系不会发生变化，在这个区域任何地方开始GC都是安全的。典型的例子就算sleep和block。

当线程执行到安全区域时，首先标示自己已经进入了安全区域，那样如果在这段时间里jvm发起gc，就不需要管标示自己在安全区域的那些线程了，在线程离开安全区域时，会检查系统是否正在执行gc，如果是，那么就等到gc完成后再离开安全区域

## 垃圾收集器

### Serial收集器

- 此收集器是单线程的，但它的“单线程”的意义并不仅仅说明它是一个CPU或一条收集线程去完成垃圾收集工作，更重要的是在进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束（会发生STW）。
- 简单高效（与其他收集器的单线程比）没有线程交互的开销，专心做垃圾收集。
- 新生代收集器、使用复制算法。
- 大多运行在Client模式下的默认新生代收集器。

### ParNew收集器

- Serial收集器的多线程版本，添加了可用的所有控制参数（例如：-xx:SurvivorRatio、-XX:PretenureSizeThreshold、-XX:HandlePromotionFailure等），收集算法、Stop the World、对象分配规则，回收策略等都与Serial收集器完全一样。
- 新生代收集器，使用复制算法。
- 大多运行在Server模式下的默认新生代收集器。
- 目前只有它能与CMS收集器配合工作。

### parallel Scavenge收集器

- 新生代收集器，使用复制算法。
- 吞吐量优先**收集器，所谓吞吐量就是用户代码运行时间/（用户代码运行时间+垃圾回收时间），虚拟机总共运行100分钟，垃圾回收1分钟，那吞吐量就是99%。高吞吐量可以更加高效的利用cpu时间，尽快完成计算任务，适合后台运算而不需要太多交互的任务。
- 控制最大垃圾收集停顿时间的-XX:MaxGCPauseMillis参数  
MaxGCPauseMillis参数允许的值是一个大于0的毫秒数，收集器将尽力保证内存回收花费的时间

**不超过设定值。**不过大家不要异想天开地认为如果把这个参数的值设置得稍小一点就能使得系统的垃圾收集速度变得更快，GC停顿时间缩短是以**牺牲吞吐量和新生代空间**来换取的：系统把新生代调小一些，收集300MB新生代肯定比收集500MB快吧，这也直接导致垃圾收集发生得更频繁一些，原来10秒收集一次、每次停顿100毫秒，现在变成5秒收集一次、每次停顿70毫秒。停顿时间的确在下降，但吞吐量也降下来了。

- 直接设置吞吐量大小的 -XX:GCTimeRatio参数。  
GCTimeRatio参数的值应当是一个大于0小于100的整数，也就是垃圾收集时间占总时间的比率。如果把此参数设置为19，那允许的最大GC时间就占总时间的5%（即 $1 / (1+19)$ ），默认值为99，就是允许最大1%（即 $1 / (1+99)$ ）的垃圾收集时间
- -XX:+UseAdaptiveSizePolicy是一个开关参数，当这个参数打开之后，就不需要手工指定新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX:SurvivorRatio）、晋升老年代对象年龄（-XX:PretenureSizeThreshold）等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量，这种调节方式称为**GC自适应的调节策略**
- Parallel Scavenge收集器能够配合自适应调节策略，把内存管理的调优任务交给虚拟机去完成。只需要把基本的内存数据设置好（如-Xmx设置最大堆），然后使用-XX:MaxGCPauseMillis参数（更关注最大停顿时间）或GCTimeRatio参数（更关注吞吐量）给虚拟机设立一个优化目标，那具体细节参数的调节工作就由虚拟机完成了。

**自适应调节策略也是Parallel Scavenge收集器与ParNew收集器的一个重要区别**

## Serial Old收集器

- 老年代收集器、单线程收集器、使用“标记-整理”算法。

## Parallel Old收集器

- 老年代收集器，使用多线程和“标记-整理”算法，JDK1.6中才开始提供。
- 吞吐量优先收集器

## CMS收集器（Concurrent Market Sweep）

- CMS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。
- CMS老年代收集器。
- 并发收集器。
- 基于“标记-清除”算法实现，整个过程分为4个步骤：

初始化标记（CMS initial mark）

并发标记（CMS concurrent mark）

重新标记（CMS remark）

并发清除（CMS concurrent Sweep）

初始标记仅仅是标记直接和GC Roots关联的对象，速度很快

并发标记阶段是进行GC Roots Tracing的过程。

重新标记阶段则是为了修正并发标记期间用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始化标记阶段稍长一些，但远比并发标记的时间短。

由于整个过程中耗时最长的并发标记和并发清除过程收集器都可以与用户线程一起工作，所以，从总体上来说，CMS收集器的内存回收过程是用户线程一起并发执行的。

特点：并发收集、低停顿。

缺点：1.对cpu资源敏感，在并发阶段虽然不会导致用户线程暂停，但是会占用一部分线程（cpu资源）  
2.无法处理浮动垃圾，可能出现Concurrent Mode Failure 失败导致另一次Full GC的发生。并发清理阶段用户线程还在运行，还产生的垃圾叫做浮动垃圾。也是由于用户线程还在运行那就还需要预留内存给用户线程使用，所以并非等到老年代塞满再GC，如果预留空间不够用户线程使用就会出现Concurrent Mode Failure 失败导致另一次Full GC  
3.会产生大量的内存碎片。keyi 设置参数进行内存整理，但会造成停顿时间变长

## G1收集器

初始标记（Initial Marking）：仅仅只是标记一下GC Roots能直接关联到的对象，并且修改TAMS（Next Top at Mark Start）的值，让下一阶段用户程序并发运行时，能在正确可用的Region中创建对象，此阶段需要停顿线程，但耗时很短。

并发标记（Concurrent Marking）：从GC Root开始对堆中对象进行可达性分析，找到存活对象，此阶段耗时较长，但可与用户程序并发执行。

最终标记（Final Marking）：为了修正正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，虚拟机将这段时间对象变化记录在线程的Remembered Set Logs里面，最终标记阶段需要把Remembered Set Logs的数据合并到Remembered Set中，这阶段需要停顿线程，但是可并行执行。

筛选回收（Live Data Counting and Evacuation）：首先对各个Region中的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划。此阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分Region，时间是用户可控制的，而且停顿用户线程将大幅度提高收集效率

G1收集器可以实现在基本不牺牲吞吐量的前提下提前完成低停顿的内存回收，这是由于它能够极力避免全区域的垃圾收集，之前的收集器的收集范围都是整个新生代或整个老年代，而G1将整个Java堆划分为多个大小固定的独立区域（Region），并且跟踪这些区域里面的垃圾堆积程度，在后台维护一个优先列表，每次根据允许的收集时间，优先回收垃圾最多的区域（Garbage First名称的由来）。由于优先列表的存在，使得G1能在有限的时间内获得最高的收集效率。

基于标记-整理算法，不会产生空间碎片

**如何确保新生代对象被老年代引用的时候不被gc？**（如创建了一个static map，这个map不会被回收，经历几代GC后，该map会进入old区，而程序会不断new object put到map，那么当minor gc时，新的缓存对象是不能被gc的，此过程中，GC算法怎么避免扫描old区？）

机制：当老年代存活对象多时，每次minor gc查询老年代所有对象影响gc效率（因为gc stop-the-world），所以在老年代有一个write barrier（写屏障）来管理的card table（卡表），card table存放了所有老年代对象对新生代对象的引用。

所以每次minor gc通过查询card table来避免查询整个老年代，以此来提高gc性能。

## 内存分配与回收策略

1. 对象优先在Eden区分配，当Eden没有足够内存进行分配时，将发生Minor GC。
2. 大对象直接进入老年代。通过设置参数指定对象大小达到参数值时直接进入老年代（serial和parnew有效）
3. 长期存活的对象将进入老年代。对象在Eden分配开始（年龄为0），每次在survivor区存活一次，年龄就加1，当年龄到达15（默认）就进入老年代
4. 动态对象年龄判定。在survivor区中相同年龄的所有对象大小总和大于survivor区的一半，则年龄等于或大于该年龄的对象就直接进入老年代
5. 空间分配担保：在minor GC之前，虚拟机会检查老年代的最大可用连续内存是否大于新生代所有对象大小之和，如果条件成立，则任务minor GC是安全的，可以正常进行。否则，就要检查参数设置是否允许空间分配担保，如果不允许，则直接进行Full GC，如果允许，则要判断老年代的最大可用连续内存是否大于历次晋升到老年代的对象的平均大小，条件满足则minor GC，否则还是full GC。

# 虚拟机监控与故障处理工具

## 命令行工具

### 1.jps

jps主要用来输出JVM中运行的进程状态信息

jps [options] [hostid] 如果不指定hostid就默认为当前主机或服务器

### 2.jstack

### 3.jmap

用于生成堆转储dump文件和查看堆内存信息

### 4.jstat

jstat [ generalOption | outputOptions vmid [interval[s|ms] [count]] ]

vmid是Java虚拟机ID，在Linux/Unix系统上一般就是进程ID。interval是采样时间间隔。count是采样数目。比如下面输出的是GC信息，采样时间间隔为250ms，采样数为4：

```
[pboss2@orallyg ~]$ jstat -gc 14429 250 4
   SOC    S1C    S0U    S1U    EC    EU      OC      OU      PC      PU      YGC      YGCT      FGC      FGCT      GCT
103936.0 104448.0  0.0   512.0 2351104.0 815589.0 280576.0 98528.7 131072.0 101473.5 15  0.815  1  0.460  1.274
103936.0 104448.0  0.0   512.0 2351104.0 815589.0 280576.0 98528.7 131072.0 101473.5 15  0.815  1  0.460  1.274
103936.0 104448.0  0.0   512.0 2351104.0 815589.0 280576.0 98528.7 131072.0 101473.5 15  0.815  1  0.460  1.274
103936.0 104448.0  0.0   512.0 2351104.0 815589.0 280576.0 98528.7 131072.0 101473.5 15  0.815  1  0.460  1.274
```

1. SOC、S1C、S0U、S1U: Survivor 0/1区容量 (Capacity) 和使用量 (Used)
2. EC、EU: Eden区容量和使用量
3. OC、OU: 年老代容量和使用量
4. PC、PU: 永久代容量和使用量
5. YGC、YGT: 年轻代GC次数和GC耗时
6. FGC、FGCT: Full GC次数和Full GC耗时
7. GCT: GC总耗时

### 5.jhat

与jmap搭配使用，分析转储文件dump

## 可视化工具

### 1.jconsole

### 2.jvisualvm

## 线上排查方式

1.jmap -heap [pid] 查看是否内存本身分配不够。展示堆大小设置（年轻代，老年代，永久代）及使用情况

2.jmap -histo:live [pid] | more 找到最耗内存的对象.会以表格的形式显示存活对象的信息，并按照所占内存大小排序

展示信息包括实例数，占用内存字节大小，类名

实例：如何定位是哪个服务进程导致CPU过载，哪个线程导致CPU过载，哪段代码导致CPU过载？

1. 执行top -c，显示进程运行信息列表
2. 键入P(大写),进程按照cpu使用率排序。发现使用率最高得进程12345
3. top -Hp 12345 显示进程得线程运行信息列表

4. 键入P(大写p), 线程按照CPU使用率排序。找到cpu使用率最高得线程12336
5. printf "%x\n" 12336 变成16进制3030
6. jstack 12345 | grep '0x3030' -C5 --color 查看堆栈信息

## 类文件结构

---

见深入虚拟机第六章

## 类加载机制

---

## 字节码执行引擎

---

## Java内存模型

---

## Spring框架

---

### Spring IOC

---

### Spring AOP

---

### Spring MVC

---

### servlet

---

## Spring Boot

---

## Spring Cloud

---

## Shiro

---

## CAS 单点登录

---

## Mysql

---

## 事务

---

### 原子性 (Atomicity)

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，因此事务的操作如果成功就必须完全应用到数据库，如果操作失败则不能对数据库有任何影响。

### 一致性 (Consistency)

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。举例来说，假设用户A和用户B两者的钱加起来一共是1000，那么不管A和B之间如何转账、转几次账，事务结束后两个用户的钱相加起来应该还得是1000，这就是事务的一致性（事务执行到一半数据库崩溃也没事，因为没提交）

## 隔离性 (Isolation)

隔离性是当多个用户并发访问数据库时，比如同时操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。关于事务的隔离性数据库提供了多种隔离级别，稍后会介绍到。

## 持久性 (Durability)

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务已经正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成。否则的话就会造成我们虽然看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。这是不允许的。

# 事务的隔离级别

---

## 读未提交

事务能够看到其他事务没有提交的修改，当另一个事务又回滚了修改的情况又被称为脏读dirty read。

## 读已提交

事务能够看到其他事务提交后的修改，这时会出现一个事务内两次读取数据可能因为其他事务提交的修改导致不一致的情况，称为不可重复读。（一个事务从开始直到提交前，所做的任何修改对其他事务都是不可见的）

## 可重复读

保证了在同一事务中，多次读取同样记录的结果是一致的。

## 串行化

强制事务串行执行

## 读分类

---

## 脏读

事务能够看到其他事务没有提交的修改，当另一个事务又回滚了修改的情况。读到了未提交的数据

## 不可重复读

事务能够看到其他事务提交后的修改，这时会出现一个事务内两次读取数据可能因为其他事务提交的修改导致不一致的情况，（update导致）

## 幻读

读到数据不存在，但执行插入却提示已存在。比如事务A 先查询主键为2的数据，没查到。此时事务B插入了主键为2的数据。然后事务A因为没查到，就插入主键为2的数据，然后报错已存在了。

PS

- 不可重复读 主要是说多次读取一条记录，发现该记录中某些列值被修改过(强调update)。
- 而 幻读 主要是说多次读取一个范围内的记录(包括直接查询所有记录结果或者做聚合统计)，发现结果不一致 (InnoDB部分解决了幻读，通过快照读解决了，但是插入的时候会报错，没彻底解决)

隔离级别	脏读可能性	不可重复读可能性	幻读可能性	加锁读
READ UNCOMMITTED	Yes	Yes	Yes	No
READ COMMITTED	No	Yes	Yes	No
REPEATABLE READ	No	No	Yes	No
SERIALIZABLE	No	No	No	Yes

## MVCC多并发版本控制

### 概念

MVCC (multi version concurrency control) 即多版本并发控制，其作用就是在让特定隔离级别的事务在并发时，保证在事务中能实现一致性读，虽然锁可以控制并发操作，但是锁的系统开销比较大，而MVCC可以在大多数情况下替代行级锁，同时可以降低系统开销。

在MVCC并发控制中，读操作可以分成两类：快照读(snapshot read)与当前读(current read)。快照读，读取的是记录的可见版本(有可能是历史版本)，不用加锁。当前读，读取的是记录的最新版本，并且，当前读返回的记录，都会加上锁，保证其他事务不会再并发修改这条记录。

**读锁：**也叫共享锁、S锁，若事务T对数据对象A加上S锁，则事务T可以读A但不能修改A，其他事务只能再对A加S锁，而不能加X锁，直到T释放A上的S锁。这保证了其他事务可以读A，但在T释放A上的S锁之前不能对A做任何修改。

**写锁：**又称排他锁、X锁。若事务T对数据对象A加上X锁，事务T可以读A也可以修改A，其他事务不能再对A加任何锁，直到T释放A上的锁。这保证了其他事务在T释放A上的锁之前不能再读取和修改A。

**表锁：**操作对象是数据表。Mysql大多数锁策略都支持，是系统开销最低但并发性最低的一个锁策略。事务t对整个表加读锁，则其他事务可读不可写，若加写锁，则其他事务增删改都不行。

**行级锁：**操作对象是数据表中的一行。是MVCC技术用的比较多的。行级锁对系统开销较大，但处理高并发较好。

### 快照读VS当前读

快照读：简单的select操作，属于快照读，不加锁。

```
select * from table where ?;
```

当前读：特殊的读操作，插入/更新/删除操作，属于当前读，需要加锁。

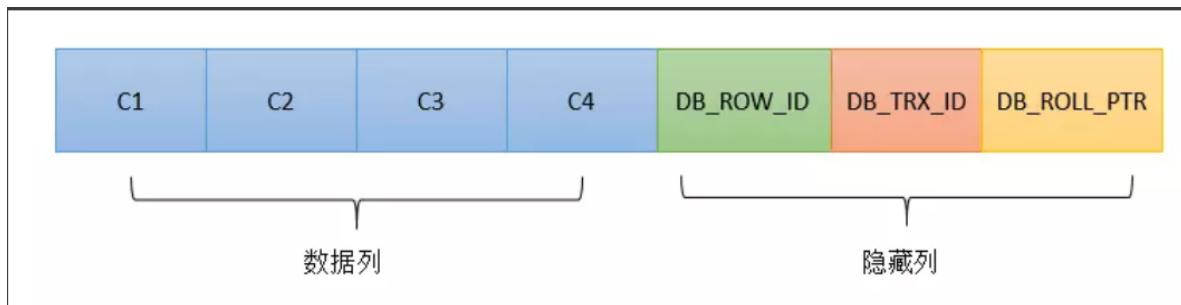
```
select * from table where ? lock in share mode;
select * from table where ? for update;
insert into table values (...);
update table set ? where ?;
delete from table where ?;
```

所有以上的语句，都属于当前读，读取记录的最新版本。并且，读取之后，还需要保证其他并发事务不能修改当前记录，对读取记录加锁。其中，除了第一条语句，对读取记录加S锁(共享锁)外，其他的操作，都加的是X锁(排它锁)。

### 原理

## 隐藏的列

innodb 向数据库中存储的每行添加三个隐藏字段（有的书上说是两个，但是我看官方文档说是三个）。



### 1 DB\_TRX\_ID 事务id

占6字节，表示这一行数据最后插入或修改的事务id。此外删除在内部也被当作一次更新，在行的特殊位置添加一个删除标记（记录头信息有一个字节存储是否删除的标记）。

### 2 DB\_ROLL\_PTR 回滚指针

占7字节，回滚指针指向被写在 `Rollback segment` 中的undoLog记录，在该行数据被更新的时候，undoLog 会记录该行修改前内容到undoLog。

### 3 DB\_ROW\_ID 行ID

占7字节，他就项自增主键一样随着插入新数据自增。如果表中不存主键 或者 唯一索引，那么数据库就会采用 DB\_ROW\_ID 生成聚簇索引。否则 DB\_ROW\_ID 不会出现在索引中。

## undo log

MySQL Innodb中存在多种日志，除了错误日志、查询日志外，还有很多和数据持久性、一致性有关的日志。

**binlog**，是mysql服务层产生的日志，常用来进行数据恢复、数据库复制，常见的mysql主从架构，就是采用slave同步master的binlog实现的，另外通过解析binlog能够实现mysql到其他数据源（如ElasticSearch）的数据复制。

**redo log** 记录了数据操作在物理层面的修改，mysql中使用了大量缓存，缓存存在于内存中，修改操作时会直接修改内存，而不是立刻修改磁盘，当内存和磁盘的数据不一致时，称内存中的数据为脏页（dirty page）。为了保证数据的安全性，事务进行中时会不断的产生redo log，在事务提交时进行一次flush操作，保存到磁盘中，redo log是按照顺序写入的，磁盘的顺序读写的速度远大于随机读写。当数据库或主机失效重启时，会根据redo log进行数据的恢复，如果redo log中有事务提交，则进行事务提交修改数据。这样实现了事务的原子性、一致性和持久性。（回滚也会记录到redo log）

**Undo Log:** 除了记录redo log外，当进行数据修改时还会记录undo log，undo log用于数据的撤回操作，它记录了修改的反向操作，比如，插入对应删除，修改对应修改为原来的数据，通过undo log可以实现事务回滚，并且可以根据undo log回溯到某个特定的版本的数据，实现MVCC

undo log 在 `Rollback segment` 中又被细分为 insert 和 update undo log，insert 类型的undo log 仅用于事务回滚，当事务一旦提交，insert undo log 就会被丢弃（提交后就无用了）。update 的 undo log（包括update和delete，在快照读的时候需要所以不会立刻删除）被用于一致性的读和事务回滚，update undo log 的清理是在没有事务需要对这部分数据快照进行一致性读的时候进行清理。

（undo log是为回滚而用，具体内容就是copy事务前的数据库内容（行）到undo buffer，在适合的时间把undo buffer中的内容刷新到磁盘。undo buffer与redo buffer一样，也是环形缓冲，但当缓冲满的时候，undo buffer中的内容会也会被刷新到磁盘；与redo log不同的是，磁盘上不存在单独的undo log文件，所有的undo log均存放在主ibd数据文件中（表空间），即使客户端设置了每表一个数据文件也是如此。）

undo log 的创建

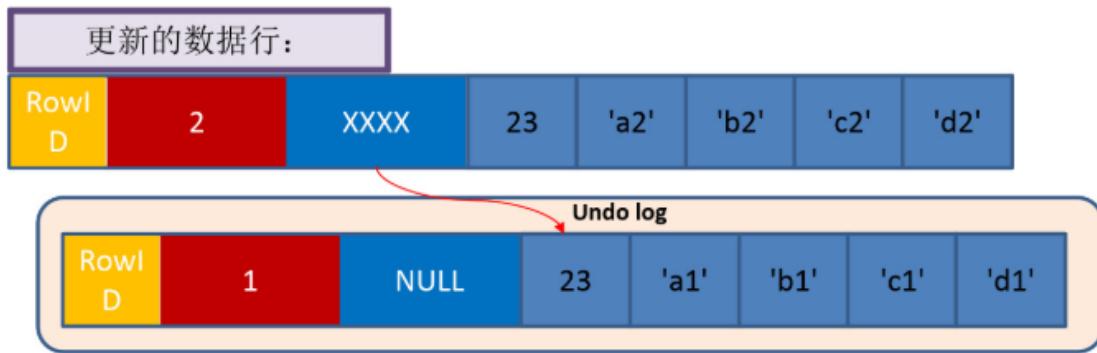
每次对数据进行更新操作时，都会copy 当前数据,保存到undo log 中。并修改 当前行的 回滚指针指向 undo log 中的 旧数据行。

事务1: `insert t(id,col1,col2,col3,col4) values(23,'a1','b1','c1','d1');`

事务ID DB_TRX_ID	回滚指针 DB_ROLL_PT	id	col1	col2	col3	col4	
Row1 D	1	NULL	23	'a1'	'b1'	'c1'	'd1'

行上有三个隐含字段：分别对应该行的rowid、事务号和回滚指针，  
id、Col1~Col4是表各列的名字，23、'a1'~'d1'是其对应的数据。

事务2: `update table t set col1='a2', col2='b2', col3='c2', col4='d2' where id=23;`



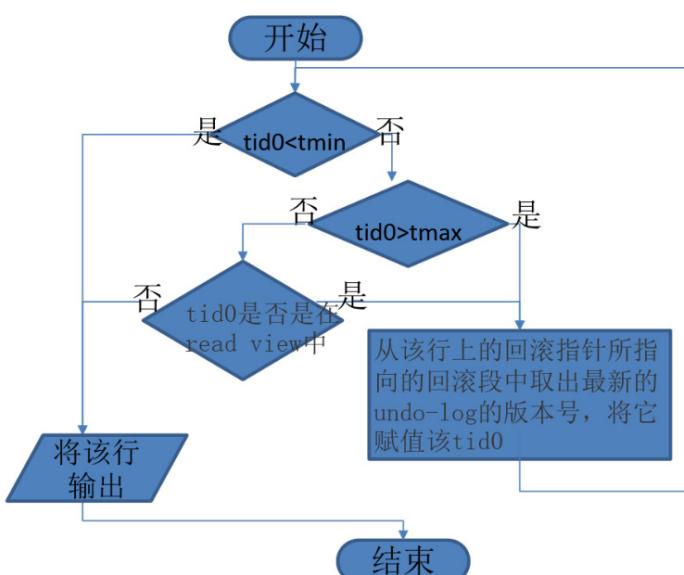
- 用排他锁锁定该行
- 把该行修改前的值Copy到undo log中。
- 修改当前行的值，填写事务编号，使回滚指针指向undo log中的修改前的行
- 记录redo log，包括undo log中的变化

## InnoDB可见性

- 可见性比较的方法：
  - 并不是用当前事务ID与表中各个数据行上的事务ID去比较的
  - 在每个事务开始的时候，会将当前系统中的所有的活跃事务拷贝到一个列表中 (read view)，根据read view最早一个事务ID和最晚的一个事务ID来做比较的，这样就能确保在当前事务之前没有提交的所有事务的变更以及后续新启动的事务的变更，在当前事务中都是看不到的。
  - 当然，当前事务自身的变更还是需要看到的。

当开始一个事务时，把当前系统中活动的事务的ID都拷贝到一个列表 (read view)中，这个列表中最早的事务ID为tmin，最晚的事務ID为tmax

当读到一行时，该行上当前事务id为tid0，当前行是否可见的判断逻辑见右图



(可重复读级别) 假设当前 数据行 事务ID 为 T0 , read view 中保存的 最老的事务id T\_min ,最新的 事务id 为 T\_max,当前进行的事务id 为 T\_new 。

- 如果  $T_0 < T_{min}$  ,那么该行数据可见。  
因为  $T_0$  在  $T_{new}$  事务开始前 已经提交。
- 如果  $T_0 > T_{max}$  ,数据行不可见。根据 DB\_ROLL\_PTR 指针 找到下一个 数据版本，再次进行数据可见性判断。  
因为  $T_0$  事务 在  $T_{new}$  开始前并不存在，也就是说  $T_0$  在  $T_{new}$  开始后 创建。
- 如果  $T_{min} \leq T_0 \leq T_{max}$  , 判断  $T_0$  是否在read\_view 中，如果 不在，该行数据可见。如果不可见根据 DB\_ROLL\_PTR 指针 找到下一个 数据版本，再次进行数据可见性判断。

#### READ-COMMITTED

事务内的每个查询语句都会重新创建Read View，这样就会产生不可重复读现象发生

#### REPEATABLE-READ

事务内开始时创建Read View，在事务结束这段时间内 每一次查询都不会重新重建Read View，从而实现了可重复读 (MVCC只使用RC 和RR) 。

注意，ReadView是与SQL绑定的，而并不是事务，所以即使在同一个事务中，每次SQL启动时构造的ReadView的up\_trx\_id和low\_trx\_id也都是不一样的，至于DATA\_TRX\_ID大于low\_trx\_id本身出现也只有当多个SQL并发的时候，在一个SQL构造完ReadView之后，另外一个SQL修改了数据后又进行了提交，对于这种情况，数据其实是不可见的。

所以案例分析

session 1	session 2
select a from test; return a = 10	
start transaction;	
update test set a = 20;	
	start transaction;
	select a from test; return ?
commit;	
	select a from test; return ?

读已提交：事务2第一次读到的是10，第二次在新的read view比较后，读的是20；可重复读，则始终是10

## innodb加锁机制

一个Update操作的具体流程。当Update SQL被发给MySQL后，MySQL Server会根据where条件，读取第一条满足条件的记录，然后InnoDB引擎会将第一条记录返回，并加锁 (current read)。待MySQL Server收到这条加锁的记录之后，会再发起一个Update请求，更新这条记录。一条记录操作完成，再读取下一条记录，直至没有满足条件的记录为止。因此，Update操作内部，就包含了一个当前读。同理，Delete操作也一样。Insert操作会稍微有些不同，简单来说，就是Insert操作可能会触发Unique Key的冲突检查，也会进行一个当前读。

### Repeatable Read (RR)

针对当前读，RR隔离级别保证对读取到的记录加锁 (记录锁)，同时保证对读取的范围加锁，新的满足查询条件的记录不能够插入 (间隙锁)。（范围查询造成间隙）

InnoDB中加锁的方法是锁住对应的索引，针对不同索引有不同处理方式：

delete from t1 where id = 10;

1.id是主键 (聚簇索引)，则添加排他锁 (X锁)

2.id是唯一索引，则对当前索引加排他锁，并且将对应的聚簇索引的行加上排他锁

3.id是普通索引，首先，通过id索引定位到第一条满足查询条件的记录，在记录上的X锁，在GAP上的GAP锁，然后加主键聚簇索引上的记录X锁，然后返回；然后读取下一条，重复进行。直至进行到第一条不满足条件的记录[11,f]，此时，不需要加记录X锁，但是仍旧需要加GAP锁，最后返回结束。

4.id没有索引，那就完蛋了；会锁上表中的所有记录，同时会锁上聚簇索引内的所有GAP，杜绝所有的并发更新/删除/插入操作。当然，也可以通过触发semi-consistent read，来缓解加锁开销与并发影响，但是semi-consistent read本身也会带来其他问题，不建议使用。

## 分布式事务

# 索引详解

## 定义

索引(Index)是帮助MySQL高效获取数据的数据结构.可以得出索引的本质就是数据结构。在数据之外,数据库还维护着满足特定查找算法的数据结构,这些数据结构以某种方式引用(指向)数据,这样就可以在这些数据结构的基础上实现高级查找算法,这种数据结构就是索引。

一般来说索引本身很大,不适合全部存储在内存中,因此索引往往以索引文件的形式存储在磁盘上。

我们平常所说的索引,如果没有特别指明,都是指B树(多路搜索树,并不一定是二叉的)结构组织的索引,其中聚集索引、次要索引、覆盖索引、复合索引、前缀索引,唯一索引默认都是使用B+树索引,统称索引.当然,除了B+树这种类型的索引之外,还有哈希索引(hash index)等

## 优点

类似大学图书馆建书目录索引,提高数据检索的效率,降低数据库的IO成本。通过索引列对数据进行排序,降低数据排序成本,降低了CPU的消耗同时也可以加速表和表之间的连接。

## 缺点

实际上索引也可以看成一张表,该表保存了主键与索引字段,并指向实体表的记录,所以索引也是要占内存空间的。

虽然索引大大提高了查询速度,但同时也会降低更新表的速度,比如对表进行insert,update和delete的操作,因为更新表时,MySQL不仅要保存数据,还要保存索引文件的每次更新,比如因为更新所带来的键值变化后的索引信息。

索引只是高效的一个因素,如果你的MySQL有大数据量的表,就需要花时间研究建立最优秀的索引,或优化查询方法。

## 分类

### 聚集索引 (聚簇索引)

定义: 数据行的物理顺序与列值 (一般是主键的那一列) 的逻辑顺序相同,一个表中只能拥有一个聚集索引。

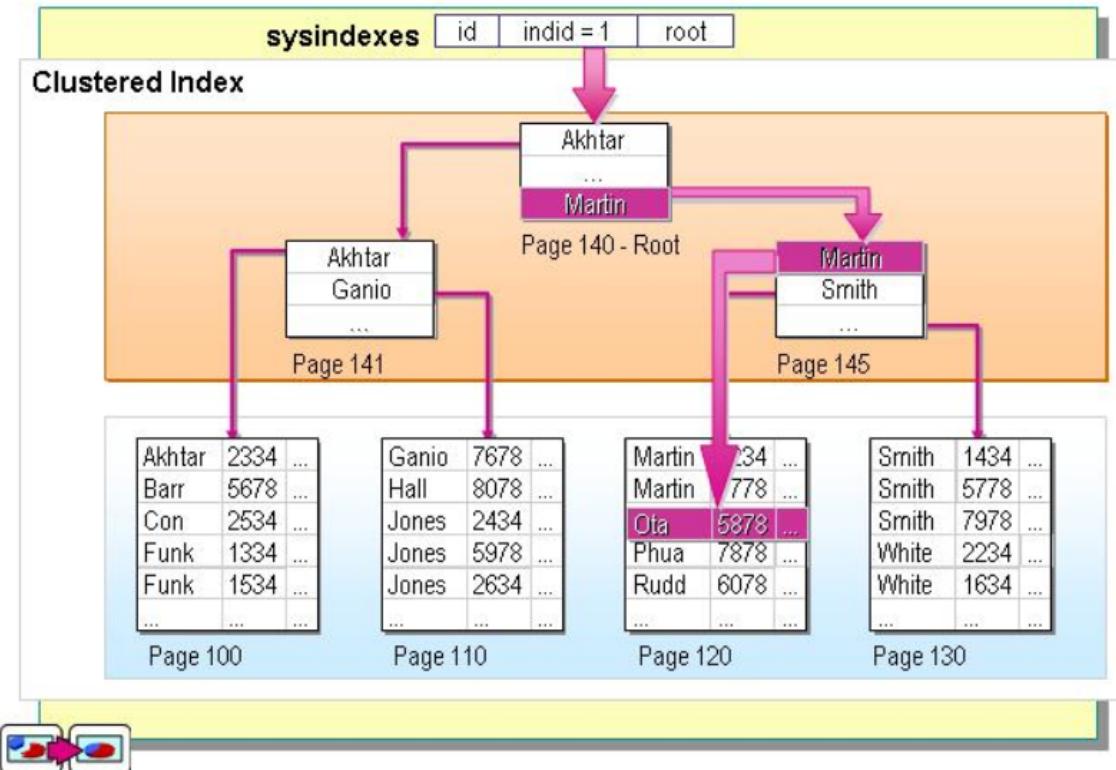
打个比方,一个表就像是我们以前用的新华字典,聚集索引就像是拼音目录,而每个字存放的页码就是我们的数据物理地址,我们如果要查询一个“哇”字,我们只需要查询“哇”字对应在新华字典拼音目录对应的页码,就可以查询到对应的“哇”字所在的位置,而拼音目录对应的A-Z的字顺序,和新华字典实际存储的字的顺序A-Z也是一样的,如果我们中文新出了一个字,拼音开头第一个是B,那么他插入的时候也要按照拼音目录顺序插入到A字的后面,现在用一个简单的示意图来大概说明一下在数据库中的样子:

地址	id	username	score
0x01	1	小明	90
0x02	2	小红	80
0x03	3	小华	92
..	..	..	..
0xff	256	小英	70

注: 第一列的地址表示该行数据在磁盘中的物理地址,后面三列才是我们SQL里面用的表里的列,其中id是主键,建立了聚集索引。

结合上面的表格就可以理解这句话了吧：数据行的物理顺序与列值的顺序相同，如果我们查询id比较靠后的数据，那么这行数据的地址在磁盘中的物理地址也会比较靠后。而且由于物理排列方式与聚集索引的顺序相同，所以也就只能建立一个聚集索引了。下图是实际存放示意图

## Finding Rows in a Clustered Index



从上图可以看出聚集索引的好处了，索引的叶子节点就是对应的数据节点（MySQL的MyISAM除外，此存储引擎的聚集索引和非聚集索引只多了个唯一约束，其他没什么区别），可以直接获取到对应的**全部列**的数据，而非聚集索引在索引没有覆盖到对应的列的时候需要进行**二次查询**，后面会详细讲。因此在查询方面，聚集索引的速度往往会更占优势。

### 创建聚集索引

如果不创建索引，系统会自动创建一个隐含列作为表的聚集索引。

创建表的时候指定主键（注意：SQL Server默认主键为聚集索引，也可以指定为非聚集索引，而MySQL里主键就是聚集索引）

```
create table t1(
    id int primary key,
    name nvarchar(255)
)
```

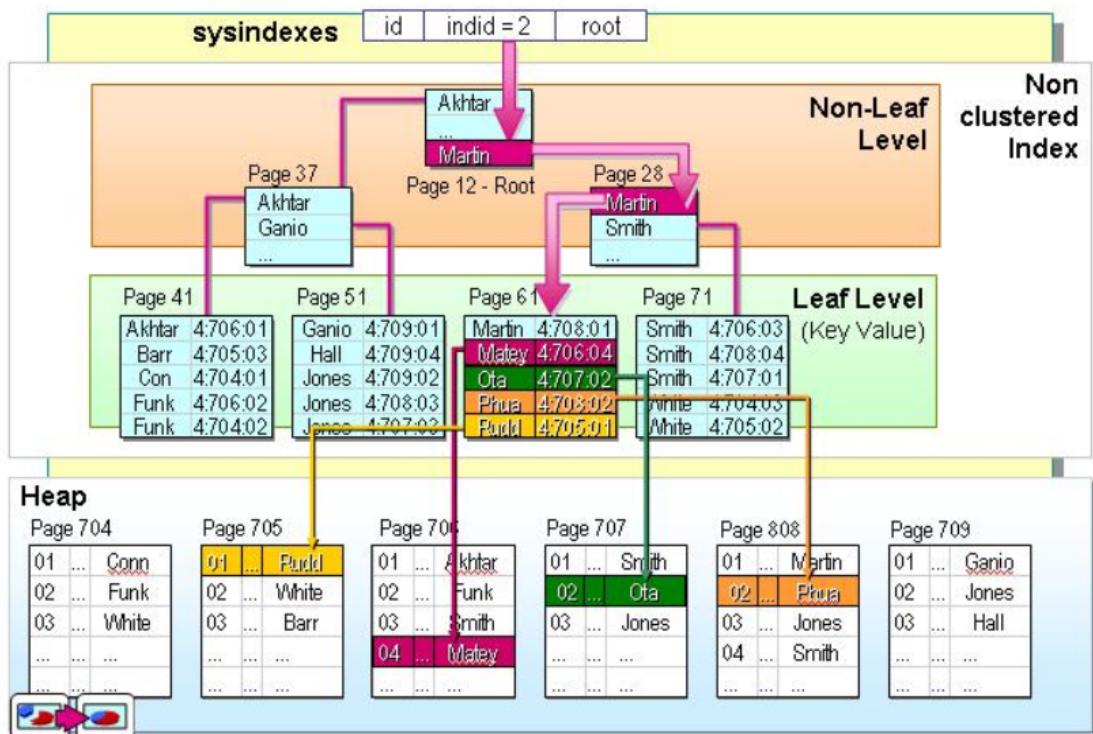
值得注意的是，最好还是在创建表的时候添加聚集索引，由于聚集索引的物理顺序上的特殊性，因此如果再在上面创建索引的时候会根据索引列的排序移动全部数据行上面的顺序，会非常地耗费时间以及性能。所以主键一般使用自增整形。

### 非聚集索引

定义：该索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同，一个表中可以拥有多个非聚集索引。

按照定义，除了聚集索引以外的索引都是非聚集索引，只是人们想细分一下非聚集索引，分成**普通索引**，**唯一索引**，**全文索引**。如果非要把非聚集索引类比成现实生活中的东西，那么非聚集索引就像新华字典的偏旁字典，他结构顺序与实际存放顺序不一定一致。

# Finding Rows in a Heap with a Nonclustered Index



非聚集索引叶子节点仍然是索引节点，只是有一个指针指向对应的数据块，此如果使用**非聚集索引查询**，而查询列中包含了其他该索引没有覆盖的列，那么他还要进行第二次的查询，查询节点上对应的数据行的数据。

如有以下表t1：

id	username	score
1	小明	90
2	小红	80
3	小华	92
..	..	..
256	小英	70

该表有聚集索引|clustered index(id), 非聚集索引|index(username)

使用以下语句进行查询，不需要进行二次查询，直接就可以从非聚集索引的节点里面就可以获取到查询列的数据。

```
select id, username from t1 where username = '小明'
select username from t1 where username = '小明'
```

但是使用以下语句进行查询，就需要二次的查询去获取原数据行的score：

```
select username, score from t1 where username = '小明'
```

所以要注意的是非聚集索引其实叶子节点除了会存储索引覆盖列的数据，也会存放聚集索引所覆盖的列数据。

## 如何解决二次查询问题

使用复合索引（覆盖索引），建立两列以上的索引，即可查询复合索引里的列的数据而不需要进行回表二次查询，如index(col1, col2)，执行下面的语句：

```
select col1, col2 from t1 where col1 = '213';
```

要注意使用复合索引需要满足最左侧索引的原则，也就是查询的时候如果where条件里面没有最左边的一到多列，索引就不会起作用。

小结：

使用聚集索引的查询效率要比非聚集索引的效率要高，但是如果需要频繁去改变聚集索引的值，写入性能并不高，因为需要移动对应数据的物理位置。

非聚集索引在查询的时候可以的话就避免二次查询，这样性能会大幅提升。

不是所有的表都适合建立索引，只有数据量大表才适合建立索引，且建立在选择性高的列上面性能会更好。

## 基本语法

创建：create [unique] index indexName on tb\_name(column\_name)

添加：alter table tb\_name add [unique] index [indexName] on (columnname)

删除：drop index [indexName] on tb\_name;

## 创建索引场景

- 主键自动建立唯一索引
- 频繁作为查询条件的字段应该创建索引
- 查询中与其他表关联的字段,外键关系建立索引
- 频繁更新的字段不适合创建索引,因为每次更新不单单是更新数据还会更新索引
- Where条件里用得到的字段适合创建索引
- 单键/组合索引的选择问题,在高并发下倾向创建组合索引
- 查询中排序的字段,排序字段若通过索引去访问将大大提高排序速度
- 查询中统计或者分组字段

## 不适合创建索引场景

- 表记录太少(一般生产环境下,三百万条记录性能就可能开始下降,官方说的是五百万到八百万)
- 经常增删改的表
- 某个数据列的值包含许多重复的内容

## 复合索引场景分析

组合索引查询的各种场景 兹有 Index (A,B,C) ——组合索引多字段是有序的，并且是个完整的BTree 索引。

可以用上该组合索引	不能用上组合索引	用上部分组合索引
A>5	B>5	A>5 AND B=2
A=5 AND B>6	B=6 AND C=7	A=5 AND B>6 AND C=2
A=5 AND B=6 AND C=7		
A=5 AND B IN (2,3) AND C>5		

## 索引注意事项

- 建什么索引用什么索引,顺序也最好保持一致
- 最佳左前缀索引名称命名（如字段name,age,city,则索引命名应该是nameAgeCity或者xxx\_nameAgeCity,顺序很重要）
- 不在索引列上做任何操作（计算, 函数, or, 类型转换）, 会导致索引失效而转向全表扫描
- 存储引擎不能使用索引中范围条件右边的列（如name='lin' and age>25 and city='qingdao',则age后面的索引会失效）
- 尽量使用覆盖索引（只访问索引的查询（索引列和要查询的列一致））, 减少select \*
- MySQL在使用不等于（!=或者<>）的时候无法使用索引会导致全表扫描
- is null, is not null 也无法使用索引
- like以通配符在这（'%abc','%abc%'）两种情况会索引失效变成全表扫描, 'abc%'则不会, 若要'%abc','%abc%'不失效, 建议使用覆盖索引, 且查询的字段要少于索引或者与索引一致, 不使用select \*。如为name, age, city建了索引, 请这么使用:select name或者select age,或者select city或者select name,age,city。如果select name,age,city,email则会全表扫描
- 字符串不加引号索引失效
- 少用or, 用他来连接时索引会失效
- select \* from A where exists (select 1 from where b.id=A.id)#当A表的数据系小于B表时, 用exists优于in
- 使用join代替子查询

## 高并发查询并插入

不存在就插入, 存在则无操作

1. 使用联合唯一索引, 直接插入, 成功则没问题, 失败表示已被插入。比如user\_id和del\_flag联合唯一索引, 对于逻辑删除可以把del\_flag修改成user\_id, 而非1, 保证索引的唯一性质。
2. 使用缓存, 把要插入的数据的唯一性标识（也可以是多字段拼接）先插入缓存, 插入成功就操作数据库, 失败就表明有其他线程插入过了。
3. 使用去重表, 也就是利用唯一索引的性质。先插入去重表, 成功就继续插入, 失败表明有其他线程插入过了

## 高并发查询并更新

不存在无操作, 存在就更新, 保证更新的安全性（就是单纯的查询更新）

1. 乐观锁, 增加字段ver

```
SELECT user, ver FROM table_b WHERE c=0
UPDATE table_b SET ver=4,user='xx' WHERE c=0 AND ver=3
//如果update的记录数为1，则表示成功;
//如果update的记录数为0，则表示已经被其他应用（线程）update过了，需作异常处理
-----
不用ver，使用代表任务状态的字段代替ver也是可以的。
```

## 2.乐观锁，查出旧值并使用

```
SELECT money FROM t_yue WHERE uid=123
UPDATE t_yue SET money=28 WHERE uid=123 AND money=100
```

## 存在更新不存在插入

不存在就插入，存在就更新(这种场景一般意味着会有覆盖，一个线程会插入，但是另一个线程又会立马更新)

mysql的ON DUPLICATE KEY UPDATE (对于唯一索引的字段，这个字段值已存在就更新这行数据，不存在就插入新数据) 可以实现不存在则插入，存在则更新。但是某些场景不安全。比如上面的money，原本是100，一个线程消费72，更新为28，另一个线程消费50，更新为50，就不可以使用ON DUPLICATE KEY UPDATE，因为会直接覆盖。

ps:

```
// index是唯一索引，id是主键
insert into test(`Id`, `Index`, `IntVal`) values (1, 1, 2);
insert into test(`Id`, `Index`, `IntVal`) values (2, 1, 2) ON DUPLICATE KEY
UPDATE IntVal=10;
```

## B+树比B树更适合索引？

1、**B+树的磁盘读写代价更低：**B+树的内部节点并没有指向关键字具体信息的指针，因此其内部节点相对B树更小，如果把所有同一内部节点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多，一次性读入内存的需要查找的关键字也就越多，相对IO读写次数就降低了。（就是为了减少磁盘io次数，因为b+树所有最终的子节点都能在叶子节点里找见，所以非叶子节点只需要存索引范围和指向下一级索引（或者叶子节点）的地址就行了，不需要存整行的数据，所以占用空间非常小，直到找到叶子节点才加载进来整行的数据。）

2、**B+树的查询效率更加稳定：**由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

3、由于B+树的数据都存储在叶子结点中，分支结点均为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，所以通常B+树用于数据库索引。

## linux相关配置

## Explain详解

## 作用

查看sql的执行计划，帮助我们分析mysql是如何解析sql语句的。其中与效率息息相关的则是type,key,ref,rows。而id,select\_type,table 用于定位查询，表示本行参数所对应的sql查询部分

1. 查看表的加载顺序。
2. 查看sql的查询类型。
3. 哪些索引可能被使用，哪些索引又被实际使用了。
4. 表之间的引用关系。
5. 一个表中有多少行被优化器查询。
6. 其他额外的辅助信息。

## ID详解

含义：select查询的序列号，是一组数字，表示的是查询中执行select子句或者是操作表的顺序。

id的情况有三种，分别是：

1. id相同表示加载表的顺序是从上到下。
2. id不同id值越大，优先级越高，越先被执行。
3. id有相同，也有不同，同时存在。id相同的可以认为是一组，从上往下顺序执行；在所有的组中，id的值越大，优先级越高，越先执行。

## select\_type

1. SIMPLE: 指示非子查询和union的简单查询。`SELECT * FROM t_user`

2. PRIMARY:指示在有子查询的语句中最外面的select,主查询。

```
SELECT * FROM t_user WHERE fuid IN (SELECT fuid FROM TABLE)
```

3. UNION:指示在使用UNION语句的第二个或者后面的select。

```
SELECT * FROM teachers UNION SELECT * FROM students
```

4. DEPENDENT UNION: 子查询union语句的第二个或后面的select。

5. UNION RESULT : union语句的结果集。

```
SELECT * FROM t_user WHERE fuid IN      DEPENDENT UNION  
  (SELECT fuid FROM teacher UNION SELECT fuid FROM students);
```

结果集 UNION RESULT

6. SUBQUERY: 子查询中的语句。与union相反理解就行了。

7. DEPENDENT SUBQUERY: 子查询中第一个语句。

```
SELECT * FROM t_user WHERE fuid IN (SELECT fuid FROM teacher);
```

8. DERIVED: 派生表的SELECT(FROM子句的子查询)。

```
SELECT * FROM (SELECT fuid, fname, fustate FROM t_user);
```

## table

就是表名，本行记录对应查询所应用的表。可以通过table表名帮助定位查询。

使用explain时，有时table字段显示的并不是表名，而是 derived2 或 derived3 等等。。。即 derived x 代表的是id为x的查询所得的结果集。

```

EXPLAIN
SELECT u.fuid,u.fuser_name,ul.fphone FROM t_user u
INNER JOIN
(SELECT fuserId,fphone FROM t_user_login_log WHERE fuserId=2) ul
ON u.fuid=ul.fuserId

```

explain后所得：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	(NULL)	(NULL)	(NULL)	(NULL)	2519 (NULL)	
1	PRIMARY	u	eq_ref	PRIMARY	PRIMARY	4	ul.fuserId	1	Using where
2	DERIVED	t_user_login_log	ALL	(NULL)	(NULL)	(NULL)	(NULL)	2519	Using where

由以上内容可知，id为2的记录对应inner join后的子查询1，第一条记录所对应的table参数值derived2即为子查询1的结果集。

案例：

1)

```

1   EXPLAIN
2   SELECT
3     u.fuid,
4     u.fuser_name,
5     ul.fphone
6   FROM
7     t_user u
8   INNER JOIN t_user_login_log ul
9   ON u.fuid = ul.fuserId

```

本例中没有子查询，所以explain后将返回两条简单查询记录

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ul	ALL	(NULL)	(NULL)	(NULL)	(NULL)	2519 (NULL)	
1	SIMPLE	u	eq_ref	PRIMARY	PRIMARY	4	db_brand.ul.fuserId	1	Using where

两条select\_type均为SIMPLE，第一条table为ul即t\_user\_login\_log表的代号，第二条则是通过fuserId在user\_login\_log表中获取fphone的过程。

2)

将上例中sql语句改写成如下等效sql:

```

EXPLAIN
SELECT u.fuid,u.fuser_name,(SELECT fphone FROM t_user_login_log WHERE fuserId=u.fuid) AS phone FROM t_user u

```

本条sql与上条sql效果相同，但是加入了子查询，故而返回参数与上个案例完全不同：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	u	index	(NULL)	idx_nickname_phone	194	(NULL)	71550	Using index
2	DEPENDENT SUBQUERY	t_user_login_log	ALL	(NULL)	(NULL)	(NULL)	(NULL)	2521	Using where

select\_type为DEPENDENT SUBQUERY的记录毫无疑问就是括号内的子查询部分喽，另一条PRIMARY的记录对应的当然就是主查询 select ... from t\_user啦~

## type

连接类型。有多个参数 **重要且困难**

表示MySQL在表中找到所需行的方式,又称“访问类型”

ALL, index, range, ref, eq\_ref, const, system, NULL

从左到右,性能从最差到最好。一般来说,要保证查询至少达到range级别,最好能达到ref

- all——MySQL将遍历全表以找到匹配的行。对于每个来自于先前的表的行组合,进行完整的表扫描。如果表是第一个没标记const的表,这通常不好,并且通常在它情况下\*很\*差。通常可以增加更多的索引而不要使用ALL,使得行能基于前面的表中的常数值或列值被检索出。
- index——Full Index Scan, index与ALL区别为index类型只遍历索引树。该连接类型与ALL相同,除了只有索引树被扫描。这通常比ALL快,因为索引文件通常比数据文件小。(也就是说虽然all和

Index都是读全表，但index是从索引中读取的，而all是从硬盘中读的)下图就是遍历了主键索引|

```
mysql> explain select id from t1;
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key      | key_len | ref   | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE     | t1    | index | NULL        | PRIMARY  | 4       | NULL  | 516  | Using index |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

- range——给定范围内的检索，使用一个索引来检查行。索引范围扫描对索引的扫描开始于某一点，返回匹配值域的行。显而易见的索引范围扫描是带有between或者where子句里带有<,>查询。当mysql使用索引去查找一系列值时，例如IN()和OR列表，也会显示range(范围扫描)，当然性能上面是有差异的。
- index\_merge ——该联接类型表示使用了索引合并优化方法。在这种情况下，key列包含了使用的索引的清单，key\_len包含了使用的索引的最长的关键元素。
- ref\_or\_null ——该联接类型如同ref，但是添加了MySQL可以专门搜索包含NULL值的行。在解决子查询中经常使用该联接类型的优化。
- ref——使用非唯一索引扫描或者唯一索引的前缀扫描，返回匹配某个单独值的记录行。对于每个来自于前面的表的行组合，所有有匹配索引值的行将从这张表中读取。如果联接只使用键的最左边的前缀，或如果键不是UNIQUE或PRIMARY KEY（换句话说，如果联接不能基于关键字选择单个行的话），则使用ref。如果使用的键仅仅匹配少量行，该联接类型是不错的。（非唯一性索引扫描，返回匹配某个单独值的所有行。本质上也是一种索引访问，它返回所有匹配某个单独值的行，可能会找到多个符合条件的行，所以这个应该属于查找和扫描的混合体）

```
mysql> create index idx_col1_col2 on t1(col1,col2);
Query OK, 1000 rows affected (0.15 sec)
Records: 1000 Duplicates: 0 Warnings: 0

mysql> select count(distinct col1) from t1;
+-----+
| count(distinct col1) |
+-----+
|      7 |
+-----+
1 row in set (0.00 sec)

mysql> explain select * from t1 where col1 = 'ac';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key      | key_len | ref   | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE     | t1    | ref  | idx_col1_col2 | idx_col1_col2 | 194    | const | 224  | Using index |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

eq\_ref和ref：就好比一个班级里面，只有一个班主任和一群学生，t2返回的只有一个记录（就好像比班主任），而col1返回的是所有col1等于ac（所有名字是ac的学生）

- eq\_ref——对于eq\_ref的解释，mysql手册是这样说的：“对于每个来自于前面的表的行组合，从该表中读取一行。这可能是最好的联接类型，除了const类型。它用在一个索引的所有部分被联接使用并且索引是UNIQUE或PRIMARY KEY”。eq\_ref可以用于使用=比较带索引的列。（就是唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键或唯一索引扫描）

```
mysql> explain select * from t1, t2 where t1.id = t2.id;
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key      | key_len | ref   | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE     | t2    | ALL  | PRIMARY      | NULL    | 4       | NULL  | 639  |           |
| 1  | SIMPLE     | t1    | eq_ref| PRIMARY      | PRIMARY  | 4       | shared.t2.ID | 1   |           |
+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

- const——**最多有一个匹配行**。表示通过索引一次就找到了，const用于比较primary key或者unique索引。因为只匹配一行数据，索引很快，如将主键置于where列表中，MySQL就能将该查询转换为一个“常量”

- system——表只有一行记录(等于系统表), 这是const类型的特例, 平时不会出现

```
mysql> explain select * from (select * from t1 where id = 1) d1;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | PRIMARY | <derived2> | system | const | NULL | NULL | NULL | 1 |
| 2 | DERIVED | t1 | const | PRIMARY | PRIMARY | 4 | NULL | 1 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

## possible\_keys

- 显示可能应用在这张表中的索引, 一个或多个。
- 查询涉及到的字段是若存在索引, 则该索引将被列出, **但不一定被查询实际使用**

简而简之: possible\_keys是MySQL分析出推测可能用到的索引有哪几个, 而key最后实际用到的索引。  
(理论: 聚会中大概能来多少人和实际来多少人的区别)

## key

- 实际使用的索引, 如果为NULL, 则没有使用索引。 (要么没建索引, 要么建了索引没用, 所谓索引失效)
- 查询中若使用了覆盖索引, 则该索引仅出现在key列表中

**覆盖索引演示:**

查询中若使用了覆盖索引, 则该索引仅出现在key列表中

```
mysql> explain select col1, col2 from t1;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | index | NULL | idx_col1_col2 | 390 | NULL | 682 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

查询字段个数和顺序和建立索引的字段个数和顺序刚好吻合

```
mysql> create index idx_col1_col2 on t2 col1,col2;
```

知乎 @很six

Query OK, 1001 rows affected (0.17 sec)

Records: 1001 Duplicates: 0 Warnings: 0

- possible\_keys,key演示:

```
mysql> explain select t2.*
   -> from t1, t2, t3
   -> where t1.id = t2.id and t1.id = t3.id
   -> and t1.other_column = '';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | ref | PRIMARY | idx_t1 | 92 | const | 1 |
| 1 | SIMPLE | t3 | eq_ref | PRIMARY | PRIMARY | 4 | test.t1.ID | 1 |
| 1 | SIMPLE | t2 | eq_ref | PRIMARY | PRIMARY | 4 | test.t1.ID | 1 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

知乎 @很six

举例：理论上t1这张表应该用到PRIMARY,idx\_t1这两个索引，但是实际中却只用到了idx\_t1这一个月索引。

## key\_len

- 表示索引中使用的字节数, 可通过该列计算查询中使用的索引的长度。在不损失精确性的情况下, 长度越短越好。 (投入少, 产物丰富, 话句话说就是: 既不给马儿吃草, 又让马儿跑)
- key\_len显示的值为索引字段的最大可能长度, **并非实际使用长度**, 即key\_len是根据表定义计算而得, 不是通过表内检索获得的 (同样的查询结果, key\_len用的越少越好)

举例：假设班级的表中在名字列加上一个索引，我们要根据名字去查询名字姓李的，此时我们还想去根据城市去查询，此时的查询条件比单查名字时更精确。此时得到的结果更精确，但是却投入了更多的条件（第一次名字，第二个城市），此时的key\_len会比上一次更多。

```
mysql> desc t1;
+-----+-----+-----+-----+-----+
| Field | Type  | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| ID   | int(11) | NO   | PRI | NULL    | auto_increment |
| col1 | char(4)  | YES  | MUL | NULL    |                |
| col2 | char(4)  | YES  |      | NULL    |                |
+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> explain select * from t1 where col1 = 'ab';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id  | select_type | table | type  | possible_keys | key   | key_len | ref   | rows  | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   | SIMPLE     | t1   | ref   | idx_col1_col2 | idx_col1_col2 | 13    | const  | 143   |        |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select * from t1 where col1 = 'ab' and col2 = 'ac';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id  | select_type | table | type  | possible_keys | key   | key_len | ref   | rows  | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   | SIMPLE     | t1   | ref   | idx_col1_col2 | idx_col1_col2 | 26    | const,const | 1       |        |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.01 sec)
```

可以看到，联合索引idx\_col1\_col2，第一个查询key\_len是13，第二个是26。

## ref

显示索引的哪一列被使用了，如果可能的话，最好是一个常数。哪些列或常量被用于查找索引列上的值

```
mysql> explain select * from t1, t2 where t1.col1 = t2.col1 and t1.col2 = 'ac';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id  | table | type | possible_keys | key   | key_len | ref   | rows  |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   | t2   | ALL  | NULL          | NULL  | NULL    | NULL  | 640   |
| 1   | t1   | ref  | idx_col1_col2 | idx_col1_col2 | 26    | shared.t2.col1,const | 82   |
+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.01 sec)
```

由key\_len可知t1表的idx\_col1\_col2被充分使用，col1匹配t2表的col1，col2匹配了一个常量，即 'ac'

MySQL处理这条语句顺序，加载(t2)后加载t1，t1中ref为shared.t2.col1和const，前面代表shared库中t2表的col1字段在和t1表中的col1做关联，const代表t1表中的col2匹配了ac这个常量。

## rows

根据表统计信息及索引选用情况，大致估算出找到所需记录所需要读取的行数（这张表有多行被优化器优化过）

```

mysql> explain select * from t1, t2 where t1.id = t2.id and t2.col1 = 'ac';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t2 | ALL | PRIMARY | NULL | NULL | NULL | 640 |
| 1 | SIMPLE | t1 | eq_ref | PRIMARY | PRIMARY | 4 | shared.t2.ID | 1 |
+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

```

没有索引的时候，两张表关联后的加载顺序t2>t1，type就是上面介绍eq\_ref和ref区别是介绍的班主任(唯一扫描，只有一行匹配)，t2.col1='ac'就好比这个班级是ac，两个表加起来读取了641行。

```

mysql> create index idx_col1_col2 on t2(col1,col2);
Query OK, 1001 rows affected (0.17 sec)
Records: 1001 Duplicates: 0 Warnings: 0

mysql> explain select * from t1, t2 where t1.id = t2.id and t2.col1 = 'ac';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t2 | ref | PRIMARY, idx_col1_col2 | idx_col1_col2 | 195 | const | 142 |
| 1 | SIMPLE | t1 | eq_ref | PRIMARY | PRIMARY | 4 | shared_t2.ID | 1 |
+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

```

建立复合索引之后(idx\_col1\_col2)，所需要读取的行数为143。

## Extra

包含不适合在其他列中显示但十分重要的额外信息

- Using filesort : 说明MySQL会对数据使用一个外部的索引排序，而不是按照表内索引顺序进行读取。MySQL中无法利用索引完成的排序操作称为“文件内排序”。

```

mysql> explain select col1 from t1 where col1 = 'ac' order by col3\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: t1
        type: ref
possible_keys: idx_col1_col2_col3
      key: idx_col1_col2_col3
    key_len: 13
      ref: const
     rows: 142
    Extra: Using where; Using index; Using filesort
1 row in set (0.00 sec)

```

知乎 @很six

Extra中包含Using where和Using index,确实看到了where条件和使用了索引(idx\_col1\_col2\_col3)，还出现了Using filesort，也就说索引只是部分使用到了。我们知道索引干两件事(排序、快速查询)，查询的时候部分使用到了(key不为null并且有值，而且type是ref，ref为const)，但是排序没有用到。

```

mysql> explain select col1 from t1 where col1 = 'ac' order by col2, col3\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: t1
        type: ref
possible_keys: idx_col1_col2_col3
      key: idx_col1_col2_col3
    key_len: 13
      ref: const
     rows: 142
    Extra: Using where; Using index
1 row in set (0.00 sec)

```

知乎 @很six

前一章索引有点说过（查询中排序的字段，排序的字段若通过索引去访问将大大提高排序速度），MySQL自我分析之后的结果和前者的区别：从两条SQL中可以看出修改后后者SQL性能高于前者，后者你怎么修的路（索引），那么我就按照你修好的路走（索引），但是前者也同样查出来了，但是修的路，并没有都走，而是自己在内部产生了一次排序，同比性能，前者在内部自己折腾了一次进行了排序结果得到的结果并没有后者好。（建立索引之后，尽可能访问的时候也是按照索引的顺序）

- Using temporary : 同比前者性能更差，使用了临时表保存中间结果，MySQL在对查询结果排序时使用了临时表。常见于排序order by和分组查询group by。

tip：临时表的创建是很伤系统性能的，因为搬数据搬到临时表，用完之后再把临时表回收，数据库内部要自己折腾，这时候查询数据几百万几千万条数据，空间要申请的多，搬数据也多，最后还需要释放，严重增加数据库负担。

```
mysql> explain select col1 from t1 where col1 in ('ac','ab','aa') group by col2\G
***** 1. row *****
    id: 1
  select_type: SIMPLE
    table: t1
    type: range
possible_keys: idx_col1_col2
      key: idx_col1_col2
    key_len: 13
      ref: NULL
     rows: 569
  Extra: Using where; Using index; Using temporary; Using filesort
1 row in set (0.00 sec)
```

```
mysql> explain select col1 from t1 where col1 in ('ac', 'ab') group by col1, col2\G
***** 1. row *****
    id: 1
  select_type: SIMPLE
    table: t1
    type: range
possible_keys: idx_col1_col2_col3
      key: idx_col1_col2_col3
    key_len: 26
      ref: NULL
     rows: 4
  Extra: Using where; Using index for group-by
1 row in set (0.00 sec)
```

分析两者SQL，使用复合索引（idx\_col1\_col2,只要key不为null就是用到索引不要怀疑），这种SQL就慢的无比，如果数据是百万千万级别系统会被拖慢。避免临时表的创建。第一个sql，索引idx\_col1\_col2, group by col2, 没走索引，第二个sql用上了索引。

- Using index : 表示相应的SELECT操作中使用了覆盖索引(Covering Index), 避免了访问表的数据行，效率还可以

1. 如果同时出现Using where, 表明索引被用来执行索引键值的查找；

```
mysql> explain select col2 from t1 where col1 = 'ab';
+-----+...+-----+-----+-----+-----+
| id | ... | possible_keys | key           | key_len | ref   | rows | Extra          |
+-----+...+-----+-----+-----+-----+
| 1  | ... | idx_col1_col2 | idx_col1_col2 | 13      | const | 143  | Using where; Using index |
+-----+...+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

创建的索引中包含col1和col2，并且查询的列就包含col2，部分列跟索引重合匹配。

2. 如果没有出现同时出现Using where, 表明索引用来读取数据而非执行查找动作

同上，只是SQL中并没有根据条件去查找

- Using where : 使用了where过滤
  - Using join buffer : 使用了连接缓存
  - 索引优化MIN/MAX操作或者对于MyIsam存储引擎优化COUNT(\*)操作, 不必等到执行阶段再进行计算, 查询执行计划生成阶段即完成优化。Impossible WHERE : where子句值总是false, 不能用来获取任何数据, 如name='张三' and name='李四'(不可能一个人名字是张三, 又是李四吧)
  - SELECT tables optimized away : 在没有group by子句的情况下
  - distinct : 优化distinct操作, 在找到第一匹配的元组后即停止找同样值的动作

热身 Case

第一行 (执行顺序4) : id列为1, 表示union里的第一个select, select\_type列的primary表示查询为外层查询, table列被标记为<devied3>, 表示查询结果来自一个衍生表, 其中derived3中3代表该查询衍生自第三个select查询, 即id为3的select。 [select d1,name...]

第二行（执行顺序2）：id为3，是整个查询中第三个select的一部分，因查询包含在from中，所以为derived。【select id,name from t1 where other\_column='」】

第三行（执行顺序3）：select列表中的子查询select\_type为subquery，为整个查询中的第二个select。【select id from t3】

第四行（执行顺序1）：select\_type为union，说明第四个select是union里的第二个select，直接执行【select name,id from t2】

第五行（执行顺序5）：代表从union的临时表中读取行的阶段，table列的<union1,4>表示用第一行和第四行的select结果进行union操作。【两个结果union操作】

ps:

```
mysql> explain select * from test01 where c1 = 'a1' order by c3,c2;
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | test01 | ref | ids_test01_c1234 | ids_test01_c1234 | 768 | const |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.07 sec)
```

```

mysql> explain select * from test01 where c1 = 'a1' and c2='a2' and c5 = 'a5' order by c3,c2;
+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+
| 1 | SIMPLE | test01 | ref | ids_test01_c1234 | ids_test01_c1234 | 1536 | const,const | 1 | Using where |
+-----+
1 row in set (0.03 sec)

```

没有出现filesort

为什么这个没有出现filesort呢？明明上面的例子中出现了filesort，这是因为此时的c2已经是一个常量了。也就是说order by c3,c2，现在变成order by c3,'a2'(一个具体的常量值)。这时候这个值排不排序就已经无关紧要了，毕竟一个值排序并无意义。那么也就不需要排序了。此时返回来看，c1, c2均为常量。那么order by c3,'a2'正好接上我们创建索引时顺序。也就不会产生filesort了。

KEY a\_b\_c(a,b,c)  
order by 能使用索引最左前缀

```

ORDER BY a
ORDER BY a,b
ORDER BY a,b,c
ORDER BY a DESC,b DESC,c DESC

```

如果where使用缩印的最左前缀定义为常量，则order by 能使用索引

```

WHERE a = const ORDER BY b,c
WHERE a = const AND b = const ORDER BY c
WHERE a = const AND b > const ORDER BY b,c

```

不适用索引进行排序

```

ORDER BY a ASC,b DESC, c DESC #序不一致
WHERE g = const ORDER BY b,c #丢失a索引
WHERE a = const ORDER BY c #丢失b索引
WHERE a = const ORDER BY a,d #d不是索引的一部分
WHERE a in (....) ORDER BY b,c #对于排序来说，多个相等条件也是范围查询

```

Where语句	索引是否被使用
where a = 3	Y, 使用到a
where a = 3 and b = 5	Y, 使用到a, b
where a = 3 and b = 5 and c = 4	Y, 使用到a,b,c
where b = 3 或者 where b = 3 and c = 4 或者 where c = 4	N
where a = 3 and c = 5	使用到a, 但是c不可以, b中间断了
where a = 3 and b > 4 and c = 5	使用到a和b, c不能用在范围之后, b断了
where a = 3 and b like 'kk%' and c = 4	Y, 使用到a,b,c
where a = 3 and b like '%kk%' and c = 4	Y, 只用到a
where a = 3 and b like '%kk%' and c = 4	Y, 只用到a
where a = 3 and b like 'k%kk%' and c = 4	Y, 使用到a,b,c

知乎 @很six

## mysql经典查询

from--where--group by--having--select--distinct--union--order by--limit

# 练手第一篇

**scott.emp** 员工表结构如下：

Name	Type	Nullable	Default	Comments
EMPNO	NUMBER(4)			员工号
ENAME	VARCHAR2(10)	Y		员工姓名
JOB	VARCHAR2(9)	Y		工作
MGR	NUMBER(4)	Y		上级编号
HIREDATE	DATE	Y		雇佣日期
SAL	NUMBER(7, 2)	Y		薪金
COMM	NUMBER(7, 2)	Y		佣金
DEPTNO	NUMBER(2)	Y		部门编号

**scott.dept** 部门表

Name	Type	Nullable	Default	Comments
DEPTNO	NUMBER(2)			部门编号
DNAME	VARCHAR2(14)	Y		部门名称
LOC	VARCHAR2(13)	Y		地点

提示：工资 = 薪金 + 佣金

回答下列问题：

1. 列出至少有一个员工的所有部门。
2. 列出薪金比“SMITH”多的所有员工。
3. 列出所有员工的姓名及其直接上级的姓名。
4. 列出受雇日期早于其直接上级的所有员工。
5. 列出部门名称和这些部门的员工信息，同时列出那些没有员工的部门
6. 列出所有“CLERK”(办事员)的姓名及其部门名称。
7. 列出最低薪金大于1500的各种工作。
8. 列出在部门“SALES”(销售部)工作的员工的姓名，假定不知道销售部的部门编号。
9. 列出薪金高于公司平均薪金的所有员工。
10. 列出与“SCOTT”从事相同工作的所有员工。
11. 列出薪金等于部门编号30中员工的薪金 的 所有员工的姓名和薪金(排除部门编号30的员工)。
12. 列出薪金高于在部门编号30工作的所有员工的薪金的员工姓名和薪金。
13. 列出在每个部门工作的员工数量、平均工资和平均服务期限。
14. 列出所有员工的姓名、部门名称和工资。
15. 列出所有部门的详细信息和部门人数。
16. 列出各种工作的最低工资。
17. 列出各个部门的MANAGER(经理)的最低薪金。
18. 列出所有员工的年工资,按年薪从低到高排序。
19. 查出各部门薪金最高的员工信息

```
1. select dept.dname from dept where dept.deptno in (select emp.deptno from emp);
   select dept.dname from dept where exists (select 1 from emp where dept.deptno = emp.deptno);
   select dept.dname from dept where dept.deptno in (select emp.deptno from emp group by emp.deptno      having count(deptno)>0);
```

```

2. select emp.ename from emp where emp.sal > (select b.sal from emp b where
b.ename = 'SMITH');
    select a.ename from emp a where exists (select 1 from emp b where b.sal <
a.sal and b.ename =
                                'SMITH');
    select a.ename from emp a,(select emp.sal from emp where emp.ename='SMITH') b
where a.sal >b.sal;

3. select a.ename, b.ename boss from emp a left join emp b on a.mgr = b.empno;
    select b.ename, (select a.ename from emp a where b.mgr = a.empno) boss from
emp b;

4. select a.ename from emp a where a.hiredate < (select b.hiredate from emp b
where a.mgr = b.empno);
    select a.ename from emp a LEFT JOIN emp b on a.mgr = b.empno where a.hiredate
< b.hiredate ;

5. select a.dname,b.ename from dept a left join emp b on a.deptno = b.deptno;

6. select a.ename,b.dname from emp a left join dept b on a.deptno = b.deptno
where a.job = 'CLERK';

7. select a.job from emp a group by a.job having min(a.sal) > 1500;

8. select a.ename from emp a left join dept b on a.deptno = b.deptno where
b.dname = 'SALES';

9. select a.ename,a.job from emp a where a.sal > (select avg(b.sal) from emp b);

10.select a.ename from emp a where a.job = (select b.job from emp b where
b.ename = 'SCOTT');

11.select a.ename,a.sal from emp a where a.sal in (select b.sal from emp b where
b.deptno = 30)
    and a.deptno <> 30;
    select a.ename,a.sal from emp a where a.deptno <> 30 and exists (select 1
from emp b where a.sal
                                = b.sal and
b.deptno = 30);
12.select a.ename,a.sal from emp a where a.sal > (select max(b.sal) from emp b
where b.deptno = 30);

13.select b.dname,p.* from (select a.deptno,count(1) num,avg(a.hiredate)
avgdate,avg(a.sal + IFNULL(a.comm, 0))from emp a group by a.deptno) p left join
dept b on p.deptno = b.deptno;

14.select a.ename,b.dname,(a.sal + IFNULL(a.comm, 0)) sa from emp a left join
dept b on a.deptno = b.deptno;

15.select b.* ,p.num from (select a.deptno,count(1) num from emp a group by
a.deptno) p left join dept b on p.deptno = b.deptno;

16.select a.job,min(a.sal + ifnull(a.comm,0)) from emp a group by a.job;
//这是一种思路，虽然不满足工资的计算方式
    select a.job,a.sal from emp a where not exists (select 1 from emp b where
a.sal > b.sal );

```

```

17.select a.deptno,min(a.sal) from emp a where a.job = 'MANAGER' group by a.deptno;

18.select a.ename,(a.sal + ifnull(a.comm,0)) * 12 sumsal from emp a order by sumsal asc;

19.select b.deptno,b.ename,b.sal from emp b where (b.deptno,b.sal) in (select a.deptno,max(a.sal)from emp a group by a.deptno);
SELECT a.deptno,a.ename,a.sal FROM emp a WHERE (SELECT count(1) FROM emp b WHERE a.sal < b.sal) = 0;

```

ps:

对于16题，19题这类，查找最高，最低，前几的查询。都可以和自己联表解决，最好把自己作为左边的表，主表，如下面的a。比如第16题，可以这么解释

```

1.select a.job,a.sal from emp a where not exists (select 1 from emp b where a.sal > b.sal );
2.SELECT a.job,a.sal FROM emp a WHERE (SELECT count(1) FROM emp b WHERE a.sal > b.sal) = 0;

```

1.在a表中，不存在a的sal > b的sal，所以就是查出a中最低的薪金。这里和自己联表，相当于挨个拿a中的sal，去和b中的sal比较，找到某个a的sal，这个sal比所有b中的sal都小（也就是说，这个sal不可能（not exists）大于b中的任何sal（a.sal > b.sal），包括和自己相等的那个。或者说，遍历a的sal去和b里的所有sal比较时，如果发生了a的sal>b.sal，那肯定说明当前这个a的sal不是最小的）。所以要找出最大薪金的方式，就是改成a.sal < b.sal

2.比1要好理解一点，也是遍历a的sal，并且统计a的sal比b的sal大的个数，要查出最小的薪金，肯定a的sal比b的sal大的个数为0啊。那要是查最大的薪金呢？肯定是a的sal比b的sal小的个数为0，才说明a的这个sal是最大的，也就是变成a.sal<b.sal。

## 衍生

- 上面的子查询，都没有限制条件，都是遍历a的sal去挨个和b的sal比较。如果查询条件变成各部门的最低薪资呢？只要在子查询里加上限制条件就行，比如1里可以加上a.deptno = b.deptno，这样就变成遍历a的sal去和同部门的人比较了（意思遍历a的当前sal去和b里与a同部门的人的sal比较）
- 查询各个部门工资前3名的员工信息。可以参考上面的2。其实就是限制条件为同部门，然后a里的sal比b里的sal小的个数不超过2个（也就是当前a的sal在前3了）

```

SELECT a.deptno,a.ename,a.sal FROM emp a WHERE (SELECT count(1) FROM emp b
WHERE a.deptno = b.deptno and a.sal < b.sal) <= 2;

```

- 查询各个部门工资最后3名的员工信息。怎么办？其实就是a里的sal比b里的sal大的个数不超过2个（超过2个了，比如说是3个，说明比b里3个都大，那肯定不是工资最低的3个里的了）

```

SELECT a.deptno,a.ename,a.sal FROM emp a WHERE (SELECT count(1) FROM emp b
WHERE a.deptno = b.deptno and a.sal > b.sal) <= 2;

```

## 练手第二篇

- 找出EMP表中的姓名（ENAME）第三个字母是A的员工姓名。

2. 找出EMP表员工名字中含有A 和N的员工姓名。
3. 找出所有有佣金的员工，列出姓名、工资、佣金，显示结果按工资从小到大，佣金从大到小。
4. 列出部门编号为20的所有职位。
5. 列出不属于 SALES 的部门。
6. 显示工资不在1000 到1500 之间的员工信息：名字、工资，按工资从大到小排序。
7. 显示职位为 MANAGER 和 SALESMAN ，年薪在15000 和20000 之间的员工的信息：名字、职位、年薪。
8. 说明以下两条SQL语句的输出结果：  
SELECT EMPNO,COMM FROM EMP WHERE COMM IS NULL;  
SELECT EMPNO,COMM FROM EMP WHERE COMM = NULL ;
9. 让SELECT 语句的输出结果为  
SELECT \* FROM SALGRADE;  
SELECT \* FROM BONUS;  
SELECT \* FROM EMP;  
SELECT \* FROM DEPT;  
.....  
列出当前用户有多少张数据表，结果集中存在多少条记录。
10. 判断 SELECT ENAME,SAL FROM EMP WHERE SAL > '1500' 是否抱错，为什么？

```
1.select a.ename from emp a where a.ename like '__A%';  
2.select a.ename from emp a where a.ename like '%A%' and a.ename like '%N%';
```

## mysql常用函数

## Oracle

## oracle基本概念

## Elasticsearch

## Mybatis

## Hibernate

## Hbase

## kafka

## Zookeeper

## Netty

