

# JAVA核心知识点

刚刚经历过秋招，看了大量的面经，顺便将常见的Java常考知识点总结了一下，并根据被问到的频率大致做了一个标注。题目后面的星星数量越多，面试被问到的频率越高，建议深入理解高频知识点并熟练掌握高频知识点的相关知识，方便面试时拓展（方便装逼），给面试官留下个好印象。

关注微信公众号“路人zhang”，获取更多面试技巧及资料，本文档持续更新，最新版可在公众号获取。



## JAVA核心知识点

- JVM、JRE及JDK的关系 \* \*
- JAVA语言特点 \* \*
- JAVA和C++的区别 \* \*
- Java的基本数据类型 \* \*
- 隐式(自动)类型转换和显示(强制)类型转换 \* \*
- 自动装箱与拆箱 \* \*
- String(不是基本数据类型)
  - String的不可变性 \* \* \*
  - 字符串常量和字符串常量的区别 \*
  - 什么是字符串常量池？ \*
  - String 类的常用方法都有那些？ \* \*
  - String和StringBuffer、StringBuilder的区别是什么？ \* \* \*
- switch 是否能作用在 byte 上，是否能作用在 long 上，是否能作用在 String 上 \*
- Java语言采用何种编码方案？有何特点？ \*
- 访问修饰符 \* \*
- 运算符 \*
- 关键字
  - static关键字 \* \* \*
  - final 关键字 \* \* \*
  - final finally finalize区别 \* \* \*
  - this关键字 \* \*
  - super关键字 \* \*
  - this与super的区别 \* \*
  - break ,continue ,return 的区别及作用 \* \*
- 面向对象和面向过程的区别 \* \*
- 面向对象三大特性(封装、继承、多态) \* \* \*
- 面向对象五大基本原则是什么 \* \*
- 抽象类和接口的对比 \* \* \*
- 在Java中定义一个不做事且没有参数的构造方法的作用 \*
- 在调用子类构造方法之前会先调用父类没有参数的构造方法，其目的是 \*

一个类的构造方法的作用是什么？若一个类没有声明构造方法，改程序能正确执行吗？为什么？ \*  
构造方法有哪些特性？ \*\*\*

变量 \*\*\*

内部类 \*\*\*

重写与重载 \*\*\*

重载和重写的区别

构造器 (constructor) 是否可被重写 (override)

重载的方法能否根据返回类型进行区分？为什么？

== 和 equals 的区别 \*\*\*

hashCode 与 equals (为什么重写equals方法后， hashCode方法也必须重写) \*\*\*

Java 中是值传递还是引用传递，还是两者共存 \*\*\*

IO流 \*

BIO,NIO,AIO 有什么区别？ \*\*

反射 \*\*\*

JAVA异常 \*\*\*

JAVA注解 \*\*

JAVA泛型 \*\*\*

JAVA序列化 \*\*

深拷贝与浅拷贝 \*\*\*

常见的Object方法 \*\*\*

## Java集合

常用的集合类有哪些？ \*\*\*

List, Set, Map三者的区别？ \*\*\*

常用集合框架底层数据结构 \*\*\*

哪些集合类是线程安全的？ \*\*\*

迭代器 Iterator 是什么 \*

Java集合的快速失败机制 “fail-fast” 和安全失败机制“fail-safe”是什么？ \*\*\*

如何遍历边移除 Collection 中的元素？ \*\*\*

Array 和 ArrayList 有何区别？ \*\*\*

comparable 和 comparator的区别？ \*\*

Collection 和 Collections 有什么区别？ \*\*

### List集合

遍历一个 List 有哪些不同的方式？ \*\*

ArrayList的扩容机制 \*\*\*

ArrayList 和 LinkedList 的区别是什么？ \*\*\*

ArrayList 和 Vector 的区别是什么？ \*\*\*

简述 ArrayList、Vector、LinkedList 的存储性能和特性？ \*\*\*

### Set集合

说一下 HashSet 的实现原理 \*\*\*

HashSet如何检查重复？ (HashSet是如何保证数据不可重复的？) \*\*\*

HashSet与HashMap的区别 \*\*\*

### Map集合

HashMap在JDK1.7和JDK1.8中有哪些不同？HashMap的底层实现 \*\*\*

HashMap 的长度为什么是2的幂次方 \*\*\*

HashMap的put方法的具体流程？ \*\*

HashMap的扩容操作是怎么实现的？ \*\*\*

HashMap默认加载因子为什么选择0.75？

为什么要将链表中转红黑树的阈值设为8？为什么不一开始直接使用红黑树？

HashMap是怎么解决哈希冲突的？ \*\*\*

HashMap为什么不直接使用hashCode()处理后的哈希值直接作为table的下标？ \*\*\*

能否使用任何类作为 Map 的 key？ \*\*\*

为什么HashMap中String、Integer这样的包装类适合作为Key？ \*\*\*

如果使用Object作为HashMap的Key，应该怎么办呢？ \*\*

HashMap 多线程导致死循环问题 \*\*

ConcurrentHashMap 底层具体实现知道吗？ \*\*

HashTable的底层实现知道吗？ \*\*

HashMap、ConcurrentHashMap及Hashtable 的区别 \*\*\*

Java集合的常用方法 \*\*

Collection常用方法

List特有方法

LinkedList特有方法

Map

Stack

Queue

## 计算机网络

什么是网络协议，为什么要对网络协议分层 \*

计算机网络的各层协议及作用 \*\*\*

URI和URL的区别 \*

DNS的工作流程 \*\*\*

了解ARP协议吗? \*\*

有了IP地址，为什么还要用MAC地址? \*\*

说一下ping的过程 \*\*

路由器和交换机的区别? \*

TCP与UDP有什么区别 \*\*\*

TCP协议如何保证可靠传输 \*\*\*

TCP的三次握手及四次挥手 \*\*\*

三次握手

四次挥手

为什么TCP连接的时候是3次？两次是否可以？

为什么TCP连接的时候是3次，关闭的时候却是4次？

TIME\_WAIT和CLOSE\_WAIT的区别在哪？

为什么客户端发出第四次挥手的确认报文后要等2MSL的时间才能释放TCP连接？

如果已经建立了连接，但是客户端突然出现故障了怎么办？

HTTP 与 HTTPS 的区别 \*\*\*

什么是对称加密与非对称加密 \*\*

HTTPS的加密过程 \*\*\*

常用HTTP状态码 \*\*\*

常见的HTTP方法 \*\*\*

GET和POST区别 \*\*\*

HTTP 1.0、HTTP 1.1及HTTP 2.0的主要区别是什么 \*\*

Session、Cookie和Token的主要区别 \*\*\*

如果客户端禁止 cookie 能实现 session 还能用吗? \*

在浏览器中输入url地址到显示主页的过程 \*\*\*

Servlet是线程安全的吗 \*

## MySQL数据库

什么是MySQL? \*

MySQL常用的存储引擎有什么？它们有什么区别? \*\*\*

数据库的三大范式 \*\*

MySQL的数据类型有哪些 \*\*

索引 \*\*\*

什么是索引？

索引的优缺点？

索引的数据结构？

Hash索引和B+树的区别？

索引的类型有哪些？

索引的种类有哪些？

B树和B+树的区别？

数据库为什么使用B+树而不是B树？

什么是聚簇索引，什么是非聚簇索引？

非聚簇索引一定会进行回表查询吗？

索引的使用场景有哪些？

索引的设计原则？

如何对索引进行优化？

如何创建/删除索引？

使用索引查询时性能一定会提升吗？

什么是前缀索引？

什么是最左匹配原则?

索引在什么情况下会失效?

数据库的事务 \* \* \*

什么是数据库的事务?

事务的四大特性是什么?

数据库的并发一致性问题

数据库的隔离级别有哪些?

隔离级别是如何实现的?

什么是MVCC?

数据库的锁 \* \* \*

什么是数据库的锁?

数据库的锁与隔离级别的关系?

数据库锁的类型有哪些?

MySQL中InnoDB引擎的行锁模式及其是如何实现的?

什么是数据库的乐观锁和悲观锁, 如何实现?

什么是死锁? 如何避免?

SQL语句基础知识

SQL语句主要分为哪几类 \*

SQL约束有哪些? \*\*

什么是子查询? \*\*\*

了解MySQL的几种连接查询吗? \*\*\*

mysql中in和exists的区别? \*\*

varchar和char的区别? \*\*\*

MySQL中int(10)和char(10)和varchar(10)的区别? \*\*\*

drop、 delete和truncate的区别? \*\*

UNION和UNION ALL的区别? \*\*

什么是临时表, 什么时候会使用到临时表, 什么时候删除临时表? \*

大表数据查询如何进行优化? \*\*\*

了解慢日志查询吗? 统计过慢查询吗? 对慢查询如何优化? \*\*\*

为什么要设置主键? \*\*

主键一般用自增ID还是UUID? \*\*

字段为什么要设置成not null? \*\*

如何优化查询过程中的数据访问? \*\*\*

如何优化长难的查询语句? \*\*

如何优化LIMIT分页? \*\*

如何优化UNION查询 \*\*

如何优化WHERE子句 \*\*\*

SQL语句执行的很慢原因是什么? \*\*\*

SQL语句的执行顺序? \*

数据库优化

大表如何优化? \*\*\*

什么是垂直分表、 垂直分库、 水平分表、 水平分库? \*\*\*

分库分表后, ID键如何处理? \*\*\*

MySQL的复制原理及流程? 如何实现主从复制? \*\*\*

了解读写分离吗? \*\*\*

## JVM、 JRE及JDK的关系 \*\*

JDK (Java Development Kit) 是针对Java开发的产品, 是整个Java的核心, 包括了Java运行环境JRE、 Java工具和Java基础类库。

Java Runtime Environment (JRE) 是运行JAVA程序所必须的环境的集合, 包含JVM标准实现及Java核心类库。

JVM是Java Virtual Machine (Java虚拟机) 的缩写, 是整个java实现跨平台的最核心的部分, 能够运行以Java语言写作的软件程序。

简单来说就是JDK是Java的开发工具，JRE是Java程序运行所需的环境，JVM是Java虚拟机。它们之间的关系是JDK包含JRE和JVM，JRE包含JVM。

## JAVA语言特点 \* \*

- Java是一种面向对象的语言
- Java通过Java虚拟机实现了平台无关性，一次编译，到处运行
- 支持多线程
- 支持网络编程
- 具有较高的安全性和可靠性

## JAVA和C++的区别 \* \*

面试时记住前四个就行了

- Java 通过虚拟机从而实现跨平台特性，但是 C++ 依赖于特定的平台。
- Java 没有指针，它的引用可以理解为安全指针，而 C++ 具有和 C 一样的指针。
- Java 支持自动垃圾回收，而 C++ 需要手动回收。
- Java 不支持多重继承，只能通过实现多个接口来达到相同目的，而 C++ 支持多重继承。
- Java 不支持操作符重载，虽然可以对两个 String 对象执行加法运算，但是这是语言内置支持的操作，不属于操作符重载，而 C++ 可以。
- Java 的 goto 是保留字，但是不可用，C++ 可以使用 goto。

## Java的基本数据类型 \* \*

注意 string 不是基本数据类型

类型	关键字	包装器类型	占用内存(字节) (重要)	取值范围	默认值
字节型	byte	Byte	1	-128(-2^7) ~ 127(2^7-1)	0
短整型	short	Short	2	-2^15 ~ 2^15-1	0
整型	int	Integer	4	-2^31 ~ 2^31-1	0
长整型	long	Long	8	-2^63 ~ 2^63-1	0L
单精度浮点型	float	Float	4	3.4e-45 ~ 1.4e38	0.0F
双精度浮点型	double	Double	8	4.9e-324 ~ 1.8e308	0.0D
字符型	char	Character	2		'\u0000'
布尔型	boolean	Boolean	1	true/false	false

## 隐式(自动)类型转换和显示(强制)类型转换 \* \*

- 隐式(自动)类型转换：从存储范围小的类型到存储范围大的类型。  
`byte → short(char) → int → long → float → double`
- 显示(强制)类型转换：从存储范围大的类型到存储范围小的类型。  
`double → float → long → int → short(char) → byte`。该类类型转换很可能存在精度的损失。

看一个经典的代码

```
short s = 1;  
s = s + 1;
```

这是会报错的，因为1是int型，`s+1`会自动转换为int型，将int型直接赋值给short型会报错。

做一下修改即可避免报错

```
short s = 1;  
s = (short)(s + 1);
```

或这样写，因为`s += 1`会自动进行强制类型转换

```
short s = 1;  
s += 1;
```

## 自动装箱与拆箱 \* \*

- 装箱：将基本类型用包装器类型包装起来
- 拆箱：将包装器类型转换为基本类型

这个地方有很多易混淆的地方，但在面试中问到的频率一般，笔试的选择题中经常出现，还有一个String创建对象和这个比较像，很容易混淆，在下文可以看到

- 下面这段代码的输出结果是什么？

```
public class Main {  
    public static void main(String[] args) {  
  
        Integer a = 100;  
        Integer b = 100;  
        Integer c = 128;  
        Integer d = 128;  
  
        System.out.println(a==b);  
        System.out.println(c==d);  
    }  
}
```

```
true  
false
```

很多人看到这个结果会很疑惑，为什么会是一个true一个false。其实从源码中可以很容易找到原因。首先找到Integer方法中的valueOf方法

```

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

```

可以看到当不满足 if 语句中的条件，就会重新创建一个对象返回，那结果必然不相等。继续打开 IntegerCache 可以看到

```

private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static final Integer cache[];

    static {
        // high value may be configured by property
        int h = 127;
        String integerCacheHighPropValue =
            sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
        if (integerCacheHighPropValue != null) {
            try {
                int i = parseInt(integerCacheHighPropValue);
                i = Math.max(i, 127);
                // Maximum array size is Integer.MAX_VALUE
                h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
            } catch( NumberFormatException nfe) {
                // If the property cannot be parsed into an int,
                ignore it.
            }
        }
        high = h;

        cache = new Integer[(high - low) + 1];
        int j = low;
        for(int k = 0; k < cache.length; k++)
            cache[k] = new Integer(j++);

        // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
    }

    private IntegerCache() {}
}

```

代码挺长，大概说的就是在通过 valueof 方法创建 Integer 对象的时候，如果数值在[-128,127]之间，便返回指向 IntegerCache.cache 中已经存在的对象的引用；否则创建一个新的 Integer 对象。所以上面代码中 a 与 b 相等， c 与 d 不相等。

- 在看下面的代码会输出什么

```

public class Main {
    public static void main(String[] args) {

        Double a = 1.0;
        Double b = 1.0;

```

```
        Double c = 2.0;
        Double d = 2.0;

        System.out.println(a==b);
        System.out.println(c==d);

    }
}
```

```
false
false
```

采用同样的方法，可以看到 `Double` 的 `valueOf` 方法，每次返回都是重新 `new` 一个新的对象，所以上面代码中的结果都不想等。

```
public static Double valueOf(double d) {
    return new Double(d);
}
```

- 最后再看这段代码的输出结果

```
public class Main {
    public static void main(String[] args) {

        Boolean a = false;
        Boolean b = false;
        Boolean c = true;
        Boolean d = true;

        System.out.println(a==b);
        System.out.println(c==d);
    }
}
```

```
true
true
```

老方法继续看 `valueOf` 方法

```
public static Boolean valueOf(boolean b) {
    return (b ? TRUE : FALSE);
}
```

再看看 `TRUE` 和 `FALSE` 是个什么东西，是两个静态成员属性。

```
public static final Boolean TRUE = new Boolean(true);
public static final Boolean FALSE = new Boolean(false);
```

**说下结论：** `Integer`、`Short`、`Byte`、`Character`、`Long` 这几个类的 `valueOf` 方法的实现是类似的。`Double`、`Float` 的 `valueOf` 方法的实现是类似的。然后是 `Boolean` 的 `valueOf` 方法是单独一组的。

- `Integer i = new Integer(xxx)` 和 `Integer i =xxx` 的区别

这两者的区别主要是第一种会触发自动装箱，第二者不会

最后看看下面这段程序的输出结果

```
public class Main {
    public static void main(String[] args) {
        Integer a = 1;
        Integer b = 2;
        Integer c = 3;
        Long g = 3L;
        int int1 = 12;
        int int2 = 12;
        Integer integer1 = new Integer(12);
        Integer integer2 = new Integer(12);
        Integer integer3 = new Integer(1);

        System.out.println("c==(a+b) ->" + (c==(a+b)));
        System.out.println("g==(a+b) ->" + (g==(a+b)));
        System.out.println("c.equals(a+b) ->" + (c.equals(a+b)));
        System.out.println("g.equals(a+b) ->" + (g.equals(a+b)));
        System.out.println("int1 == int2 -> " + (int1 == int2));
        System.out.println("int1 == integer1 -> " + (int1 == integer1));
        System.out.println("integer1 == integer2 -> " + (integer1 == integer2));
        System.out.println("integer3 == a1 -> " + (integer3 == a));
    }
}
```

```
c==(a+b) ->true
g==(a+b) ->true
c.equals(a+b) ->true
g.equals(a+b) ->false
int1 == int2 -> true
int1 == integer1 -> true
integer1 == integer2 -> false
integer3 == a1 -> false
```

下面简单解释这些结果。

1.当`"=="`运算符的两个操作数都是包装器类型的引用，则是比较指向的是否是同一个对象，而如果其中有一个操作数是表达式（即包含算术运算）则比较的是数值（即会触发自动拆箱的过程）。所以`c==a+b`, `g==a+b`为`true`。

2.而对于`equals`方法会先触发自动拆箱过程，再触发自动装箱过程。也就是说`a+b`, 会先各自调用`intValue`方法，得到了加法运算后的数值之后，便调用`Integer.valueOf`方法，再进行`equals`比较。所以`c.equals(a+b)`为`true`。而对于`g.equals(a+b)`, `a+b`会先拆箱进行相加运算，在装箱进行`equals`比较，但是装箱后为`Integer`, `g`为`Long`, 所以`g.equals(a+b)`为`false`。

3.`int1 == int2`为`true`无需解释，`int1 == integer1`, 在进行比较时，`integer1`会先进行一个拆箱操作变成`int`型在进行比较，所以`int1 == integer1`为`true`。

4. `integer1 == integer2` -> `false`。`integer1` 和 `integer2` 都是通过 `new` 关键字创建的，可以看成两个对象，所以 `integer1 == integer2` 为 `false`。`integer3 == a1` -> `false`，`integer3` 是一个对象类型，而 `a1` 是一个常量它们存放内存的位置不一样，所以 `integer3 == a1` 为 `false`，具体原因可学习下 Java 的内存模型。

## String(不是基本数据类型)

### String的不可变性 \* \* \*

在 Java 8 中，`String` 内部使用 `char` 数组存储数据。并且被声明为 `final`，因此它不可被继承。

```
public final class String implements java.io.Serializable, Comparable<String>,  
CharSequence {  
    private final char value[];  
}
```

为什么 `String`` 要设计成不可变的呢（不可变性的好处）：

1. 可以缓存 hash 值 ()

因为 `String` 的 `hash` 值经常被使用，例如 `String` 用做 `HashMap` 的 `key`。不可变的特性可以使 `hash` 值也不可变，因此只需要进行一次计算。

2. 常量池优化

`String` 对象创建之后，会在字符串常量池中进行缓存，如果下次创建同样的对象时，会直接返回缓存的引用。

3. 线程安全

`String` 不可变性天生具备线程安全，可以在多个线程中安全地使用。

### 字符型常量和字符串常量的区别 \*

- 形式上：字符常量是单引号引起的一个字符 字符串常量是双引号引起的若干个字符
- 含义上：字符常量相当于一个整形值(ASCII值)，可以参加表达式运算 字符串常量代表一个地址值(该字符串在内存中存放位置)
- 占内存大小：字符常量占两个字节 字符串常量占若干个字节(至少一个字符结束标志)

### 什么是字符串常量池？ \*

字符串常量池位于堆内存中，专门用来存储字符串常量，可以提高内存的使用率，避免开辟多块空间存储相同的字符串，在创建字符串时 JVM 会首先检查字符串常量池，如果该字符串已经存在池中，则返回它的引用，如果不存在，则实例化一个字符串放到池中，并返回其引用。

### String 类的常用方法都有那些？ \* \*

面试时一般不会问，但面试或笔试写字符串相关的算法题经常会涉及到，还是得背一背（以下大致是按使用频率优先级排序）

- `length()`：返回字符串长度
- `charAt()`：返回指定索引处的字符
- `substring()`：截取字符串
- `trim()`：去除字符串两端空白
- `split()`：分割字符串，返回一个分割后的字符串数组。
- `replace()`：字符串替换。

- `indexOf()`: 返回指定字符的索引。
- `toLowerCase()`: 将字符串转成小写字母。
- `toUpperCase()`: 将字符串转成大写字母。

## String和StringBuffer、StringBuilder的区别是什么? \* \* \*

### 1.可变性

`String` 不可变, `StringBuilder` 和 `StringBuffer` 是可变的

### 2.线程安全性

`String` 由于是不可变的, 所以线程安全。`StringBuffer` 对方法加了同步锁或者对调用的方法加了同步锁, 所以是线程安全的。`StringBuilder` 并没有对方法进行加同步锁, 所以是非线程安全的。

### 3.性能

`StringBuilder` > `StringBuffer` > `String`

为了方便记忆, 总结如下

	是否可变	是否安全	性能
<code>String</code>	不可变	安全	低
<code>StringBuilder</code>	可变	不安全	高
<code>StringBuffer</code>	可变	安全	较高

## switch 是否能作用在 byte 上, 是否能作用在 long 上, 是否能作用在 String 上 \* \*

`switch` 可以作用于 `char` `byte` `short` `int` 及它们对应的包装类型, `switch` 不可作用于 `long` `double` `float` `boolean` 及他们的包装类型。在JDK1.5之后可以作用于枚举类型, 在JDK1.7之后可作用于 `String` 类型。

## Java语言采用何种编码方案? 有何特点? \*

Java语言采用Unicode编码标准, 它为每个字符制订了一个唯一的数值, 因此在任何的语言, 平台, 程序都可以放心的使用。

## 访问修饰符 \* \*

在Java编程语言中有四种权限访问控制符, 这四种访问权限的控制符能够控制类中成员的可见性。其中类有两种 `public`、`default`。而方法和变量有4种: `public`、`default`、`protected`、`private`。

- `public`: 对所有类可见。使用对象: 类、接口、变量、方法
- `protected`: 对同一包内的类和所有子类可见。使用对象: 变量、方法。注意: 不能修饰类(外部类)。
- `default`: 在同一包内可见, 不使用任何修饰符。使用对象: 类、接口、变量、方法。
- `private`: 在同一类内可见。使用对象: 变量、方法。注意: 不能修饰类(外部类)

修饰符	当前类	同包内	子类(同包)	其他包
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	Y	N
private	Y	N	N	N

## 运算符 \*

- &&和&

&& 和 & 都可以表示逻辑与，但他们是有区别的，共同点是他们两边的条件都成立的时候最终结果才是 true；不同点是 && 只要是第一个条件不成立为 false，就不会再去判断第二个条件，最终结果直接为 false，而 & 判断的是所有的条件。

- ||和|

|| 和 | 都表示逻辑或，共同点是只要两个判断条件其中有一个成立最终的结果就是 true，区别是 || 只要满足第一个条件，后面的条件就不再判断，而 | 要对所有的条件进行判断。

## 关键字

### static关键字 \* \* \*

static 关键字的主要用途就是方便在没有创建对象时调用方法和变量和优化程序性能

#### 1.static变量 (静态变量)

用 static 修饰的变量被称为静态变量，也被称为类变量，可以直接通过类名来访问它。静态变量被所有的对象共享，在内存中只有一个副本，仅当在类初次加载时会被初始化，而非静态变量在创建对象的时候被初始化，并且存在多个副本，各个对象拥有的副本互不影响。

#### 2.static方法(静态方法)

static 方法不依赖于任何对象就可以进行访问，在 static 方法中不能访问类的非静态成员变量和非静态成员方法，因为非静态成员方法/变量都是必须依赖具体的对象才能够被调用，但是在非静态成员方法中是可以访问静态成员方法/变量的。

```
public class Main {
    public static String s1 = "s1"; //静态变量
    String s2 = "s2";
    public void fun1(){
        System.out.println(s1);
        System.out.println(s2);
    }

    public static void fun2(){
        System.out.println(s1);
        System.out.println(s2); //此处报错，静态方法不能调用非静态变量
    }
}
```

#### 3.static代码块 (静态代码块)

静态代码块的主要用途是可以用来优化程序的性能，因为它只会在类加载时加载一次，很多时候会将一些只需要进行一次的初始化操作都放在 static 代码块中进行。如果程序中有多个 static 块，在类初次被加载的时候，会按照 static 块的顺序来执行每个 static 块。

```
public class Main {  
    static {  
        System.out.println("hello,word");  
    }  
    public static void main(String[] args) {  
        Main m = new Main();  
    }  
}
```

#### 4.可以通过this访问静态成员变量吗？（可以）

this 代表当前对象，可以访问静态变量，而静态方法中是不能访问非静态变量，也不能使用 this 引用。

#### 5.初始化顺序

静态变量和静态语句块优先于实例变量和普通语句块，静态变量和静态语句块的初始化顺序取决于它们在代码中的顺序。如果存在继承关系的话，初始化顺序为父类中的静态变量和静态代码块——子类中的静态变量和静态代码块——父类中的实例变量和普通代码块——父类的构造函数——子类的实例变量和普通代码块——子类的构造函数

#### final 关键字 \* \* \*

final 关键字主要用于修饰类，变量，方法。

1. 类：被 final 修饰的类不可以被继承
2. 方法：被 final 修饰的方法不可以被重写
3. 变量：被 final 修饰的变量是基本类型，变量的数值不能改变；被修饰的变量是引用类型，变量便不能在引用其他对象，但是变量所引用的对象本身是可以改变的。

```
public class Main {  
    int a = 1;  
    public static void main(String[] args) {  
        final int b = 1;  
        b = 2; //报错  
        final Main m = new Main();  
        m.a = 2; //不报错，可以改变引用类型变量所指向的对象  
    }  
}
```

#### final finally finalize区别 \* \* \*

- final 主要用于修饰类，变量，方法
- finally 一般作用在 try-catch 代码块中，在处理异常的时候，通常我们将一定要执行的代码放在 finally 代码块中，表示不管是否出现异常，该代码块都会执行，一般用来存放一些关闭资源的代码。
- finalize 是一个属于 Object 类的一个方法，该方法一般由垃圾回收器来调用，当我们调用 System.gc() 方法的时候，由垃圾回收器调用 finalize()，回收垃圾，但 Java 语言规范并不保证 finalize 方法会被及时地执行、而且根本不会保证它们会被执行。

## this关键字 \* \*

重点掌握前三种即可

1. `this` 关键字可用来引用当前类的实例变量。主要用于形参与成员名字重名，用 `this` 来区分。

```
public Person(String name, int age) {  
    this.name = name;  
    this.age = age;  
}
```

2. `this` 关键字可用于调用当前类方法。

```
public class Main {  
    public void fun1(){  
        System.out.println("hello,word");  
    }  
    public void fun2(){  
        this.fun1(); //this可省略  
    }  
  
    public static void main(String[] args) {  
        Main m = new Main();  
        m.fun2();  
    }  
}
```

3. `this()` 可以用来调用当前类的构造函数。(注意：`this()` 一定要放在构造函数的第一行，否则编译不通过)

```
class Person{  
    private String name;  
    private int age;  
  
    public Person() {}  
  
    public Person(String name) {  
        this.name = name;  
    }  
    public Person(String name, int age) {  
        this(name);  
        this.age = age;  
    }  
}
```

4. `this` 关键字可作为调用方法中的参数传递。

5. `this` 关键字可作为参数在构造函数调用中传递。

6. `this` 关键字可用于从方法返回当前类的实例。super

## super关键字 \* \*

1. `super` 可以用来引用直接父类的实例变量。和 `this` 类似，主要用于区分父类和子类中相同的字段
2. `super` 可以用来调用直接父类构造函数。(注意：`super()` 一定要放在构造函数的第一行，否则编译不通过)
3. `super` 可以用来调用直接父类方法。

```
public class Main {  
    public static void main(String[] args) {  
        Child child = new Child("Father", "Child");  
        child.test();  
    }  
  
    class Father{  
        protected String name;  
  
        public Father(String name) {  
            this.name = name;  
        }  
  
        public void say(){  
            System.out.println("Hello, child");  
        }  
  
    }  
  
    class Child extends Father{  
        private String name;  
  
        public Child(String name1, String name2) {  
            super(name1); // 调用直接父类构造函数  
            this.name = name2;  
        }  
  
        public void test(){  
            System.out.println(this.name);  
            System.out.println(super.name); // 引用直接父类的实例变量  
            super.say(); // 调用直接父类方法  
        }  
    }  
}
```

## this与super的区别 \* \*

- 相同点：
  1. `super()` 和 `this()` 都必须在构造函数的第一行进行调用，否则就是错误的
  2. `this()` 和 `super()` 都指的是对象，所以，均不可以在 `static` 环境中使用。
- 不同点：
  1. `super()` 主要是对父类构造函数的调用，`this()` 是对重载构造函数的调用
  2. `super()` 主要在继承了父类的子类的构造函数中使用，是在不同类中的使用；`this()` 主要在同一类的不同构造函数中的使用

## break ,continue ,return 的区别及作用 \* \*

- break 结束当前的循环体
- continue 结束本次循环,进入下一次循环
- return 结束当前方法

## 面向对象和面向过程的区别 \* \*

- 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。

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

## 面向对象三大特性(封装、继承、多态) \* \* \*

- 封装

封装就是隐藏对象的属性和实现细节，仅对外公开接口，控制在程序中属性的读和修改的访问级别。

- 继承

继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。

- 多态（重要）

多态是同一个行为具有多个不同表现形式或形态的能力。这句话不是很好理解，可以看这个解释，在Java语言中，多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

在Java中实现多态的三个必要条件：继承、重写、向上转型。继承和重写很好理解，向上转型是指在多态中需要将子类的引用赋给父类对象。

```
public class Main {  
    public static void main(String[] args) {  
        Person person = new Student(); //向上转型  
        person.run();  
    }  
}  
  
class Person {  
    public void run() {  
        System.out.println("Person");  
    }  
}  
  
class Student extends Person { //继承  
    @Override  
    public void run() { //重载  
        System.out.println("Student");  
    }  
}
```

运行结果为

Student

## 面向对象五大基本原则是什么 \* \*

- **单一职责原则 (Single-Responsibility Principle)**

一个类，最好只做一件事，只有一个引起它的变化。单一职责原则可以看做是低耦合、高内聚在面向对象原则上的引申，将职责定义为引起变化的原因，以提高内聚性来减少引起变化的原因。

- **开放封闭原则 (Open-Closed principle)**

软件实体应该是可扩展的，而不可修改的。也就是，对扩展开放，对修改封闭的。

- **里氏替换原则 (Liskov-Substitution Principle)**

**子类必须能够替换其基类。**这一思想体现为对继承机制的约束规范，只有子类能够替换基类时，才能保证系统在运行期内识别子类，这是保证继承复用的基础。在父类和子类的具体行为中，必须严格把握继承层次中的关系和特征，将基类替换为子类，程序的行为不会发生任何变化。同时，这一约束反过来则是不成立的，子类可以替换基类，但是基类不一定能替换子类。

- **依赖倒置原则 (Dependency-Inversion Principle)**

**依赖于抽象。**具体而言就是高层模块不依赖于底层模块，二者都同依赖于抽象；抽象不依赖于具体，具体依赖于抽象。

- **接口隔离原则 (Interface-Segregation Principle)**

使用多个小的专门的接口，而不要使用一个大的总接口。

## 抽象类和接口的对比 \* \* \*

在Java语言中，`abstract class` 和 `interface` 是支持抽象类定义的两种机制。抽象类：用来捕捉子类的通用特性的。接口：抽象方法的集合。

**相同点：**

- 接口和抽象类都不能实例化
- 都包含抽象方法，其子类都必须覆写这些抽象方法

**不同点：**

类型	抽象类	接口
定义	abstract class	Interface
实现	extends(需要提供抽象类中所有声明的方法的实现)	implements (需要提供接口中所有声明的方法的实现)
继承	抽象类可以继承一个类和实现多个接口；子类只可以继承一个抽象类	接口只可以继承接口（一个或多个）；子类可以实现多个接口
访问修饰符	抽象方法可以有public、protected和default这些修饰符	接口方法默认修饰符是public。你不可以使用其它修饰符
构造器	抽象类可以有构造器	接口不能有构造器
字段声明	抽象类的字段声明可以是任意的	接口的字段默认都是 static 和 final 的

## 在Java中定义一个不做事且没有参数的构造方法的作用 \*

Java程序存在继承，在执行子类的构造方法时，如果没有用super()来调用父类特定的构造方法，则会调用父类中“没有参数的构造方法”。如果父类只定义了有参数的构造函数，而子类的构造函数没有用super调用父类那个特定的构造函数，就会出错。

## 在调用子类构造方法之前会先调用父类没有参数的构造方法，其目的是 \*

帮助子类做初始化工作。

## 一个类的构造方法的作用是什么？若一个类没有声明构造方法，改程序能正确执行吗？为什么？ \*

主要作用是完成对类对象的初始化工作。可以执行。因为一个类即使没有声明构造方法也会有默认的不带参数的构造方法。

## 构造方法有哪些特性？ \* \*

- 方法名称和类同名
- 不用定义返回值类型
- 不可以写return语句
- 构造方法可以被重载

## 变量 \* \*

- 类变量：独立于方法之外的变量，用static修饰。
- 实例变量：独立于方法之外的变量，不过没有static修饰。
- 局部变量：类的方法中的变量。
- 成员变量：成员变量又称全局变量，可分为类变量和实例变量，有static修饰为类变量，没有static修饰为实例变量。

	类变量	实例变量	局部变量
定义位置	类中, 方法外	类中, 方法外	方法中
初始值	有默认初始值	有默认初始值	无默认初始值
存储位置	方法区	堆	栈
生命周期	类何时被加载和卸载	实例何时被创建及销毁	方法何时被调用及结束调用

## 内部类 \* \*

内部类包括这四种：成员内部类、局部内部类、匿名内部类和静态内部类

- 成员内部类

1.成员内部类定义为位于另一个类的内部，成员内部类可以无条件访问外部类的所有成员属性和成员方法（包括 private 成员和静态成员）。

```
class Outer{
    private double a = 0;
    public static int b =1;
    public Outer(double a) {
        this.a = a;
    }

    class Inner {      //内部类
        public void fun() {
            System.out.println(a);
            System.out.println(b);
        }
    }
}
```

2.当成员内部类拥有和外部类同名的成员变量或者方法时，即默认情况下访问的是成员内部类的成员。如果要访问外部类的同名成员，需要以下面的形式进行访问：外部类. `this`.成员变量

3.在外部类中如果要访问成员内部类的成员，必须先创建一个成员内部类的对象，再通过指向这个对象的引用来访问。

4.成员内部类是依附外部类而存在的，如果要创建成员内部类的对象，前提是必须存在一个外部类的对象。创建成员内部类对象的一般方式如下：

```
class Outer{
    private double a = 0;
    public static int b =1;
    public Outer(){}
    public Outer(double a) {
        this.a = a;
        Inner inner = new Inner();
        inner.fun();      //调用内部类的方法
    }
}
```

```

class Inner { //内部类
    int b = 2;
    public void fun() {
        System.out.println(a);
        System.out.println(b); //访问内部类的b
        System.out.println(Outer.this.b); //访问外部类的b
    }
}
public class Main{
    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner(); //创建内部类的对象
    }
}

```

- 局部内部类

局部内部类是定义在一个方法或者一个作用域里面的类，它和成员内部类的区别在于局部内部类的访问仅限于方法内或者该作用域内。定义在实例方法中的局部类可以访问外部类的所有变量和方法，定义在静态方法中的局部类只能访问外部类的静态变量和方法。

```

class Outer {

    private int outer_a = 1;
    private static int static_b = 2;

    public void test1(){
        int inner_c =3;
        class Inner {
            private void fun(){
                System.out.println(outer_a);
                System.out.println(static_b);
                System.out.println(inner_c);
            }
        }
        Inner inner = new Inner(); //创建局部内部类
        inner.fun();
    }

    public static void test2(){
        int inner_d =3;
        class Inner {
            private void fun(){
                System.out.println(outer_a); //编译错误，定义在静态方法中的局部
                System.out.println(static_b);
                System.out.println(inner_d);
            }
        }
        Inner inner = new Inner();
        inner.fun();
    }
}

```

- 匿名内部类

匿名内部类只没有名字的内部类，在日常开发中使用较多。使用匿名内部类的前提条件：必须继承一个父类或实现一个接口。

```
interface Person {
    public void fun();
}

class Demo {
    public static void main(String[] args) {
        new Person() {
            public void fun() {
                System.out.println("hello,word");
            }
        }.fun();
    }
}
```

- 静态内部类

静态内部类也是定义在另一个类里面的类，只不过在类的前面多了一个关键字 `static`。静态内部类是不需要依赖于外部类的，并且它不能使用外部类的非 `static` 成员变量或者方法，这点很好理解，因为在没有外部类的对象的情况下，可以创建静态内部类的对象，如果允许访问外部类的非 `static` 成员就会产生矛盾，因为外部类的非 `static` 成员必须依附于具体的对象。

```
class Outer {
    int a = 1;
    static int b = 2;
    public Outer() {

    }

    static class Inner {
        public Inner() {
            System.out.println(a); // 报错，静态内部类不能访问非静态变量
            System.out.println(b);
        }
    }
}

public class Main{
    public static void main(String[] args) {
        Outer.Inner inner = new Outer.Inner();
    }
}
```

- 内部类的优点：

1. 内部类不为同一包的其他类所见，具有很好的封装性；
  2. 匿名内部类可以很方便的定义回调。
  3. 每个内部类都能独立的继承一个接口的实现，所以无论外部类是否已经继承了某个(接口的)实现，对于内部类都没有影响。
  4. 内部类有效实现了“多重继承”，优化 java 单继承的缺陷。
- 局部内部类和匿名内部类访问局部变量的时候，为什么变量必须要加上 `final`？

```
public class Main {
    public static void main(String[] args) {
```

```

    }

    public void fun(final int b) {
        final int a = 10;
        new Thread() {
            public void run() {
                System.out.println(a);
                System.out.println(b);
            }
        }.start();
    }
}

```

对于变量 `a` 可以从生命周期的角度理解，局部变量直接存储在栈中，当方法执行结束后，非 `final` 的局部变量就被销毁，而局部内部类对局部变量的引用依然存在，如果局部内部类要调用没有 `final` 修饰的局部变量时，就会造成生命周期不一致出错。

对于变量 `b`，其实是将 `fun` 方法中的变量 `b` 以参数的形式对匿名内部类中的拷贝（变量 `b` 的拷贝）进行赋值初始化。在 `run` 方法中访问的变量 `b` 根本就不是 `test` 方法中的局部变量 `b`，而是一个拷贝值，所以不存在生命周期不一致的问题，但如果在 `run` 方法中修改变量 `b` 的值会导致数据不一致，所以需要加 `final` 修饰。

## 重写与重载 \* \* \*

### 重载和重写的区别

- 重载：发生在同一个类中，方法名相同参数列表不同（参数类型不同、个数不同、顺序不同），与方法返回值和访问修饰符无关，即重载的方法不能根据返回类型进行区分。
- 重写：发生在父子类中，方法名、参数列表必须相同，返回值小于等于父类，抛出的异常小于等于父类，访问修饰符大于等于父类（里氏代换原则）；如果父类方法访问修饰符为 `private` 则子类中就不是重写。

### 构造器 (constructor) 是否可被重写 (override)

构造器可以被重载，不能被重写

### 重载的方法能否根据返回类型进行区分？为什么？

不能，因为调用时不能指定类型信息，编译器不知道你要调用哪个函数。

## == 和 equals 的区别 \* \* \*

- ==

对于基本数据类型，`==` 比较的是值；对于引用数据类型，`==` 比较的是内存地址。

- equals

对于没有重写 `equals` 方法的类，`equals` 方法和 `==` 作用类似；对于重写过 `equals` 方法的类，`equals` 比较的是值。

# hashCode 与 equals (为什么重写equals方法后， hashCode方法也必须重写) \* \* \*

- equals

先看下 `String` 类中重写的 `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;  
}
```

从源码中可以看到：

1. `equals` 方法首先比较的是内存地址，如果内存地址相同，直接返回 `true`；如果内存地址不同，再比较对象的类型，类型不同直接返回 `false`；类型相同，再比较值是否相同；值相同返回 `true`，值不同返回 `false`。总结一下，`equals` 会比较 **内存地址、对象类型、以及值**，内存地址相同，`equals` 一定返回 `true`；对象类型和值相同，`equals` 方法一定返回 `true`。
2. 如果没有重写 `equals` 方法，那么 `equals` 和 `==` 的作用相同，比较的是对象的地址值。

- `hashCode`

`hashCode` 方法返回对象的散列码，返回值是 `int` 类型的散列码。散列码的作用是确定该对象在哈希表中的索引位置。

关于 `hashCode` 有一些约定：

1. 两个对象相等，则 `hashCode` 一定相同。
  2. 两个对象有相同的 `hashCode` 值，它们不一定相等。
  3. `hashCode()` 方法默认是对堆上的对象产生独特值，如果没有重写 `hashCode()` 方法，则该类的两个对象的 `hashCode` 值肯定不同
- 为什么重写 `equals` 方法后，`hashCode` 方法也必须重写
    1. 根据规定，两个对象相等，`hashCode` 值也许相同，所以重写 `equals` 方法后，`hashCode` 方法也必须重写（面试官肯定不是想听这个答案）
    2. `hashCode` 在具有哈希机制的集合中起着非常关键的作用，比如 `HashMap`、`HashSet` 等。以 `HashSet` 为例，`HashSet` 的特点是存储元素时无序且唯一，在向 `HashSet` 中添加对象时，首先会计算对象的 `HashCode` 值来确定对象的存储位置，如果该位置没有其他对象，直接将该对象添加到该位置；如果该存储位置有存储其他对象（新添加的对象和该存储位置的对象的 `HashCode` 值相同），调用 `equals` 方法判断两个对象是否相同，如果相同，则添加对象失

败，如果不相同，则会将该对象重新散列到其他位置。所以重写 `equals` 方法后，`hashCode` 方法不重写的话，会导致所有对象的 `HashCode` 值都不相同，都能添加成功，那么 `HashSet` 中会出现很多重复元素，`HashMap` 也是同理（因为 `HashSet` 的底层就是通过 `HashMap` 实现的），会出现大量相同的 `Key`（`HashMap` 中的 `key` 是唯一的，但不同的 `key` 可以对应相同的 `value`）。所以重写 `equals` 方法后，`hashCode` 方法也必须重写。同时因为两个对象的 `hashCode` 值不同，则它们一定不相等，所以先计算对象的 `hashCode` 值可以在一定程度上判断两个对象是否相等，提高了集合的效率。总结一下，一共两点：**第一，在 `HashSet` 等集合中，不重写 `hashCode` 方法会导致其功能出现问题；第二，可以提高集合效率。**

## Java 中是值传递还是引用传递，还是两者共存 \* \*

这是一个很容易搞混又很难解释清楚的问题，先说结论，Java 中只有值传递

先看这样一段代码

```
public class Main{  
    public static void main(String[] args) {  
        int a = 1;  
        printValue(a);  
        System.out.println("a:" + a);  
    }  
    public static void printValue(int b){  
        b = 2;  
        System.out.println("b:"+ b);  
    }  
}
```

输出

```
b:2  
a:1
```

可以看到将 `a` 的值传到 `printValue` 方法中，并将其值改为2。但方法调用结束后，`a` 的值还是1，并未发生改变，所以这种情况下为值传递。

再看这段代码

```
public class Main{  
    public static void main(String[] args) {  
        Preson p = new Preson();  
        p.name = "zhangsan";  
        printValue(p);  
        System.out.println("p.name: " + p.name);  
    }  
    public static void printValue(Preson q){  
        q.name = "lisi";  
        System.out.println("q.name: "+ q.name);  
    }  
}  
class Preson{  
    public String name;  
}
```

输出结果

```
q.name: lisi  
p.name: lisi
```

在将 `p` 传入 `printValue` 方法后，方法调用结束，`p` 的 `name` 属性竟然被改变了！所以得出结论，参数为基本类型为值传递，参数为引用类型时为引用传递。这个结论是错误的，下面来看看判断是值传递还是值传递的关键是什么，先看定义

- 值传递：是指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。
- 引用传递：是指在调用函数时将实际参数的地址直接传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数。

从定义中可以明显看出，区分是值传递还是引用传递主要是看向方法中传递的是实际参数的副本还是实际参数的地址。上面第一个例子很明显是值传递，其实第二个例子中向 `printValue` 方法中传递的是一个引用的副本，只是这个副本引用和原始的引用指向的同一个对象，所以副本引用修改过对象属性后，通过原始引用查看对象属性肯定也是被修改过的。换句话说，`printValue` 方法中修改的是副本引用指向的对象的属性，不是引用本身，如果修改的是引用本身，那么原始引用肯定不受影响。看下面这个例子

```
public class Main{  
    public static void main(String[] args) {  
        Preson p = new Preson();  
        p.name = "zhangsan";  
        printValue(p);  
        System.out.println("p.name: " + p.name);  
    }  
    public static void printValue(Preson q){  
        q = new Preson();  
        q.name = "lisi";  
        System.out.println("q.name: "+ q.name);  
    }  
}  
class Preson{  
    public String name;  
}
```

输出结果

```
q.name: lisi  
p.name: zhangsan
```

可以看到将 `p` 传入 `printValue` 方法后，`printValue` 方法调用结束后，`p` 的属性 `name` 没有改变，这是因为在 `printValue` 方法中并没有改变副本引用 `q` 所指向的对象，而是改变了副本引用 `q` 本身，将副本引用 `q` 指向了另一个对象并对这个对象的属性进行修改，所以原始引用 `p` 所指向的对象不受影响。所以证明Java中只存在值传递。

## IO流 \*

Java IO流主要可以分为输入流和输出流。按照操作单元划分，可以划分为字节流和字符流。按照流的角色划分为节点流和处理流。

Java IO流的40多个类都是从4个抽象类基类中派生出来的。

- `InputStream`: 字节输入流
- `Reader`: 字符输入流

- OutputStream: 字节输出流
- Writer: 字符输出流

## BIO,NIO,AIO 有什么区别? \* \*

- **BIO (Blocking I/O):** 服务器实现模式为一个连接一个线程, 即客户端有连接请求时服务器就需要启动一个线程进行处理, 如果这个连接不做任何事情会造成不必要的线程开销, 可以通过线程池机制来改善。BIO方式适用于连接数目比较小且固定的架构, 这种方式对服务端资源要求比较高, 并发局限于应用中, 在jdk1.4以前是唯一的io
- **NIO (New I/O):** 服务器实现模式为一个请求一个线程, 即客户端发送的连接请求都会注册到多路复用器上, 多路复用器轮询到连接有IO请求时才启动一个线程进行处理。NIO方式适用于连接数目多且连接比较短(轻操作)的架构, 比如聊天服务器, 并发局限于应用中, 编程比较复杂, jdk1.4开始支持
- **AIO (Asynchronous I/O):** 服务器实现模式为一个有效请求一个线程, 客户端的IO请求都是由操作系统先完成了再通知服务器用其启动线程进行处理。AIO方式适用于连接数目多且连接比较长(重操作)的架构, 比如相册服务器, 充分调用OS参与并发操作, 编程比较复杂, jdk1.7开始支持。

这些概念看着比较枯燥, 可以从这个经典的烧开水的例子去理解

**BIO** : 来到厨房, 开始烧水NIO, 并坐在水壶面前一直等着水烧开。

**NIO** : 来到厨房, 开AIO始烧水, 但是我们不一直坐在水壶前面等, 而是做些其他事, 然后每隔几分钟到厨房看一下水有没有烧开。

**AIO** : 来到厨房, 开始烧水, 我们不一直坐在水壶前面等, 而是在水壶上面装个开关, 水烧开之后它会通知我。

## 反射 \* \* \*

JAVA反射机制是在运行状态中, 对于任意一个类, 都能够知道这个类的所有属性和方法; 对于任意一个对象, 都能够调用它的任意一个方法和属性; 这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

Java获取Class对象的三种方式

```
class Person {
    public String name = "zhangsan";
    public Person() {
    }
}
```

```
public class Main{
    public static void main(String[] args) throws ClassNotFoundException {
        //方式1
        Person p1 = new Person();
        Class c1 = p1.getClass();
        //方式2
        Class c2 = Person.class;
        //方式3可能会抛出ClassNotFoundException异常
        Class c3 = Class.forName("com.company");
    }
}
```

因为在一个类在JVM中只会有一个 `Class` 实例, 所以对 `c1`、`c2`、`c3` 进行 `equals` 比较时返回的都是 `true`。

## 反射优缺点：

- 优点：运行期类型的判断，动态加载类，提高代码灵活度。
- 缺点：性能比直接的java代码要慢很多。

## 反射应用场景：

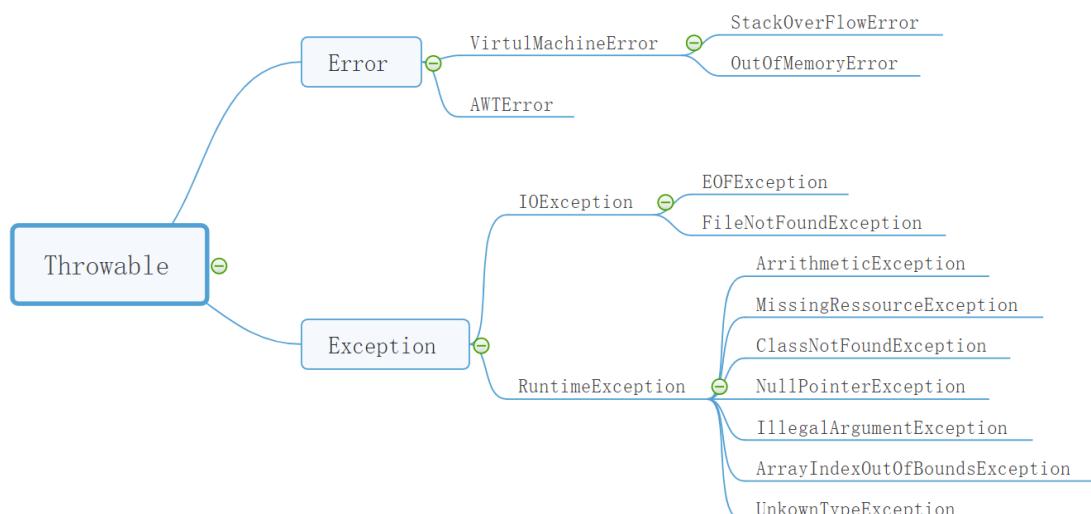
- Java的很多框架都用到了反射，例如 spring 中的xml的配置模式等
- 动态代理设计模式也采用了反射机制

# JAVA异常 \* \* \*

异常主要分为 `Error` 和 `Exception` 两种

- `Error`: `Error` 类以及他的子类的实例，代表了JVM本身错误。错误不能被程序员通过代码处理。
- `Exception`: `Exception` 以及他的子类，代表程序运行时发送的各种不期望发生的事件。可以被 Java 异常处理机制使用，是异常处理的核心。

## 异常框图



除了以上的分类，异常还能分为非检查异常和检查异常

- 非检查异常 (unchecked exception)** : 该类异常包括运行时异常 (`RuntimeException`及其子类) 和错误 (`Error`)。编译器不会进行检查并且不要求必须处理的异常，也就是说当程序中出现此类异常时，即使我们没有 `try-catch` 捕获它，也没有使用 `throws` 抛出该异常，编译也会正常通过。因为这样的异常发生的原因很可能是代码写的有问题。
- 检查异常 (checked exception)** : 除了 `Error` 和 `RuntimeException` 的其它异常。这是编译器要求必须处理的异常。这样的异常一般是由程序的运行环境导致的。因为程序可能被运行在各种未知的环境下，而程序员无法干预用户如何使用他编写的程序，所以必须处理这些异常。

下面来看下 `try{}catch(){}` 和 `finally{}` 之间的“恩恩怨怨”，这里有些乱，面试时问的也不是很多，实在记不住就算啦。还是先看代码猜结果。

```
public class Main{  
    public static void main(String[] args) {  
        int a = test1();  
        System.out.println(a);  
        int b = test2();  
        System.out.println(b);  
        int c = test3();  
        System.out.println(c);  
        int d = test4();  
    }  
}
```

```
System.out.println(d);
int e = test5();
System.out.println(e);
}

public static int test1(){
    int a = 1;
    try{
        a = 2;
        return a;
    }catch(Exception e){
        System.out.println("hello,test1");
        a = 3;
    }finally{
        a = 4;
    }
    return a;
}
//输出 2

public static int test2(){
    int a = 1;
    try{
        a = 2;
        return a;
    }catch(Exception e){
        System.out.println("hello,test2");
        a = 3;
        return a;
    }finally{
        a = 4;
    }
}
//输出 2

public static int test3(){
    int a = 1;
    try{
        a = 2/0;
        return a;
    }catch(Exception e){
        System.out.println("hello,test3");
        a = 3;
    }finally{
        a = 4;
    }
    return a;
}
//输出 hello,test3
// 4

public static int test4(){
    int a = 1;
    try{
        a = 2/0;
        return a;
    }
```

```

        }catch(Exception e){
            System.out.println("hello,test4");
            a = 3;
            return a;
        }finally{
            a = 4;
        }
    }
    //输出 hello,test4
    // 3

    public static int test5(){
        int a = 1;
        try{
            a = 2/0;
            return a;
        }catch(Exception e){
            a = 3;
            return a;
        }finally{
            a = 4;
            return a;
        }
    }

    //输出 4
}

```

如果没有仔细的研究过，应该好多会猜错，下面总结下规律。

- 从三个例子可以看出如果 `try{}` 中的代码没有异常，`catch(){}` 代码块中的代码不会执行。所以如果 `try{}` 和 `catch(){}` 都含有 `return` 时，无异常执行 `try{}` 中的 `return`，存在异常执行 `catch(){}` 的 `return`。
- 不管任何情况，就算 `try{}` 或 `catch(){}` 中含有 `return`，`finally{}` 中的代码一定会执行，那么为什么 `test1`、`test2`、`test3` 中的结果不是4呢，因为虽然 `finally` 是在 `return` 后面的表达式运算之后执行的，但此时并没有返回运算之后的值，而是把值保存起来，不管 `finally` 对该值做任何的改变，返回的值都不会改变，依然返回保存起来的值。也就是说方法的返回值是在 `finally` 运算之前就确定了的。
- 如果 `return` 的数据是引用数据类型，而在 `finally` 中对该引用数据类型的属性值的改变起作用，`try` 中的 `return` 语句返回的就是在 `finally` 中改变后的该属性的值。这个不理解可以看看上面提到的Java的值传递的问题。
- 如果 `finally{}` 中含有 `return`，会导致程序提前退出，不在执行 `try{}` 或 `catch(){}` 中的 `return`。所以 `test5` 返回的值是4。

最后总计一下 `try{}catch(){finally{}}` 的执行顺序。

- 先执行 `try` 中的语句，包括 `return` 后面的表达式；
- 有异常时，执行 `catch` 中的语句，包括 `return` 后面的表达式，无异常跳过 `catch` 语句；
- 然后执行 `finally` 中的语句，如果 `finally` 里面有 `return` 语句，执行 `return` 语句，程序结束；
- `finally{}` 中没有 `return` 时，无异常执行 `try` 中的 `return`，如果有异常时则执行 `catch` 中的 `return`。前两步执行的 `return` 只是确定返回的值，程序并未结束，`finally{}` 执行之后，最后将前两步确定的 `return` 的返回值返回。

## JAVA注解 \* \*

面试问的不多，但是在使用框架开发时会经常使用，但东西太多了，这里只是简单介绍下概念。

Annotation 注解可以看成是java中的一种标记记号，用来给java中的类，成员，方法，参数等任何程序元素添加一些额外的说明信息，同时不改变程序语义。注解可以分为三类：基本注解，元注解，自定义注解

- 标准注解

1. @Deprecated：该注解用来说明程序中的某个元素（类，方法，成员变量等）已经不再使用，如果使用的话的编译器会给出警告。

2. @SuppressWarnings(value="")：用来抑制各种可能出现的警告。

3. @Override：用来说明子类方法覆盖了父类的方法，保护覆盖方法的正确使用

- 元注解（元注解也称为元数据注解，是对注解进行标注的注解，元注解更像是一种对注解的规范说明，用来对定义的注解进行行为的限定。例如说明注解的生存周期，注解的作用范围等）

1. @Target(value=" ")：该注解是用来限制注解的使用范围的，即该注解可以用于哪些程序元素。

2. @Retention(value=" ")：用于说明注解的生存周期

3. @Document：用来说明指定被修饰的注解可以被javadoc.exe工具提取进入文档中，所有使用了该注解进行标注的类在生成API文档时都在包含该注解的说明。

4. @Inherited：用来说明使用了该注解的父类，其子类会自动继承该注解。

5. @Repeatable：java1.8新出的元注解，如果需要在给程序元素使用相同类型的注解，则需将该注解标注上。

- 自定义注解：用@Interface来声明注解。

## JAVA泛型 \* \* \*

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

- 泛型擦除（这是面试考察泛型时经常问到的问题）

Java的泛型基本上都是在编译器这个层次上实现的，在生成的字节码中是不包含泛型中的类型信息的，使用泛型的时候加上类型参数，在编译器编译的时候会去掉，这个过程成为类型擦除。看下面代码

```
public class Main{
    public static void main(String[] args) {
        ArrayList<Integer> arrayList1 = new ArrayList<>();
        ArrayList<String> arrayList2 = new ArrayList<>();

        System.out.println(arrayList1.getClass() == arrayList2.getClass());
    }
}
```

输出结果

```
true
```

可以看到 `ArrayList<Integer>` 和 `ArrayList<String>` 的原始类型是相同，在编译成字节码文件后都会变成 `List`，JVM看到的只有 `List`，看不到泛型信息，这就是泛型的类型擦除。在看下面这段代码

```
public class Main{
    public static void main(String[] args) throws Exception {
        ArrayList<Integer> arrayList = new ArrayList<>();
        arrayList.add(1);
        arrayList.getClass().getMethod("add",
Object.class).invoke(arrayList, "a");
        System.out.println(arrayList.get(0));
        System.out.println(arrayList.get(1));
    }
}
```

输出

```
1
```

```
a
```

可以看到通过反射进行 add 操作，`ArrayList<Integer>` 竟然可以存储字符串，这是因为在反射就是在运行期调用的 add 方法，在运行期泛型信息已经被擦除。

- 既然存在类型擦除，那么Java是如何保证在 `ArrayList<Integer>` 添加字符串会报错呢？

Java编译器是通过先检查代码中泛型的类型，然后在进行类型擦除，再进行编译。

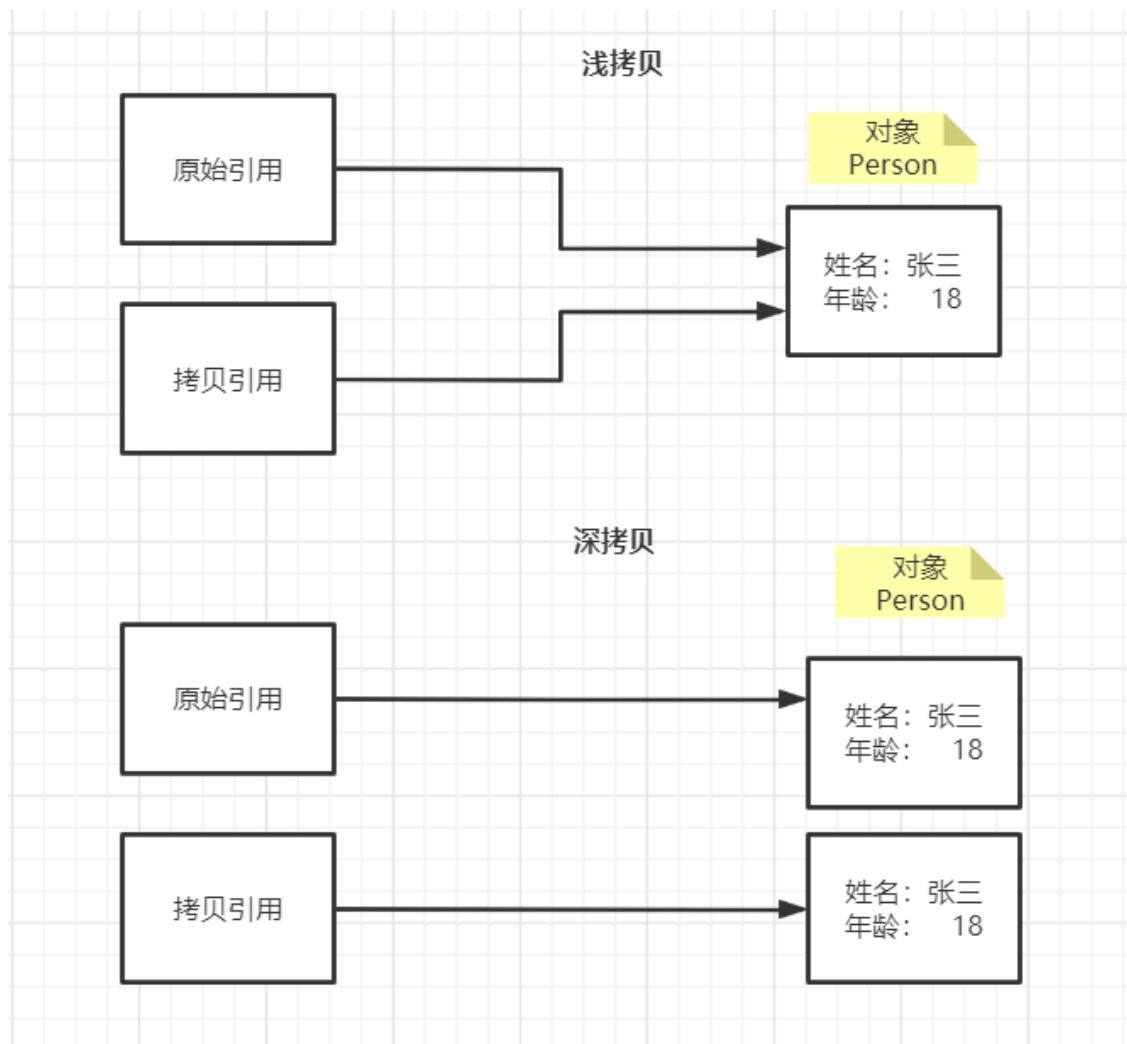
## JAVA序列化 \* \*

- 序列化：将对象写入到IO流中
- 反序列化：从IO流中恢复对象
- 序列化的意义：将Java对象转换成字节序列，这些字节序列更加便于通过网络传输或存储在磁盘上，在需要时可以通过反序列化恢复成原来的对象。
- 实现方式：
  - 实现**Serializable**接口
  - 实现**Externalizable**接口
- 序列化的注意事项：
  - 对象的类名、实例变量会被序列化；方法、类变量、`transient`实例变量都不会被序列化。
  - 某个变量不被序列化，可以使用 `transient`修饰。
  - 序列化对象的引用类型成员变量，也必须是可序列化的，否则，会报错。
  - 反序列化时必须有序列化对象的 `class` 文件。

## 深拷贝与浅拷贝 \* \* \*

- 深拷贝：对基本数据类型进行值传递，对引用数据类型，创建一个新的对象，并复制其内容，两个引用指向两个对象，但对象内容相同。
- 浅拷贝：对基本数据类型进行值传递，对引用数据类型复制一个引用指向原始引用的对象，就是复制的引用和原始引用指向同一个对象。

具体区别看下图



- 深拷贝的实现方式

1. 重载 clone 方法

```

public class Main{
    public static void main(String[] args) throws NoSuchMethodException,
    InvocationTargetException, IllegalAccessException,
    CloneNotSupportedException {
        Address s = new Address("天津");
        Person p = new Person("张三", 23, s);
        System.out.println("克隆前的地址: " + p.getAddress().getName());
        Person cloneP = (Person) p.clone();
        cloneP.getAddress().setName("北京");
        System.out.println("克隆后的地址: " +
cloneP.getAddress().getName());
    }
}

class Address implements Cloneable {
    private String city;
    public Address(String name){
        this.city = name;
    }
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
    public String getName() {
        return city;
    }
}

```

```

    }
    public void setName(String name) {
        this.city = name;
    }
}
class Person implements Cloneable{
    private String name;
    private int age;
    private Address address;
    public Person(String name, int age, Address address){
        this.name = name;
        this.age = age;
        this.address = address;
    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        Person person = (Person) super.clone();
        person.address = (Address)address.clone();
        return person;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public Address getAddress() {
        return address;
    }
    public void setAddress(Address address) {
        this.address = address;
    }
}

```

输出

```

克隆前的地址: 天津
克隆后的地址: 北京

```

其实就是 Person 类和 Address 类都要重写 clone 方法，这里面需要注意的一点是 super.clone() 为浅克隆，所以在在 Person 类中重写 clone 方法时， address 对象需要调用 address.clone() 重新赋值，因为 address 类型为引用类型。

## 2. 序列化

```

public class Main{

```

```
    public static void main(String[] args) throws IOException,
    ClassNotFoundException {
        Address s = new Address("天津");
        Person p = new Person("张三", 23, s);
        System.out.println("克隆前的地址: " + p.getAddress().getName());
        Person cloneP = (Person) p.deepClone();
        cloneP.getAddress().setName("北京");
        System.out.println("克隆后的地址: " +
cloneP.getAddress().getName());
    }
}

class Address implements Serializable{
    private String city;
    public Address(String name){
        this.city = name;
    }

    public String getName() {
        return city;
    }

    public void setName(String name) {
        this.city = name;
    }
}
class Person implements Serializable{
    private String name;
    private int age;
    private Address address;
    public Person(String name, int age, Address address){
        this.name = name;
        this.age = age;
        this.address = address;
    }

    public Object deepClone() throws IOException, ClassNotFoundException
{
    ByteArrayOutputStream bos = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(bos);
    oos.writeObject(this);
    ByteArrayInputStream bis = new
ByteArrayInputStream(bos.toByteArray());
    ObjectInputStream ois = new ObjectInputStream(bis);
    return ois.readObject();
}

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }
}
```

```
public void setAge(int age) {  
    this.age = age;  
}  
public Address getAddress() {  
    return address;  
}  
public void setAddress(Address address) {  
    this.address = address;  
}  
}
```

输出

```
克隆前的地址: 天津  
克隆后的地址: 北京
```

## 常见的Object方法 \* \* \*

这些方法都很重要，面试经常会问到，要结合其他知识将这些方法理解透彻

- `Object clone()`: 创建与该对象的类相同的新对象
- `boolean equals(Object)`: 比较两对象是否相等
- `void finalize()`: 当垃圾回收器确定不存在对该对象的更多引用时，对象垃圾回收器调用该方法
- `Class getClass()`: 返回一个对象运行时的实例类
- `int hashCode()`: 返回该对象的散列码值
- `void notify()`: 唤醒等待在该对象的监视器上的一个线程
- `void notifyAll()`: 唤醒等待在该对象的监视器上的全部线程
- `String toString()`: 返回该对象的字符串表示
- `void wait()`: 在其他线程调用此对象的 `notify()` 方法或 `notifyAll()` 方法前，导致当前线程等待

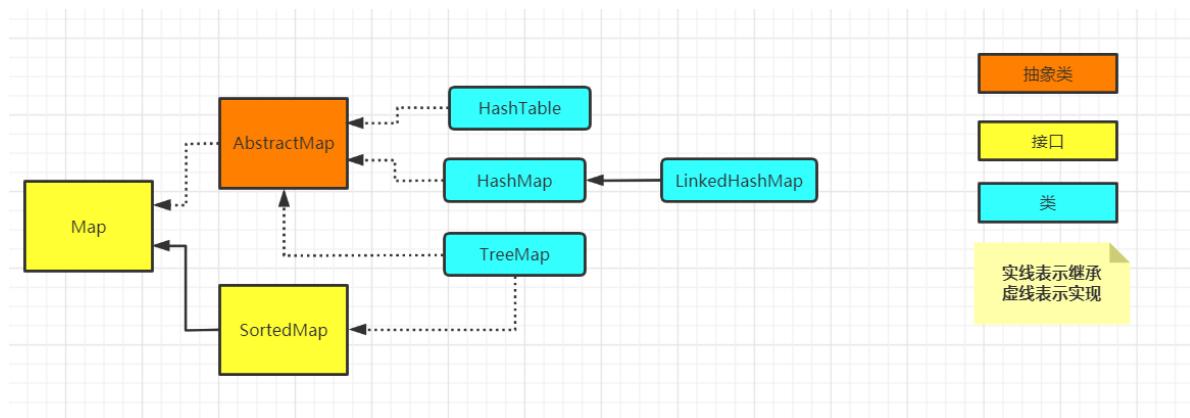
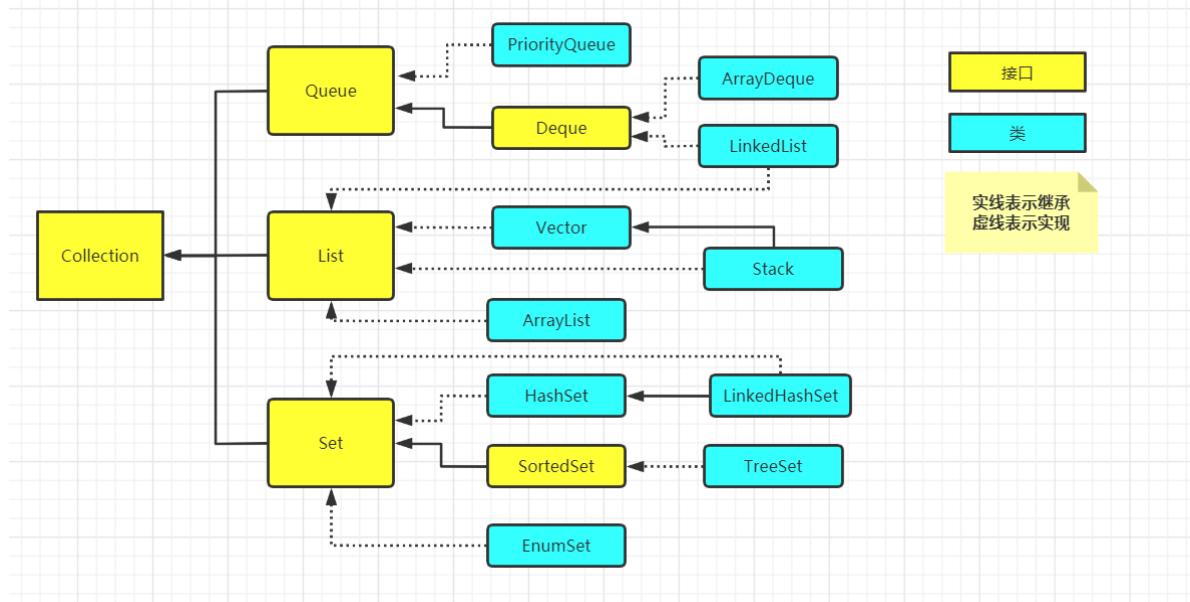
关注微信公众号“路人zhang”，获取更多面试技巧及资料

关注知乎“路人zhang”，聊聊码农那些事



## 常用的集合类有哪些? \* \* \*

Map接口和Collection接口是所有集合框架的父接口。下图中的实线和虚线看着有些乱，其中接口与接口之间如果有联系为继承关系，类与类之间如果有联系为继承关系，类与接口之间则是类实现接口。重点掌握的抽象类有 `HashMap`, `LinkedList`, `HashTable`, `ArrayList`, `HashSet`, `Stack`, `TreeSet`, `TreeMap`。注意：Collection 接口不是 Map 的父接口。



## List, Set, Map三者的区别? \* \* \*

- **List**: **有序集合**（有序指存入的顺序和取出的顺序相同，不是按照元素的某些特性排序），**可存储重复元素，可存储多个 null**。
- **Set**: **无序集合**（元素存入和取出顺序不一定相同），**不可存储重复元素，只能存储一个 null**。
- **Map**: 使用键值对的方式对元素进行存储，**key** 是无序的，且是唯一的。**value** 值不唯一。不同的 **key** 值可以对应相同的 **value** 值。

## 常用集合框架底层数据结构 \* \* \*

- **List**:
  1. `ArrayList`: 数组
  2. `LinkedList`: 双线链表
- **Set**:
  1. `HashSet`: 底层基于 `HashMap` 实现，`HashSet` 存入读取元素的方式和 `HashMap` 中的 `key` 是一致的。
  2. `TreeSet`: 红黑树
- **Map**:

1. `HashMap` : JDK1.8之前 `HashMap` 由数组+链表组成的，JDK1.8之后有数组+链表/红黑树组成，当链表长度大于8时，链表转化为红黑树，当长度小于6时，从红黑树转化为链表。这样做的目的是能提高 `HashMap` 的性能，因为红黑树的查找元素的时间复杂度远小于链表。
2. `HashTable` : 数组+链表
3. `TreeMap` : 红黑树

## 哪些集合类是线程安全的? \* \* \*

- `Vector` : 相当于有同步机制的 `ArrayList`
- `Stack` : 栈
- `HashTable`
- `Enumeration` : 枚举

## 迭代器 Iterator 是什么 \*

`Iterator` 是 Java 迭代器最简单的实现，它不是一个集合，它是一种用于访问集合的方法，`Iterator` 接口提供遍历任何 `Collection` 的接口。

## Java集合的快速失败机制“fail-fast”和安全失败机制“fail-safe”是什么? \* \* \*

- 快速失败

Java的快速失败机制是Java集合框架中的一种错误检测机制，当多个线程同时对集合中的内容进行修改时可能就会抛出 `ConcurrentModificationException` 异常。其实不仅仅是在多线程状态下，在单线程中用增强 `for` 循环中一边遍历集合一边修改集合的元素也会抛出

`ConcurrentModificationException` 异常。看下面代码

```
public class Main{
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        for(Integer i : list){
            list.remove(i); //运行时抛出ConcurrentModificationException异常
        }
    }
}
```

正确的做法是用迭代器的 `remove()` 方法，便可正常运行。

```
public class Main{
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        Iterator<Integer> it = list.iterator();
        while(it.hasNext()){
            it.remove();
        }
    }
}
```

造成这种情况的原因是什么？细心的同学可能已经发现两次调用的 `remove()` 方法不同，一个带参数，一个不带参数，这个后面再说，经过查看 `ArrayList` 源码，找到了抛出异常的代码

```
final void checkForComodification() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
}
```

从上面代码中可以看到如果 modCount 和 expectedModCount 这两个变量不相等就会抛出 ConcurrentModificationException 异常。那这两个变量又是什么呢？继续看源码

```
protected transient int modCount = 0; //在AbstractList中定义的变量
```

```
int expectedModCount = modCount; //在ArrayList中的内部类Itr中定义的变量
```

从上面代码可以看到， modCount 初始值为0，而 expectedModCount 初始值等于 modCount。也就是说在遍历的时候直接调用集合的 remove() 方法会导致 modCount 不等于 expectedModCount 进而抛出 ConcurrentModificationException 异常，而使用迭代器的 remove() 方法则不会出现这种问题。那么只能在看看 remove() 方法的源码找找原因了

```
public E remove(int index) {
    rangeCheck(index);

    modCount++;
    E oldValue = elementData(index);

    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                        numMoved);
    elementData[--size] = null; // clear to let GC do its work

    return oldValue;
}
```

从上面代码中可以看到只有 modCount++ 了，而 expectedModCount 没有操作，当每一次迭代时，迭代器会比较 expectedModCount 和 modCount 的值是否相等，所以在调用 remove() 方法后， modCount 不等于 expectedModCount 了，这时就报 ConcurrentModificationException 异常。但用迭代器中 remove() 的方法为什么不抛异常呢？原来迭代器调用的 remove() 方法和上面的 remove() 方法不是同一个！迭代器调用的 remove() 方法长这样：

```
public void remove() {
    if (lastRet < 0)
        throw new IllegalStateException();
    checkForComodification();

    try {
        ArrayList.this.remove(lastRet);
        cursor = lastRet;
        lastRet = -1;
        expectedModCount = modCount; //这行代码保证了
        expectedModCount 和 modCount 是相等的
    } catch (IndexOutOfBoundsException ex) {
        throw new ConcurrentModificationException();
    }
}
```

从上面代码可以看到 `expectedModCount = modCount`，所以迭代器的 `remove()` 方法保证了 `expectedModCount` 和 `modCount` 是相等的，进而保证了在增强 `for` 循环中修改集合内容不会报 `ConcurrentModificationException` 异常。

上面介绍的只是单线程的情况，用迭代器调用 `remove()` 方法即可正常运行，但如果是多线程会怎么样呢？

答案是在多线程的情况下即使用了迭代器调用 `remove()` 方法，还是会报 `ConcurrentModificationException` 异常。这又是为什么呢？还是要从 `expectedModCount` 和 `modCount` 这两个变量入手分析，刚刚说了 `modCount` 在 `AbstractList` 类中定义，而 `expectedModCount` 在 `ArrayList` 内部类中定义，所以 `modCount` 是个共享变量而 `expectedModCount` 是属于线程各自的。简单说，线程1更新了 `modCount` 和属于自己的 `expectedModCount`，而在线程2看来只有 `modCount` 更新了，`expectedModCount` 并未更新，所以会抛出 `ConcurrentModificationException` 异常。

- 安全失败

采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。所以在遍历过程中对原集合所作的修改并不能被迭代器检测到，所以不会抛出 `ConcurrentModificationException` 异常。缺点是迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生了修改，迭代器是无法访问到修改后的内容。

`java.util.concurrent` 包下的容器都是安全失败，可以在多线程下并发使用。

## 如何边遍历边移除 Collection 中的元素? \* \* \*

从上文“快速失败机制”可知在遍历集合时如果直接调用 `remove()` 方法会抛出 `ConcurrentModificationException` 异常，所以使用迭代器中调用 `remove()` 方法。

## Array 和 ArrayList 有何区别? \* \* \*

- `Array` 可以包含基本类型和对象类型，`ArrayList` 只能包含对象类型。
- `Array` 大小是固定的，`ArrayList` 的大小是动态变化的。`(ArrayList 的扩容是个常见面试题)`
- 相比于 `Array`，`ArrayList` 有着更多的内置方法，如 `addAll()`，`removeAll()`。
- 对于基本类型数据，`ArrayList` 使用自动装箱来减少编码工作量；而当处理固定大小的基本数据类型的时候，这种方式相对比较慢，这时候应该使用 `Array`。

## comparable 和 comparator的区别? \* \*

- `comparable` 接口出自 `java.lang` 包，可以理解为一个内比较器，因为实现了 `Comparable` 接口的类可以和自己比较，要和其他实现了 `Comparable` 接口类比较，可以使用 `compareTo(Object obj)` 方法。`compareTo` 方法的返回值是 `int`，有三种情况：
  1. 返回正整数（比较者大于被比较者）
  2. 返回0（比较者等于被比较者）
  3. 返回负整数（比较者小于被比较者）
- `comparator` 接口出自 `java.util` 包，它有一个 `compare(Object obj1, Object obj2)` 方法用来排序，返回值同样是 `int`，有三种情况，和 `compareTo` 类似。

它们之间的区别：很多包装类都实现了 `comparable` 接口，像 `Integer`、`String` 等，所以直接调用 `Collections.sort()` 直接可以使用。如果对类里面自带的自然排序不满意，而又不能修改其源代码的情况下，使用 `Comparator` 就比较合适。此外使用 `Comparator` 可以避免添加额外的代码与我们的目标类耦合，同时可以定义多种排序规则，这一点是 `Comparable` 接口没法做到的，从灵活性和扩展性讲 `Comparator` 更优，故在面对自定义排序的需求时，可以优先考虑使用 `Comparator` 接口。

# Collection 和 Collections 有什么区别? \* \*

- `collection` 是一个**集合接口**。它提供了对集合对象进行基本操作的通用接口方法。
- `Collections` 是一个**包装类**。它包含有各种有关集合操作的**静态多态方法**，例如常用的 `sort()` 方法。此类**不能实例化**，就像一个**工具类**，服务于Java的 `collection` 框架。

## List集合

### 遍历一个 List 有哪些不同的方式? \* \*

先说一下常见的元素在内存中的存储方式，主要有两种：

1. 顺序存储 (Random Access)：相邻的数据元素在内存中的位置也是相邻的，可以根据元素的位置 (如 `ArrayList` 中的下表) 读取元素。
2. 链式存储 (Sequential Access)：每个数据元素包含它下一个元素的内存地址，在内存中不要求相邻。例如 `LinkedList`。

主要的遍历方式主要有三种：

1. `for` 循环遍历：遍历者自己在集合外部维护一个计数器，依次读取每一个位置的元素。
2. `Iterator` 遍历：基于顺序存储集合的 `Iterator` 可以直接按位置访问数据。基于链式存储集合的 `Iterator`，需要保存当前遍历的位置，然后根据当前位置来向前或者向后移动指针。
3. `foreach` 遍历：`foreach` 内部也是采用了 `Iterator` 的方式实现，但使用时不需要显示地声明 `Iterator`。

那么对于以上三种遍历方式应该如何选取呢？

在Java集合框架中，提供了一个 `RandomAccess` 接口，该接口没有方法，只是一个标记。通常用来标记 `List` 的实现是否支持 `RandomAccess`。所以在遍历时，可以先判断是否支持 `RandomAccess` (`list instanceof RandomAccess`)，如果支持可用 `for` 循环遍历，否则建议用 `Iterator` 或 `foreach` 遍历。

### ArrayList的扩容机制 \* \* \*

先说下结论，一般面试时需要记住，`ArrayList` 的初始容量为10，扩容时对是旧的容量值加上旧的容量数值进行右移一位（位运算，相当于除以2，位运算的效率更高），所以每次扩容都是旧的容量的1.5倍。

具体的实现大家可查看下 `ArrayList` 的源码。

### ArrayList 和 LinkedList 的区别是什么? \* \* \*

- 是否线程安全：`ArrayList` 和 `LinkedList` 都是不保证线程安全的
- 底层实现：`ArrayList` 的底层实现是数组，`LinkedList` 的底层是双向链表。
- 内存占用：`ArrayList` 会存在一定的空间浪费，因为每次扩容都是之前的1.5倍，而 `LinkedList` 中的每个元素要存放直接后继和直接前驱以及数据，所以对于每个元素的存储都要比 `ArrayList` 花费更多的空间。
- 应用场景：`ArrayList` 的底层数据结构是数组，所以在插入和删除元素时的时间复杂度都会收到位置的影响，平均时间复杂度为  $O(n)$ ，在读取元素的时候可以根据下标直接查找到元素，不受位置的影响，平均时间复杂度为  $O(1)$ ，所以 `ArrayList` 更加适用于**多读，少增删的场景**。`LinkedList` 的底层数据结构是双向链表，所以插入和删除元素不受位置的影响，平均时间复杂度为  $O(1)$ ，如果是在指定位置插入则是  $O(n)$ ，因为在插入之前需要先找到该位置，读取元素的平均时间复杂度为  $O(n)$ 。所以 `LinkedList` 更加适用于**多增删，少读写的场景**。

## ArrayList 和 Vector 的区别是什么? \* \* \*

- 相同点
  1. 都实现了 List 接口
  2. 底层数据结构都是数组
- 不同点
  1. 线程安全: Vector 使用了 synchronized 来实现线程同步, 所以是线程安全的, 而 ArrayList 是线程不安全的。
  2. 性能: 由于 vector 使用了 synchronized 进行加锁, 所以性能不如 ArrayList。
  3. 扩容: ArrayList 和 vector 都会根据需要动态的调整容量, 但是 ArrayList 每次扩容为旧容量的1.5倍, 而 vector 每次扩容为旧容量的2倍。

## 简述 ArrayList、Vector、LinkedList 的存储性能和特性? \* \* \*

- ArrayList 底层数据结构为数组, 对元素的读取速度快, 而增删数据慢, 线程不安全。
- LinkedList 底层为双向链表, 对元素的增删数据快, 读取慢, 线程不安全。
- vector 的底层数据结构为数组, 用 synchronized 来保证线程安全, 性能较差, 但线程安全。

## Set集合

### 说一下 HashSet 的实现原理 \* \* \*

HashSet 的底层是 HashMap, 默认构造函数是构建一个初始容量为16, 负载因子为0.75 的 HashMap。 HashSet 的值存放于 HashMap 的 key 上, HashMap 的 value 统一为 PRESENT。

### HashSet如何检查重复? (HashSet是如何保证数据不可重复的?) \* \* \*

这里面涉及到了 hashCode() 和 equals() 两个方法。

- equals()

先看下 string 类中重写的 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;
}
```

从源码中可以看到：

1. `equals` 方法首先比较的是内存地址，如果内存地址相同，直接返回 `true`；如果内存地址不同，再比较对象的类型，类型不同直接返回 `false`；类型相同，再比较值是否相同；值相同返回 `true`，值不同返回 `false`。总结一下，`equals` 会比较内存地址、对象类型、以及值，内存地址相同，`equals` 一定返回 `true`；对象类型和值相同，`equals` 方法一定返回 `true`。
2. 如果没有重写 `equals` 方法，那么 `equals` 和 `==` 的作用相同，比较的是对象的地址值。

- `hashCode`

`hashCode` 方法返回对象的散列码，返回值是 `int` 类型的散列码。散列码的作用是确定该对象在哈希表中的索引位置。

关于 `hashCode` 有一些约定：

1. 两个对象相等，则 `hashCode` 一定相同。
2. 两个对象有相同的 `hashCode` 值，它们不一定相等。
3. `hashCode()` 方法默认是对堆上的对象产生独特值，如果没有重写 `hashCode()` 方法，则该类的两个对象的 `hashCode` 值肯定不同

介绍完 `equals()` 方法和 `hashCode()` 方法，继续说下 `HashSet` 是如何检查重复的。

`HashSet` 的特点是存储元素时无序且唯一，在向 `HashSet` 中添加对象时，首先会计算对象的 `HashCode` 值来确定对象的存储位置，如果该位置没有其他对象，直接将该对象添加到该位置；如果该存储位置有存储其他对象（新添加的对象和该存储位置的对象的 `HashCode` 值相同），调用 `equals` 方法判断两个对象是否相同，如果相同，则添加对象失败，如果不相同，则会将该对象重新散列到其他位置。

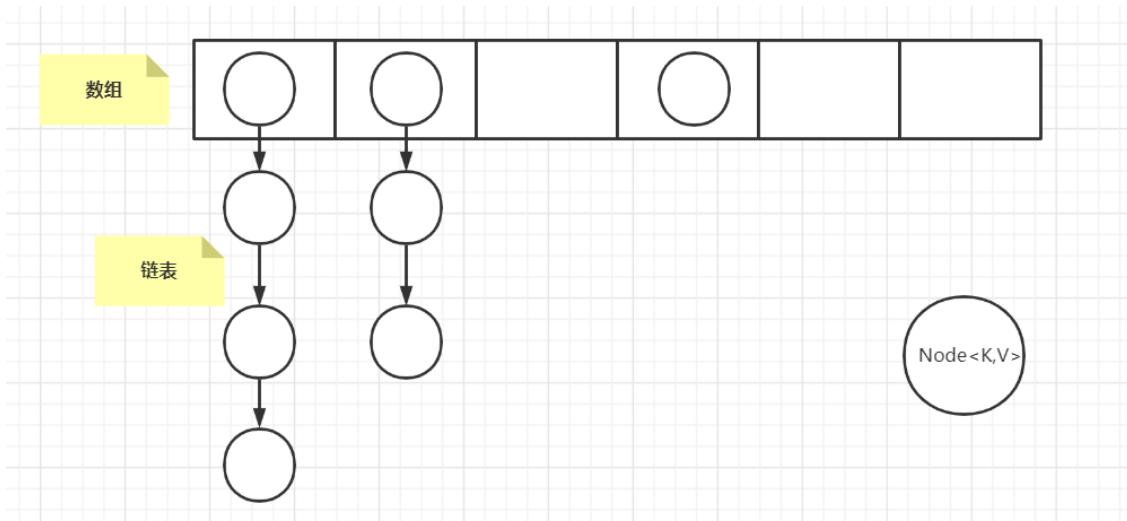
## HashSet与HashMap的区别 \* \* \*

HashMap	HashSet
实现了 <code>Map</code> 接口	实现了 <code>Set</code> 接口
存储键值对	存储对象
<code>key</code> 唯一， <code>value</code> 不唯一	存储对象唯一
<code>HashMap</code> 使用键（ <code>Key</code> ）计算 <code>Hashcode</code>	<code>HashSet</code> 使用成员对象来计算 <code>hashcode</code> 值
速度相对较快	速度相对较慢

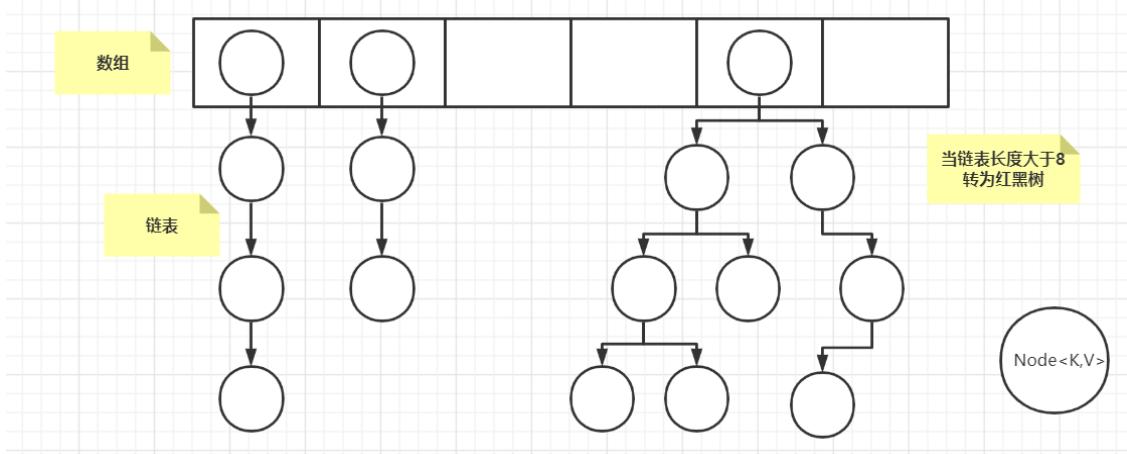
## Map集合

### HashMap在JDK1.7和JDK1.8中有哪些不同？HashMap的底层实现 \* \* \*

- JDK1.7的底层数据结构(数组+链表)



- JDK1.8的底层数据结构(数组+链表)



- JDK1.7的Hash函数

```
static final int hash(int h){
    h ^= (h >>> 20) ^ (h >>> 12);
    return h^(h >>> 7) ^ (h >>> 4);
}
```

- JDK1.8的Hash函数

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

JDK1.8的函数经过了一次异或一次位运算一共两次扰动，而JDK1.7经过了四次位运算五次异或一共九次扰动。这里简单解释下JDK1.8的hash函数，面试经常问这个，两次扰动分别是  
`key.hashCode()`与`key.hashCode()`右移16位进行异或。这样做的目的是，高16位不变，低16位与高16位进行异或操作，进而减少碰撞的发生，高低Bit都参与到Hash的计算。如何不进行扰动处理，因为hash值有32位，直接对数组的长度求余，起作用只是hash值的几个低位。

HashMap在JDK1.7和JDK1.8中有哪些不同点：

	JDK1.7	JDK1.8	JDK1.8的优势
底层结构	数组+链表	数组+链表/红黑树(链表大于8)	避免单条链表过长而影响查询效率，提高查询效率
hash值计算方式	$9 \text{次扰动} = 4 \text{次位运算} + 5 \text{次异或运算}$	$2 \text{次扰动} = 1 \text{次位运算} + 1 \text{次异或运算}$	可以均匀地把之前的冲突的节点分散到新的桶（具体细节见下面扩容部分）
插入数据方式	头插法（先讲原位置的数据移到后1位，再插入数据到该位置）	尾插法（直接插入到链表尾部/红黑树）	解决多线程造成死循环问题
扩容后存储位置的计算方式	重新进行hash计算	原位置或原位置+旧容量	省去了重新计算hash值的时间

## HashMap 的长度为什么是2的幂次方 \* \* \*

因为 HashMap 是通过 key 的hash值来确定存储的位置，但Hash值的范围是-2147483648到2147483647，不可能建立一个这么大的数组来覆盖所有hash值。所以在计算完hash值后会对数组的长度进行取余操作，如果数组的长度是2的幂次方，`(length - 1)&hash` 等同于 `hash%length`，可以用`(length - 1)&hash`这种位运算来代替%取余的操作进而提高性能。

## HashMap的put方法的具体流程? \* \*

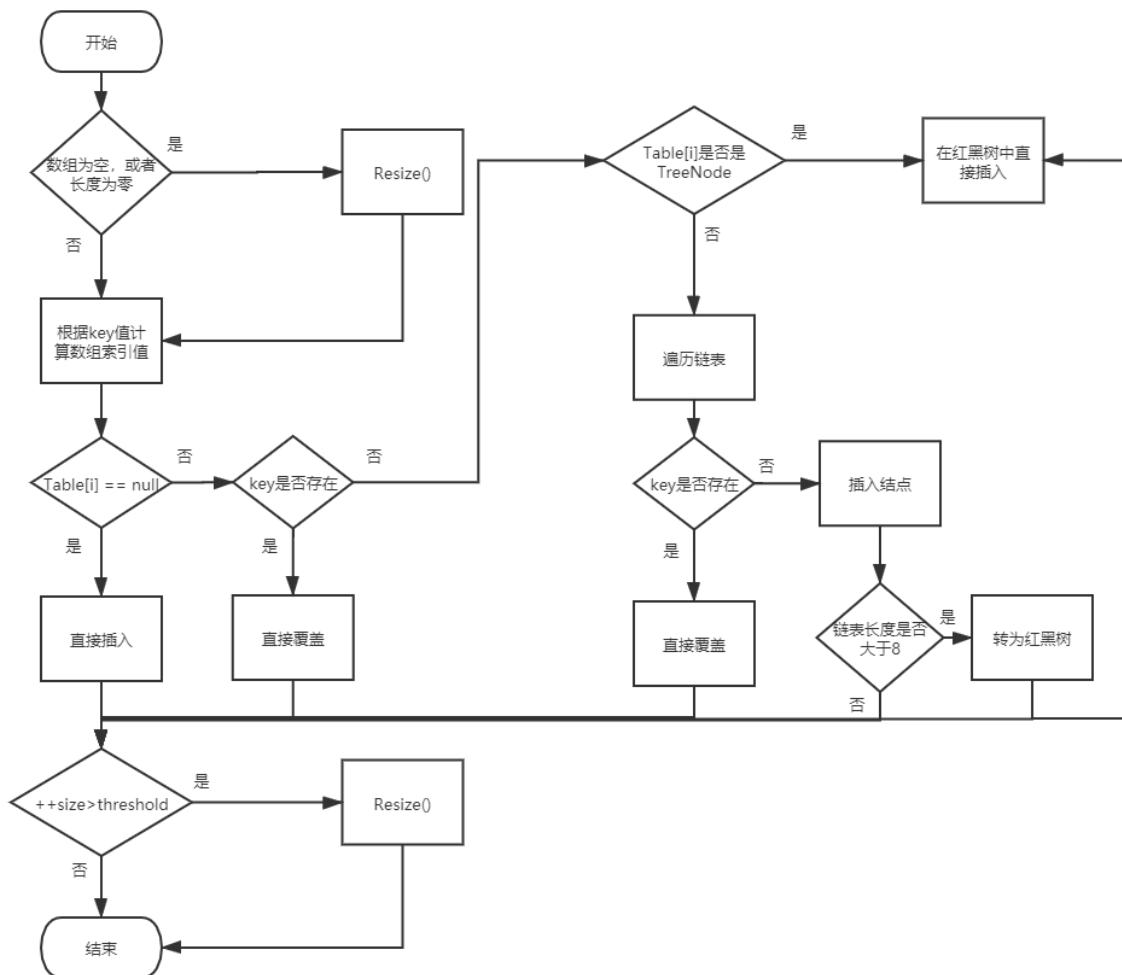
HashMap的主要流程可以看下面这个流程图，逻辑非常清晰。

计算索引、逻辑判断

扩容

链表

红黑树



## HashMap的扩容操作是怎么实现的？ \* \* \*

- 初始值为16，负载因子为0.75，阈值为负载因子\*容量
- `resize()` 方法是在 `hashmap` 中的键值对大于阈值时或者初始化时，就调用 `resize()` 方法进行扩容。
- 每次扩容，容量都是之前的两倍
- 扩容时有个判断 `e.hash & oldCap` 是否为零，也就是相当于hash值对数组长度的取余操作，若等于0，则位置不变，若等于1，则位置变为原位置加旧容量。

源码如下：

```

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) { //如果旧容量已经超过最大值，阈值为整数最  
大值
            threshold = Integer.MAX_VALUE;
            return oldTab;
        } else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&  
oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; //没有超过最大值就变为原来的2倍
    }
}

```

```

    }

    else if (oldThr > 0)
        newCap = oldThr;

    else {
        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 {
                    Node<K,V> loHead = null, loTail = null;//loHead,loTail 代表扩容后在原位置
                    Node<K,V> hiHead = null, hiTail = null;//hiHead,hiTail 代表扩容后在原位置+旧容量
                    Node<K,V> next;
                    do {
                        next = e.next;
                        if ((e.hash & oldCap) == 0) { //判断是否为零, 为零赋值到loHead, 不为零赋值到hiHead
                            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; //loHead放在原位置
                    }
                    if (hiTail != null) {
                        hiTail.next = null;
                    }
                }
            }
        }
    }
}

```

```

        newTab[j + oldCap] = hiHead; //hiHead放在原位置+旧容量
    }
}
}
}
return newTab;
}

```

## HashMap默认加载因子为什么选择0.75?

这个主要是考虑空间利用率和查询成本的一个折中。如果加载因子过高，空间利用率提高，但是会使得哈希冲突的概率增加；如果加载因子过低，会频繁扩容，哈希冲突概率降低，但是会使得空间利用率变低。具体为什么是0.75，不是0.74或0.76，这是一个基于数学分析（泊松分布）和行业规定一起得到的一个结论。

## 为什么要将链表中转红黑树的阈值设为8？为什么不一开始直接使用红黑树？

可能有很多人会问，既然红黑树性能这么好，为什么不一开始直接使用红黑树，而是先用链表，链表长度大于8时，才转换为红黑树。

- 因为红黑树的节点所占的空间是普通链表节点的两倍，但查找的时间复杂度低，所以只有当节点特别多时，红黑树的优点才能体现出来。至于为什么是8，是通过数据分析统计出来的一个结果，链表长度到达8的概率是很低的，综合链表和红黑树的性能优缺点考虑将大于8的链表转化为红黑树。
- 链表转化为红黑树除了链表长度大于8，还要HashMap中的数组长度大于64。也就是说如果HashMap长度小于64，链表长度大于8是不会转化为红黑树的，而是直接扩容。

## HashMap是怎么解决哈希冲突的？ \* \* \*

哈希冲突：HashMap在存储元素时会先计算key的hash值来确定存储位置，因为key的hash值计算最后有个对数组长度取余的操作，所以即使不同的key也可能计算出相同的hash值，这样就引起了hash冲突。HashMap的底层结构中的链表/红黑树就是用来解决这个问题的。

HashMap中的哈希冲突解决方式可以主要从三方面考虑（以JDK1.8为背景）

- 拉链法

HashMap中的数据结构为数组+链表/红黑树，当不同的key计算出的hash值相同时，就用链表的形式将Node结点（冲突的key及key对应的value）挂在数组后面。

- hash函数

key的hash值经过两次扰动，key的hashCode值与key的hashCode值的右移16位进行异或，然后对数组的长度取余（实际为了提高性能用的是位运算，但目的和取余一样），这样做可以让hashCode取出的高位也参与运算，进一步降低hash冲突的概率，使得数据分布更平均。

- 红黑树

在拉链法中，如果hash冲突特别严重，则会导致数组上挂的链表长度过长，性能变差，因此在链表长度大于8时，将链表转化为红黑树，可以提高遍历链表的速度。

## HashMap为什么不直接使用hashCode()处理后的哈希值直接作为table的下标？ \* \* \*

hashCode()处理后的哈希值范围太大，不可能在内存建立这么大的数组。

## 能否使用任何类作为 Map 的 key? \* \* \*

可以，但要注意以下两点：

- 如果类重写了 `equals()` 方法，也应该重写 `hashCode()` 方法。
- 最好定义 key 类是不可变的，这样 key 对应的 `hashCode()` 值可以被缓存起来，性能更好，这也是为什么 `String` 特别适合作为 `HashMap` 的 key。

## 为什么 `HashMap` 中 `String`、`Integer` 这样的包装类适合作为 Key? \* \* \*

- 这些包装类都是 `final` 修饰，是不可变性的，保证了 key 的不可更改性，不会出现放入和获取时哈希值不同的情况。
- 它们内部已经重写过 `hashCode()`, `equals()` 等方法。

## 如果使用 `Object` 作为 `HashMap` 的 Key，应该怎么办呢？ \* \*

- 重写 `hashCode()` 方法，因为需要计算 hash 值确定存储位置
- 重写 `equals()` 方法，因为需要保证 key 的唯一性。

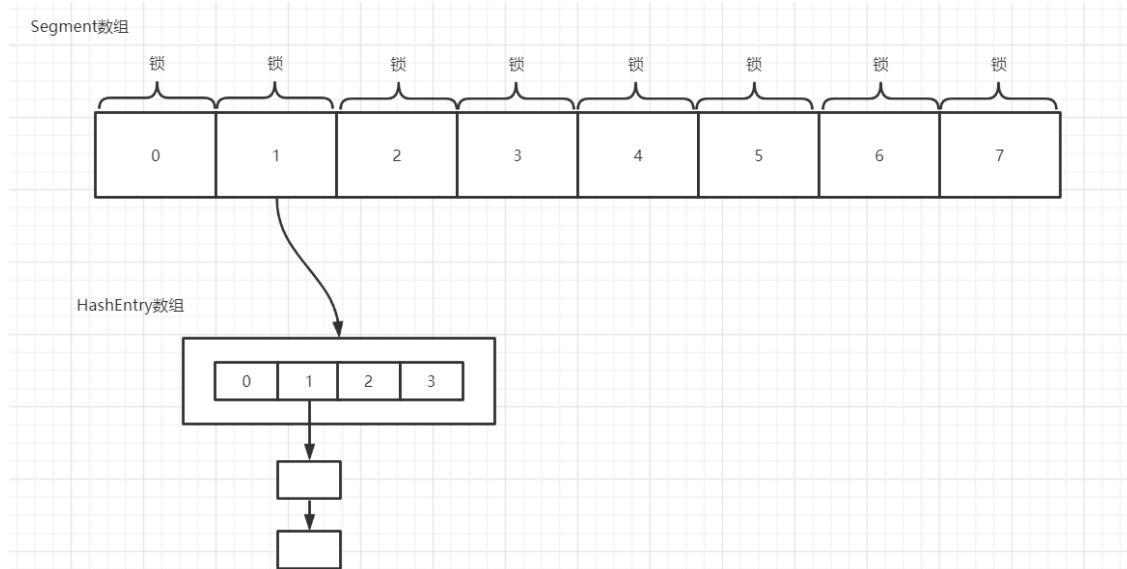
## HashMap 多线程导致死循环问题 \* \* \*

由于 JDK1.7 的 `HashMap` 遇到 hash 冲突采用的是头插法，在多线程情况下会存在死循环问题，但 JDK1.8 已经改成了尾插法，不存在这个问题了。但需要注意的是 JDK1.8 中的 `HashMap` 仍然是不安全的，在多线程情况下使用仍然会出现线程安全问题。基本上面试时说到这里既可以了，具体流程用口述是很难说清的，感兴趣的可以看这篇文章。[HASHMAP 的死循环](#)

## ConcurrentHashMap 底层具体实现知道吗？ \* \*

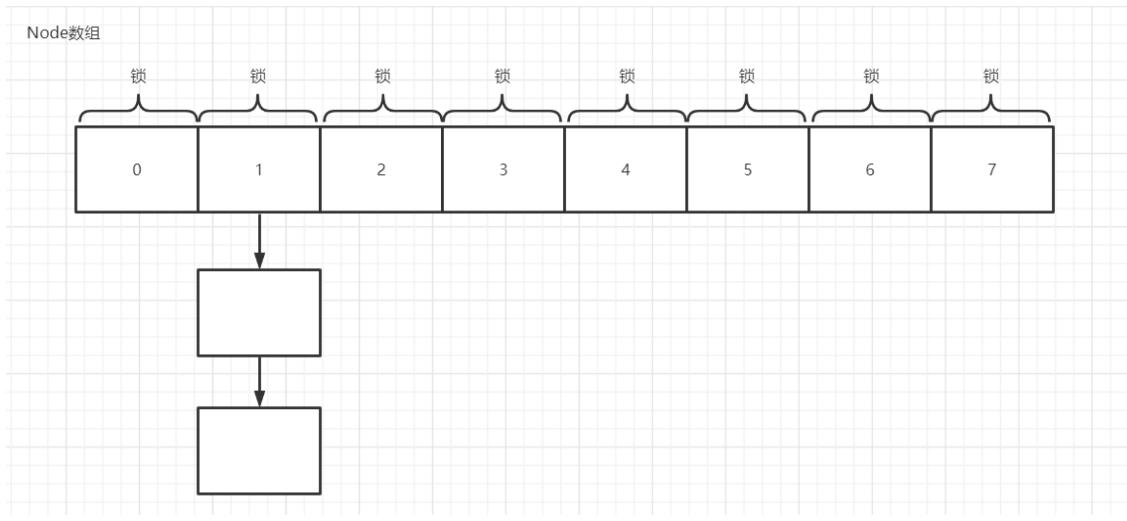
- JDK1.7

在 JDK1.7 中，`ConcurrentHashMap` 采用 `Segment` 数组 + `HashEntry` 数组的方式进行实现。`Segment` 实现了 `ReentrantLock`，所以 `Segment` 有锁的性质，`HashEntry` 用于存储键值对。一个 `ConcurrentHashMap` 包含着一个 `Segment` 数组，一个 `Segment` 包含着一个 `HashEntry` 数组，`HashEntry` 是一个链表结构，如果要获取 `HashEntry` 中的元素，要先获得 `Segment` 的锁。



- JDK1.8

在 JDK1.8 中，不再是 `Segment` + `HashEntry` 的结构了，而是和 `HashMap` 类似的结构，`Node` 数组 + 链表/红黑树，采用 `CAS` + `synchronized` 来保证线程安全。当链表长度大于 8，链表转化为红黑树。在 JDK1.8 中 `synchronized` 只锁链表或红黑树的头节点，是一种相比于 `segment` 更为细粒度的锁，锁的竞争变小，所以效率更高。

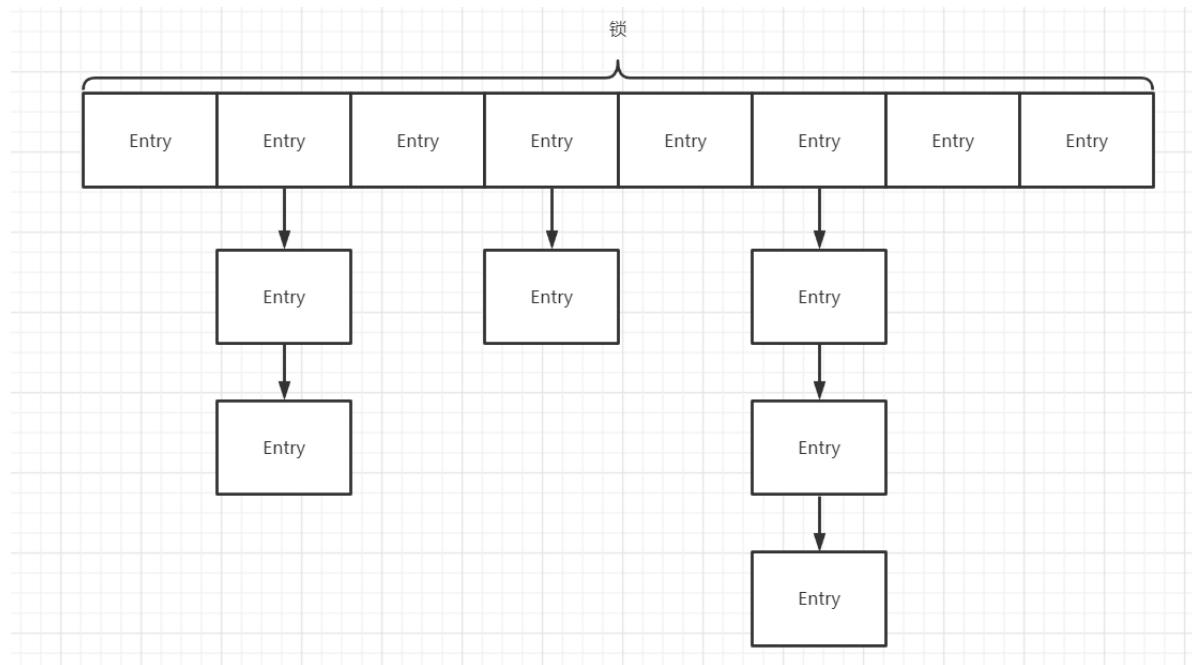


总结一下：

- JDK1.7底层是 ReentrantLock + Segment + HashEntry，JDK1.8底层是 synchronized + CAS + 链表/红黑树
- JDK1.7采用的是分段锁，同时锁住几个 HashEntry，JDK1.8锁的是Node节点，只要没有发生哈希冲突，就不会产生锁的竞争。所以JDK1.8相比于JDK1.7提供了一种粒度更小的锁，减少了锁的竞争，提高了 ConcurrentHashMap 的并发能力。

## HashTable的底层实现知道吗？ \* \*

HashTable 的底层数据结构是数组+链表，链表主要是为了解决哈希冲突，并且整个数组都是 synchronized 修饰的，所以 HashTable 是线程安全的，但锁的粒度太大，锁的竞争非常激烈，效率很低。



## HashMap、ConcurrentHashMap及Hashtable的区别 \* \* \*

	<b>HashMap(JDK1.8)</b>	<b>ConcurrentHashMap(JDK1.8)</b>	<b>Hashtable</b>
底层实现	数组+链表/红黑树	数组+链表/红黑树	数组+链表
线程安全	不安全	安全( <code>synchronized</code> 修饰Node 节点)	安全 ( <code>synchronized</code> 修饰整个表)
效率	高	较高	低
扩容	初始16, 每次扩容成 $2n$	初始16, 每次扩容成 $2n$	初始11, 每次扩容成 $2n+1$
是否支持 Null key和 Null Value	可以有一个Null key, Null Value多个	不支持	不支持

## Java集合的常用方法 \* \*

这些常用方法是需要背下来的，虽然面试用不上，但是笔试或者面试写算法题时会经常用到。

### Collection常用方法

方法	
<code>booleans add(E e)</code>	在集合末尾添加元素
<code>boolean remove(Object o)</code>	若本类集中有值与o的值相等的元素，移除该元素并返回true
<code>void clear()</code>	清除本类中所有元素
<code>boolean contains(Object o)</code>	判断集合中是否包含该元素
<code>boolean isEmpty()</code>	判断集合是否为空
<code>int size()</code>	返回集合中元素的个数
<code>boolean addAll(Collection c)</code>	将一个集合中c中的所有元素添加到另一个集合中
<code>Object[] toArray()</code>	返回一个包含本集所有元素的数组，数组类型为Object[]
<code>boolean equals(Object c)``</code>	判断元素是否相等
<code>int hashCode()</code>	返回元素的hash值

### List特有方法

方法	
<code>void add(int index, Object obj)</code>	在指定位置添加元素
<code>Object remove(int index)</code>	删除指定元素并返回
<code>Object set(int index, Object obj)</code>	把指定索引位置的元素更改为指定值并返回修改前的值
<code>int indexOf(Object o)</code>	返回指定元素在集合中第一次出现的索引
<code>Object get(int index)</code>	返回指定位置的元素
<code>List subList(int fromIndex, int toIndex)</code>	截取集合(左闭右开)

## LinkedList特有方法

方法	
<code>addFirst()</code>	在头部添加元素
<code>addLast()</code>	在尾部添加元素
<code>removeFirst()</code>	在头部删除元素
<code>removeLast()</code>	在尾部删除元素
<code>getFirst()</code>	获取头部元素
<code>getLast()</code>	获取尾部元素

## Map

方法	
<code>void clear()</code>	清除集合内的元素
<code>boolean containsKey(Object key)</code>	查询Map中是否包含指定key,如果包含则返回true
<code>Set entrySet()</code>	返回Map中所包含的键值对所组成的Set集合, 每个集合元素都是Map.Entry的对象
<code>Object get(Object key)</code>	返回key指定的value,若Map中不包含key返回null
<code>boolean isEmpty()</code>	查询Map是否为空, 若为空返回true
<code>Set keySet()</code>	返回Map中所有key所组成的集合
<code>Object put(Object key, Object value)</code>	添加一个键值对, 如果已有一个相同的key,则新的键值对会覆盖旧的键值对,返回值为覆盖前的value值, 否则为null
<code>void putAll(Map m)</code>	将制定Map中的键值对复制到Map中
<code>Object remove(Object key)</code>	删除指定key所对应的键值对, 返回所关联的value,如果key不存在返回null
<code>int size()</code>	返回Map里面的键值对的个数
<code>Collection values()</code>	返回Map里所有values所组成的Collection
<code>boolean containsValue (Object value)</code>	判断映像中是否存在值 value

## Stack

方法	
<code>boolean empty()</code>	测试堆栈是否为空。
<code>E peek()</code>	查看堆栈顶部的对象, 但不从堆栈中移除它。
<code>E pop()</code>	移除堆栈顶部的对象, 并作为此函数的值返回该对象。
<code>E push(E item)</code>	把项压入堆栈顶部。
<code>int search(Object o)</code>	返回对象在堆栈中的位置, 以 1 为基数。

## Queue

方法	
boolean add(E e)	将指定元素插入到队列的尾部(队列满了话，会抛出异常)
boolean offer(E e)	将指定元素插入此队列的尾部(队列满了话，会返回false)
E remove()	返回取队列头部的元素，并删除该元素(如果队列为空，则抛出异常)
E poll()	返回队列头部的元素，并删除该元素(如果队列为空，则返回null)
E element()	返回队列头部的元素，不删除该元素(如果队列为空，则抛出异常)
E peek()	返回队列头部的元素，不删除该元素(如果队列为空，则返回null)

关注公众号“路人zhang”，获取更多面试技巧及资料

关注知乎“路人zhang”，聊聊码农那些事



## 计算机网络

### 什么是网络协议，为什么要对网络协议分层 \* \*

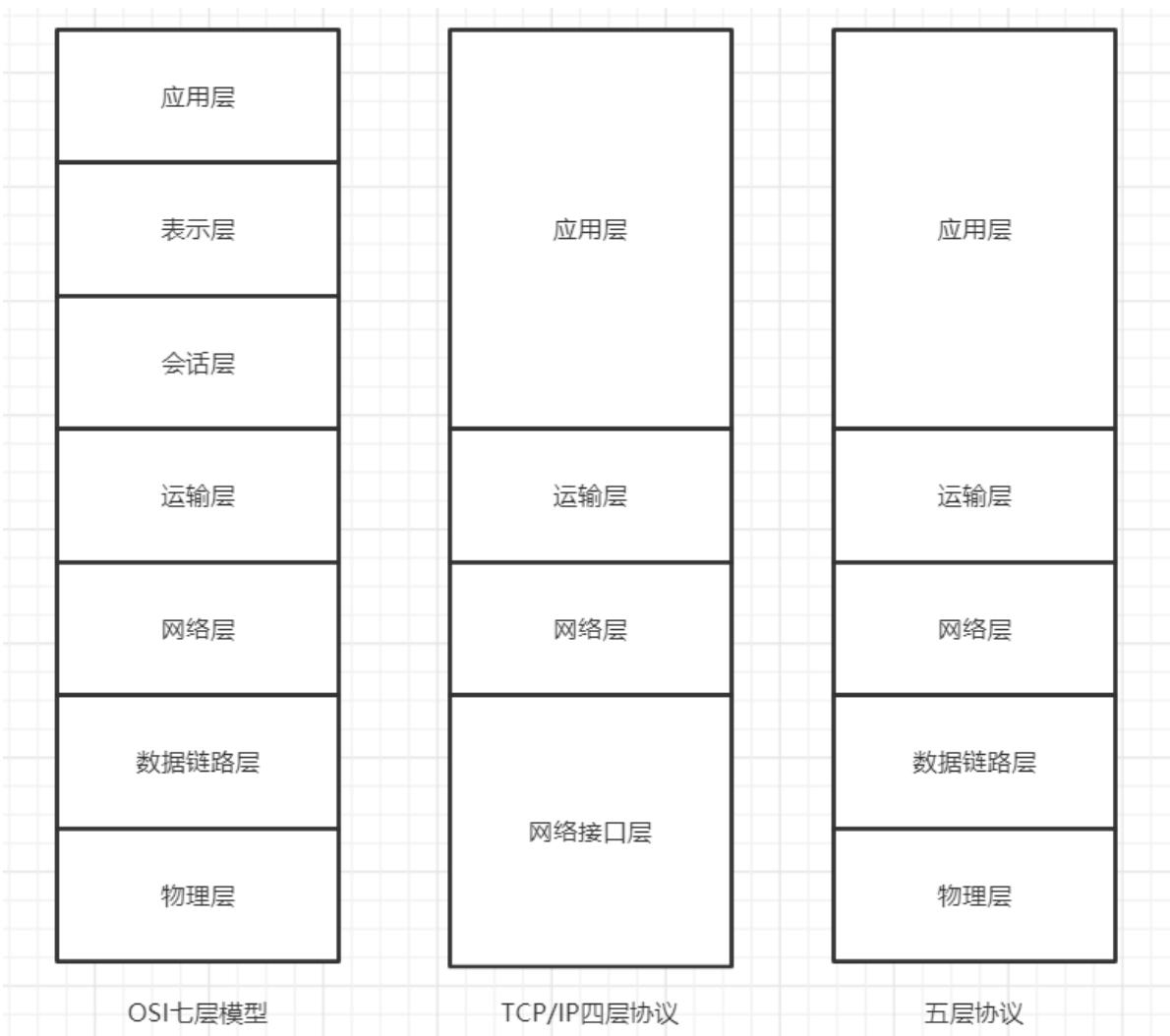
网络协议是计算机在通信过程中要遵循的一些约定好的规则。

网络分层的原因：

- 易于实现和维护，因为各层之间是独立的，层与层之间不会收到影响。
- 有利于标准化的制定

### 计算机网络的各层协议及作用 \* \* \*

计算机网络体系可以大致分为一下三种，七层模型、五层模型和TCP/IP四层模型，一般面试能流畅回答出五层模型就可以了，表示层和会话层被问到的不多。



- **应用层**

应用层的任务是通过应用进程之间的交互来完成特定的网络作用，常见的应用层协议有域名系统 DNS，HTTP协议等。

- **表示层**

表示层的主要作用是数据的表示、安全、压缩。可确保一个系统的应用层所发送的信息可以被另一个系统的应用层读取。

- **会话层**

会话层的主要作用是建立通信链接，保持会话过程通信链接的畅通，同步两个节点之间的对话，决定通信是否被中断以及通信中断时决定从何处重新发送。。

- **传输层**

传输层的主要作用是负责向两台主机进程之间的通信提供数据传输服务。传输层的协议主要有传输控制协议TCP和用户数据协议UDP。

- **网络层**

网络层的主要作用是选择合适的网间路由和交换结点，确保数据及时送达。常见的协议有IP协议。

- **数据链路层**

数据链路层的作用是在物理层提供比特流服务的基础上，建立相邻结点之间的数据链路，通过差错控制提供数据帧（Frame）在信道上无差错的传输，并进行各电路上的动作系列。常见的协议有SDLC、HDLC、PPP等。

- **物理层**

物理层的主要作用是实现相邻计算机结点之间比特流的透明传输，并尽量屏蔽掉具体传输介质和物理设备的差异。

## URI和URL的区别 \*

- URI(Uniform Resource Identifier): 中文全称为统一资源标志符，主要作用是唯一标识一个资源。
- URL(Uniform Resource Location): 中文全称为统一资源定位符，主要作用是提供资源的路径。

有个经典的比喻是URI像是身份证，可以唯一标识一个人，而URL更像一个住址，可以通过URL找到这个人。

## DNS的工作流程 \* \* \*

DNS的定义：DNS的全称是domain name system，即域名系统。DNS是因特网上作为域名和IP地址相互映射的一个分布式数据库，能够使用户更方便的去访问互联网而不用去记住能够被机器直接读取的IP地址。比如大家访问百度，更多地肯定是访问[www.baidu.com](http://www.baidu.com)，而不是访问112.80.248.74，因为这几乎无规则的IP地址实在太难记了。DNS要做的就是将[www.baidu.com](http://www.baidu.com)解析成112.80.248.74。

### DNS是集群式的工作方式还是单点式的，为什么？

答案是集群式的，很容易想到的一个方案就是只用一个DNS服务器，包含了所有域名和IP地址的映射。尽管这种设计方式看起来很简单，但是缺点显而易见，如果这个唯一的DNS服务器出了故障，那么就全完了，因特网就几乎崩了。为了避免这种情况出现，DNS系统采用的是分布式的层次数据数据库模式，还有缓存的机制也能解决这种问题。

### DNS的工作流程

主机向本地域名服务器的查询一般是采用递归查询，而本地域名服务器向根域名的查询一般是采用迭代查询。

递归查询主机向本地域名发送查询请求报文，而本地域名服务器不知道该域名对应的IP地址时，本地域名会继续向根域名发送查询请求报文，不是通知主机自己向根域名发送查询请求报文。迭代查询是，本地域名服务器向根域名发出查询请求报文后，根域名不会继续向顶级域名服务器发送查询请求报文，而是通知本地域名服务器向顶级域名发送查询请求报文。

简单来说，递归查询就是，小明问了小红一个问题，小红不知道，但小红是个热心肠，小红就去问小王了，小王把答案告诉小红后，小红又去把答案告诉了小明。迭代查询就是，小明问了小红一个问题，小红也不知道，然后小红让小明去问小王，小明又去问小王了，小王把答案告诉了小明。

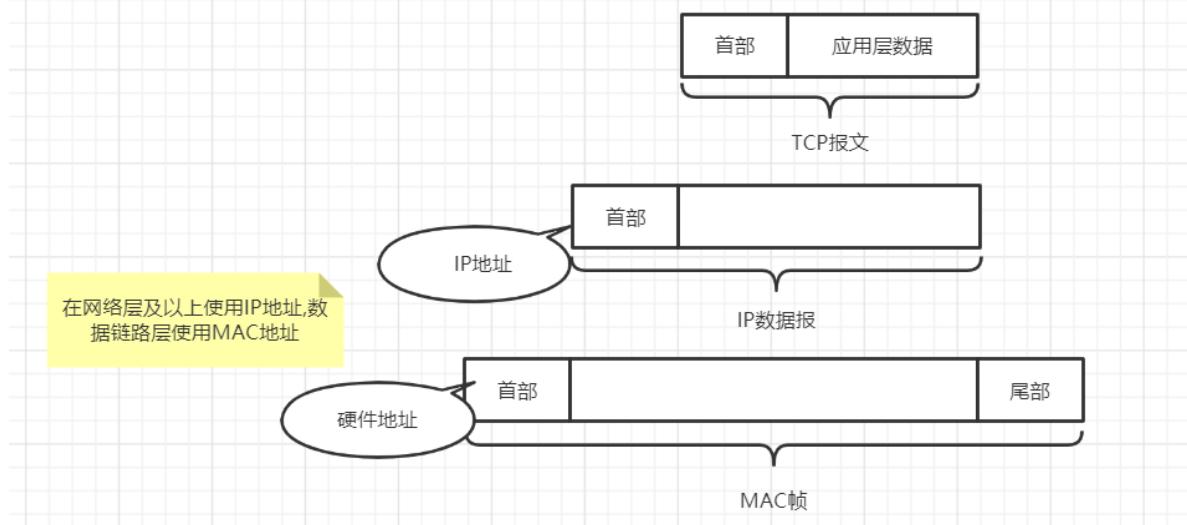
1. 在浏览器中输入[www.baidu.com](http://www.baidu.com)域名，操作系统会先检查自己本地的hosts文件是否有这个域名的映射关系，如果有，就先调用这个IP地址映射，完成域名解析。
2. 如果hosts文件中没有，则查询本地DNS解析器缓存，如果有，则完成地址解析。
3. 如果本地DNS解析器缓存中没有，则去查找本地DNS服务器，如果查到，完成解析。
4. 如果没有，则本地服务器会向根域名服务器发起查询请求。根域名服务器会告诉本地域名服务器去查询哪个顶级域名服务器。
5. 本地域名服务器向顶级域名服务器发起查询请求，顶级域名服务器会告诉本地域名服务器去查找哪个权限域名服务器。
6. 本地域名服务器向权限域名服务器发起查询请求，权限域名服务器告诉本地域名服务器[www.baidu.com](http://www.baidu.com)所对应的IP地址。
7. 本地域名服务器告诉主机[www.baidu.com](http://www.baidu.com)所对应的IP地址。

## 了解ARP协议吗？ \*

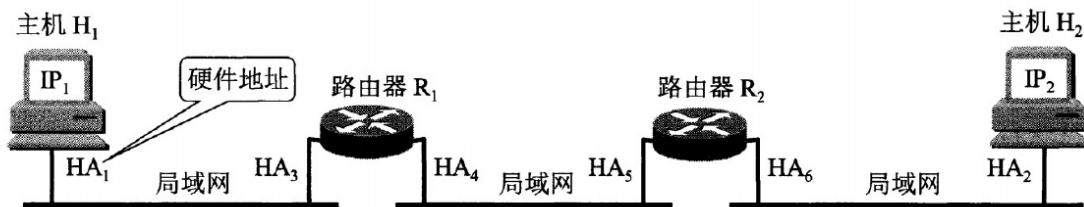
ARP协议属于网络层的协议，主要作用是实现从IP地址转换为MAC地址。在每个主机或者路由器中都建有一个ARP缓存表，表中有IP地址及IP地址对应的MAC地址。先来看一下什么时IP地址和MAC地址。

- IP地址：IP地址是指互联网协议地址，IP地址是IP协议提供的一种统一的地址格式，它为互联网上的每一个网络和每一台主机分配一个逻辑地址，以此来屏蔽物理地址的差异。
- MAC地址：MAC地址又称物理地址，由网络设备制造商生产时写在硬件内部，不可更改，并且每一个以太网设备的MAC地址都是唯一的。

数据在传输过程中，会先从高层传到底层，然后在通信链路上传输。从下图可以看到TCP报文在网络层会被封装成IP数据报，在数据链路层被封装成MAC帧，然后在通信链路中传输。在网络层使用的是IP地址，在数据链路层使用的是MAC地址。MAC帧在传送时的源地址和目的地址使用的都是MAC地址，在通信链路上的主机或路由器也都是根据MAC帧首部的MAC地址接收MAC帧。并且在数据链路层是看不到IP地址的，只有当数据传到网络层时去掉MAC帧的首部和尾部时才能在IP数据报的首部中找到源IP地址和目的地址。



网络层实现的是主机之间的通信，而链路层实现的是链路之间的通信，所以从下图可以看出，在数据传输过程中，IP数据报的源地址(IP1)和目的地址(IP2)是一直不变的，而MAC地址(硬件地址)却一直随着链路的改变而改变。



ARP的工作流程(面试时问ARP协议主要说这个就可以了):

1. 在局域网内，主机A要向主机B发送IP数据报时，首先会在主机A的ARP缓存表中查找是否有IP地址及其对应的MAC地址，如果有，则将MAC地址写入到MAC帧的头部，并通过局域网将该MAC帧发送到MAC地址所在的主机B。
2. 如果主机A的ARP缓存表中没有主机B的IP地址及所对应的MAC地址，主机A会在局域网内广播发送一个ARP请求分组。局域网内的所有主机都会收到这个ARP请求分组。
3. 主机B在看到主机A发送的ARP请求分组中有自己的IP地址，会像主机A以单播的方式发送一个带有自己MAC地址的响应分组。
4. 主机A收到主机B的ARP响应分组后，会在ARP缓存表中写入主机B的IP地址及其IP地址对应的MAC地址。
5. 如果主机A和主机B不在同一个局域网内，即使知道主机B的MAC地址也是不能直接通信的，必须通过路由器转发到主机B的局域网才可以通过主机B的MAC地址找到主机B。并且主机A和主机B已经可以通信的情况下，主机A的ARP缓存表中存的并不是主机B的IP地址及主机B的MAC地址，而是主机B的IP地址及该通信链路上的下一跳路由器的MAC地址。这就是上图中的源IP地址和目的IP地址一直不变，而MAC地址却随着链路的不同而改变。
6. 如果主机A和主机B不在同一个局域网，参考上图中的主机H<sub>1</sub>和主机H<sub>2</sub>，这时主机H<sub>1</sub>需要先广播找到路由器R<sub>1</sub>的MAC地址，再由R<sub>1</sub>广播找到路由器R<sub>2</sub>的MAC地址，最后R<sub>2</sub>广播找到主机H<sub>2</sub>的MAC地址，建立起通信链路。

## 有了IP地址，为什么还要用MAC地址？ \* \*

简单来说，标识网络中的一台计算机，比较常用的就是IP地址和MAC地址，但计算机的IP地址可由用户自行更改，管理起来相对困难，而MAC地址不可更改，所以一般会把IP地址和MAC地址组合起来使用。具体是如何组合使用的在上面的ARP协议中已经讲的很清楚了。

那只用MAC地址不用IP地址可不可以呢？其实也是不行的，因为在最早就是MAC地址先出现的，并且当时并不用IP地址，只用MAC地址，后来随着网络中的设备越来越多，整个路由过程越来越复杂，便出现了子网的概念。对于目的地址在其他子网的数据包，路由只需要将数据包送到那个子网即可，这个过程就是上面说的ARP协议。

那为什么要用IP地址呢？是因为IP地址是和地域相关的，对于同一个子网上的设备，IP地址的前缀都是一样的，这样路由器通过IP地址的前缀就知道设备在哪个子网上了，而只用MAC地址的话，路由器则需要记住每个MAC地址在哪个子网，这需要路由器有极大的存储空间，是无法实现的。

IP地址可以比作为地址，MAC地址为收件人，在一次通信过程中，两者是缺一不可的。

## 说一下ping的过程 \* \*

ping是ICMP(网际控制报文协议)中的一个重要应用，ICMP是网络层的协议。ping的作用是测试两个主机的连通性。

ping的工作过程：

1. 向目的主机发送多个ICMP回送请求报文
2. 根据目的主机返回的回送报文的时间和成功响应的次数估算出数据包往返时间及丢包率。

## 路由器和交换机的区别？ \*

	所属网络模型的层级	功能
路由器	网络层	识别IP地址并根据IP地址转发数据包，维护数据表并基于数据表进行最佳路径选择
交换机	数据链路层	识别MAC地址并根据MAC地址转发数据帧

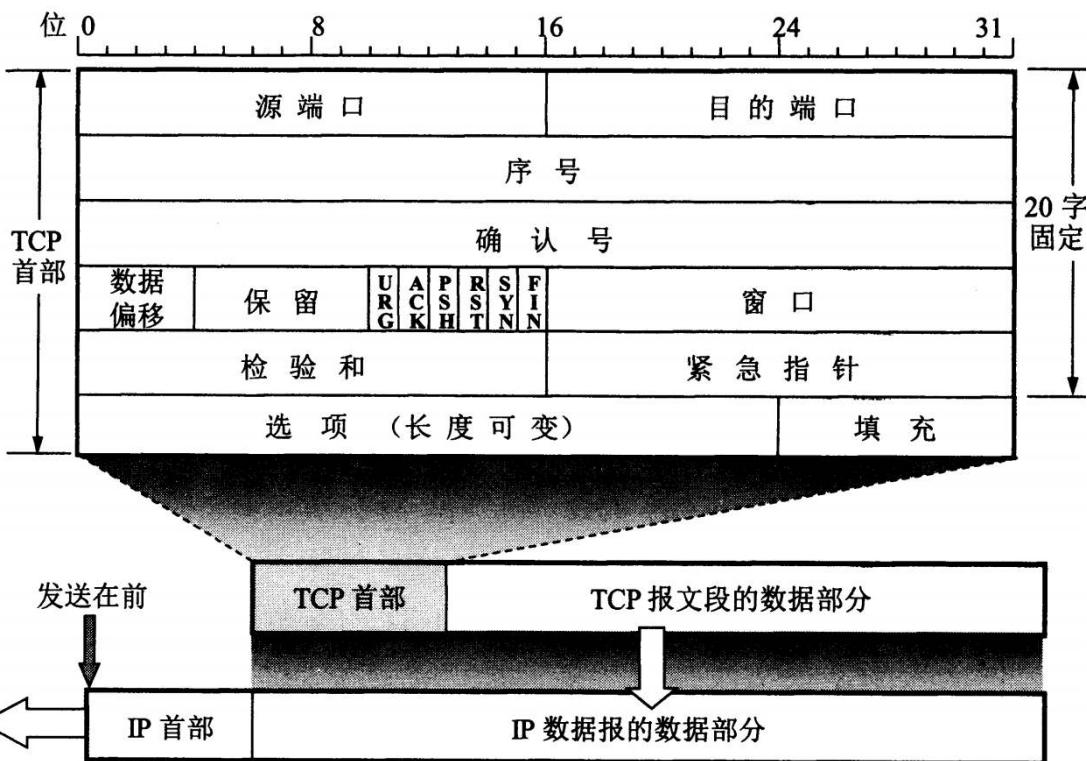
## TCP与UDP有什么区别 \* \* \*

	是否面向连接	可靠性	传输形式	传输效率	消耗资源	应用场景	首部字节
TCP	面向连接	可靠	字节流	慢	多	文件/邮件传输	20~60
UDP	无连接	不可靠	数据报文段	快	少	视频/语音传输	8

有时候面试还会问到TCP的首部都包含什么

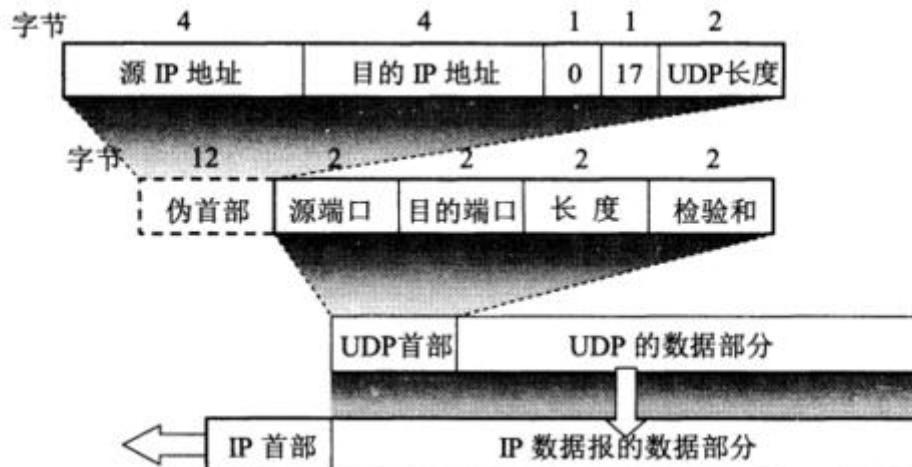
- TCP首部(图片来源于网络)：

前20个字节是固定的，后面有 $4n$ 个字节是根据需求增加的选项，所以TCP首部最小长度为20字节。



- UDP首部(图片来源于网络):

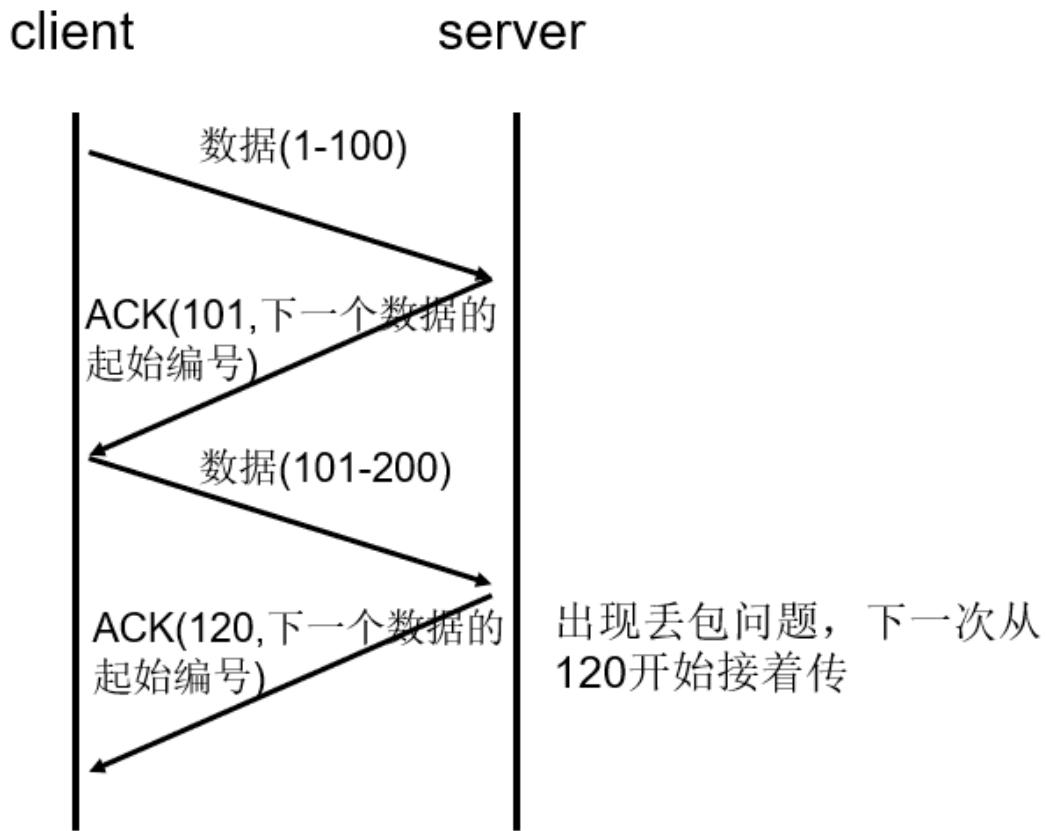
UDP的首部只有8个字节，源端口号、目的端口号、长度和校验和各两个字节。



## TCP协议如何保证可靠传输 \* \* \*

主要有校验和、序列号、超时重传、流量控制及拥塞避免等几种方法。

- 校验和：在发送端和接收端分别计算数据的校验和，如果两者不一致，则说明数据在传输过程中出现了差错，TCP将丢弃和不确认此报文段。
- 序列号：TCP会对每一个发送的字节进行编号，接收方接到数据后，会对发送方发送确认应答(ACK报文)，并且这个ACK报文中带有相应的确认编号，告诉发送方，下一次发送的数据从编号多少开始发。如果发送方发送相同的数据，接收端也可以通过序列号判断出，直接将数据丢弃。如果

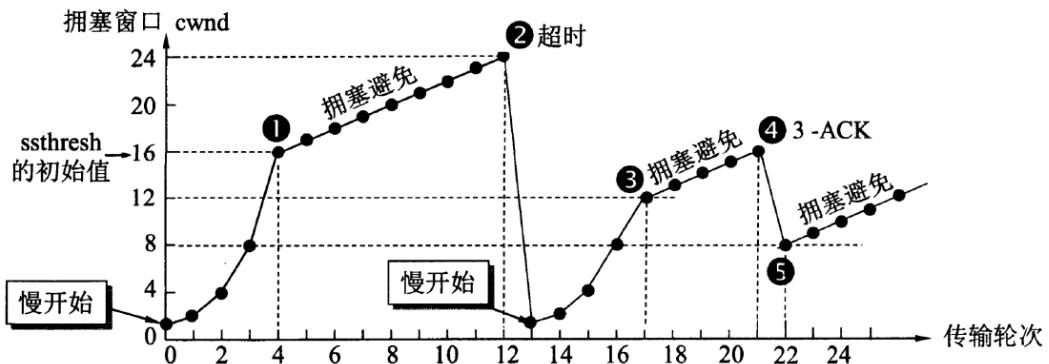


- 超时重传：在上面说了序列号的作用，但如果发送方在发送数据后一段时间内（可以设置重传计时器规定这段时间）没有收到确认序号ACK，那么发送方就会重新发送数据。

这里发送方没有收到ACK可以分两种情况，如果是发送方发送的数据包丢失了，接收方收到发送方重新发送的数据包后会马上给发送方发送ACK；如果是接收方之前接收到了发送方发送的数据包，而返回给发送方的ACK丢失了，这种情况，发送方重传后，接收方会直接丢弃发送方冲重传的数据包，然后再次发送ACK响应报文。

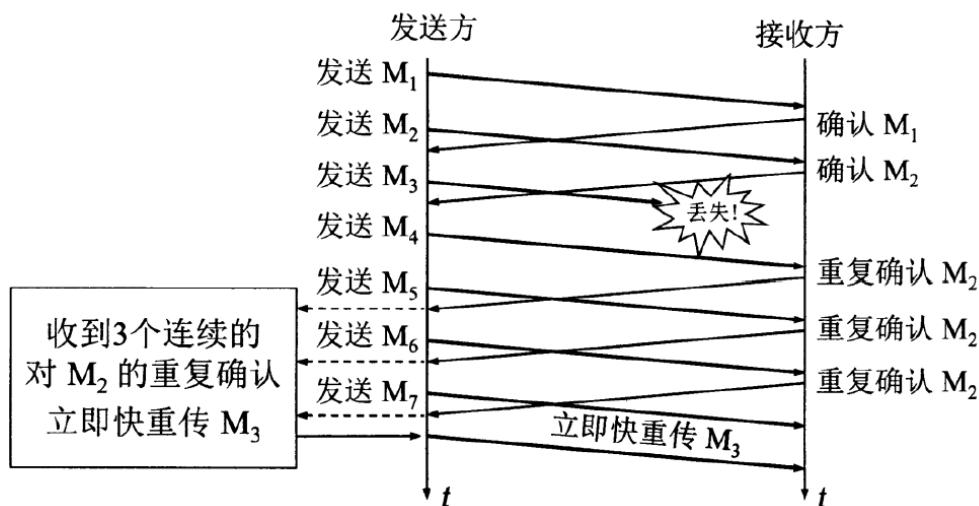
如果数据被重发之后还是没有收到接收方的确认应答，则进行再次发送。此时，等待确认应答的时间将会以2倍、4倍的指数函数延长，直到最后关闭连接。

- 流量控制：如果发送端发送的数据太快，接收端来不及接收就会出现丢包问题。为了解决这个问题，TCP协议利用了滑动窗口进行了流量控制。在TCP首部有一个16位字段大小的窗口，窗口的大小就是接收端接收数据缓冲区的剩余大小。接收端会在收到数据包后发送ACK报文时，将自己的窗口大小填入ACK中，发送方会根据ACK报文中的窗口大小进而控制发送速度。如果窗口大小为零，发送方会停止发送数据。
- 拥塞控制：如果网络出现拥塞，则会产生丢包等问题，这时发送方会将丢失的数据包继续重传，网络拥塞会更加严重，所以在网络出现拥塞时应注意控制发送方的发送数据，降低整个网络的拥塞程度。拥塞控制主要有四部分组成：慢开始、拥塞避免、快重传、快恢复，如下图(图片来源于网络)。



这里的发送方会维护一个拥塞窗口的状态变量，它和流量控制的滑动窗口是不一样的，滑动窗口是根据接收方数据缓冲区大小确定的，而拥塞窗口是根据网络的拥塞情况动态确定的，一般来说发送方真实的发送窗口为滑动窗口和拥塞窗口中的最小值。

1. 慢开始：为了避免一开始发送大量的数据而产生网络阻塞，会先初始化cwnd为1，当收到ACK后到下一个传输轮次，cwnd为2，以此类推成指数形式增长。
2. 拥塞避免：因为cwnd的数量在慢开始是指数增长的，为了防止cwnd数量过大而导致网络阻塞，会设置一个慢开始的门限值ssthresh，当cwnd $\geq$ ssthresh时，进入到拥塞避免阶段，cwnd每个传输轮次加1。但网络出现超时，会将门限值ssthresh变为出现超时cwnd数值的一半，cwnd重新设置为1，如上图，在第12轮出现超时后，cwnd变为1，ssthresh变为12。
3. 快重传：在网络中如果出现超时或者阻塞，则按慢开始和拥塞避免算法进行调整。但如果只是丢失某一个报文段，如下图(图片来源于网络)，则使用快重传算法。



从上图可知，接收方正确地接收到M1和M2，而M3丢失，由于没有接收到M3，在接收方收到M5、M6和M7时，并不会进行确认，也就是不会发送ACK。这时根据前面说的保证TCP可靠性传输中的序列号的作用，接收方这时不会接收M5，M6，M7，接收方可以什么都不会，因为发送方长时间未收到M3的确认报文，会对M3进行重传。除了这样，接收方也可以重复发送M2的确认报文，这样发送端长时间未收到M3的确认报文也会继续发送M3报文。

**但是根据快重传算法，要求在这种情况下，需要快速向发送端发送M2的确认报文，在发送方收到三个M2的确认报文后，无需等待重传计时器所设置的时间，可直接进行M3的重传，这就是快重传。(面试时说这一句就够了，前面是帮助理解)**

4. 快恢复：从上上图圈4可以看到，当发送收到三个重复的ACK，会进行快重传和快恢复。快恢复是指将ssthresh设置为发生快重传时的cwnd数量的一半，而cwnd不是设置为1而是设置为门限值ssthresh，并开始拥塞避免阶段。

# TCP的三次握手及四次挥手 \* \* \*

## 必考题

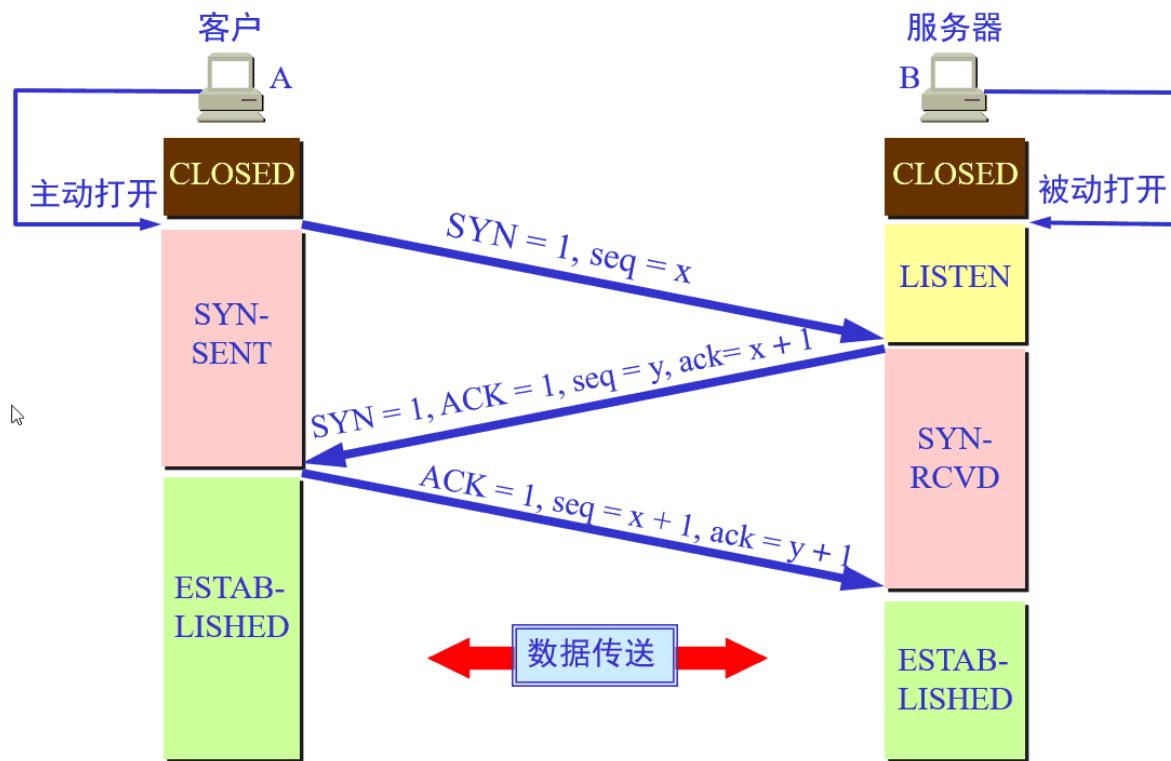
在介绍三次握手和四次挥手之前，先介绍一下TCP头部的一些常用字段。

- 序号：seq，占32位，用来标识从发送端到接收端发送的字节流。
- 确认号：ack，占32位，只有ACK标志位为1时，确认序号字段才有效， $ack=seq+1$ 。
- 标志位：
  - SYN：发起一个新连接。
  - FIN：释放一个连接。
  - ACK：确认序号有效。

## 三次握手

三次握手的本质就是确定发送端和接收端具备收发信息的能力，在能流畅描述三次握手的流程及其中的字段含义作用的同时还需要记住每次握手时接收端和发送端的状态。这个比较容易忽略。

先看一张很经典的图（图片来源于网络），发送端有CLOSED、SYN-SENT、ESTABLISHED三种状态，接收端有CLOSED、LISTEN、SYN-RCVD、ESTABLISHED四种状态。



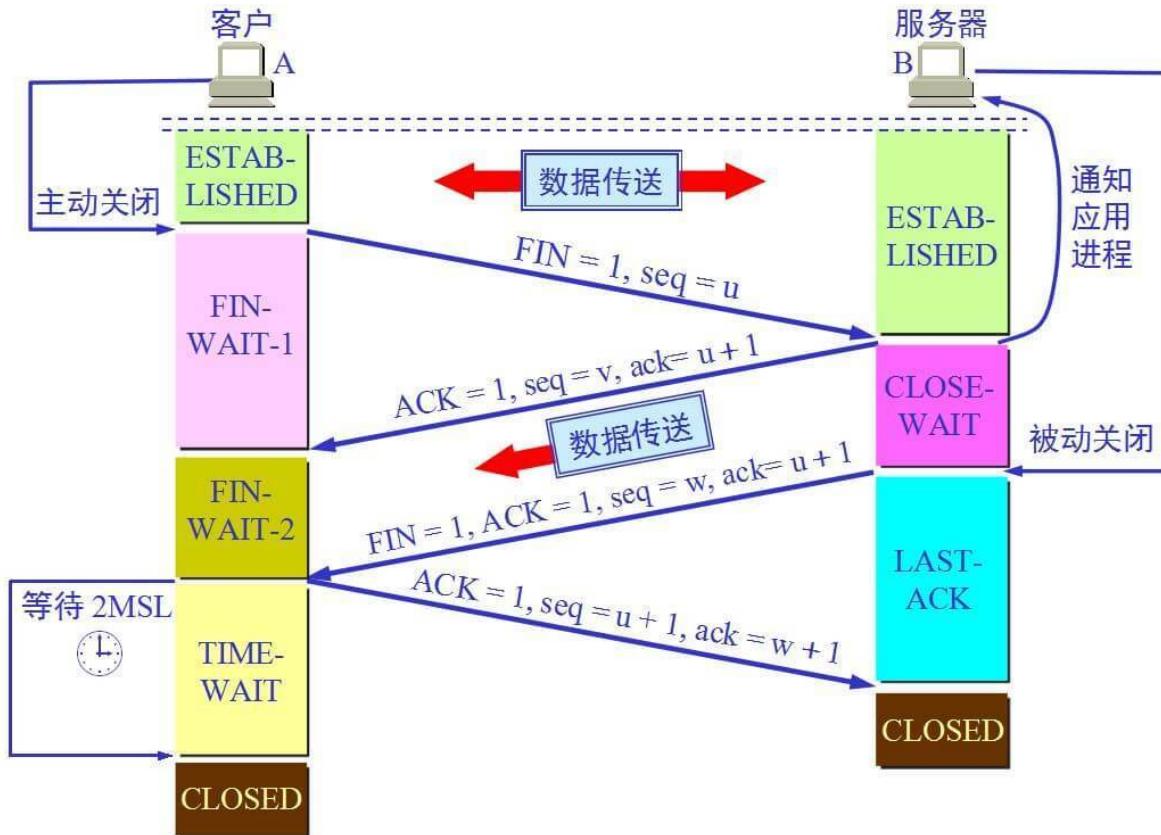
假设发送端为客户端，接收端为服务端。开始时客户端和服务端的状态都是CLOSE。

- 第一次握手：客户端向服务端发起建立连接请求，客户端会随机生成一个起始序列号x，客户端向服务端发送的字段中包含标志位SYN=1，序列号seq=100。第一次握手前客户端的状态为CLOSE，第一次握手后客户端的状态为SYN-SENT。此时服务端的状态为LISTEN
- 第二次握手：服务端在收到客户端发来的报文后，会随机生成一个服务端的起始序列号y，然后给客户端回复一段报文，其中包括标志位SYN=1，ACK=1，序列号seq=y，确认号ack=x+1。第二次握手前服务端的状态为LISTEN，第二次握手后服务端的状态为SYN-RCVD，此时客户端的状态为SYN-SENT。（其中SYN=1表示要和客户端建立一个连接，ACK=1表示确认序号有效）
- 第三次握手：客户端收到服务端发来的报文后，会再向服务端发送报文，其中包括标志位ACK=1，序列号seq=x+1，确认号ack=y+1。第三次握手前客户端的状态为SYN-SENT，第三次握手后客户端和服务端的状态都为ESTABLISHED。

需要注意的一点是，第一次握手，客户端向服务端发起建立连接报文，会占一个序列号。但是第三次握手，同样是客户端向服务端发送报文，这次却不占序列号，所以建立连接后，客户端向服务端发送的第一个数据的序列号为 $x+1$ 。

## 四次挥手

和三次握手一样，先看一张非常经典的图（图片来源于网络），客户端在四次挥手过程中有ESTABLISHED、FIN-WAIT-1、FIN-WAIT-2、TIME-WAIT、CLOSED等五个状态，服务端有ESTABLISHED、CLOSE-WAIT、LAST-ACK、CLOSED等四种状态。最好记住每次挥手时服务端和客户端的状态。



假设客户端首先发起的断开连接请求

- 第一次挥手：客户端向服务端发送的数据完成后，向服务端发起释放连接报文，报文包含标志位FIN=1，序列号seq=u。此时客户端只能接收数据，不能向服务端发送数据。
- 第二次挥手：服务端收到客户端的释放连接报文后，向客户端发送确认报文，包含标志位ACK=1，序列号seq=v，确认号ack=u+1。此时客户端到服务端的连接已经释放掉，客户端不能像服务端发送数据，服务端也不能向客户端发送数据。但服务端到客户端的单向连接还能正常传输数据。
- 第三次挥手：服务端发送完数据后向客户端发出连接释放报文，报文包含标志位FIN=1，标志位ACK=1，序列号seq=w，确认号ack=u+1。
- 第四次挥手：客户端收到服务端发送的释放连接请求，向服务端发送确认报文，包含标志位ACK=1，序列号seq=u+1，确认号ack=w+1。

## 为什么TCP连接的时候是3次？两次是否可以？

不可以，主要从以下两方面考虑（假设客户端是首先发起连接请求）：

- 假设建立TCP连接仅需要两次握手，那么如果第二次握手时，服务端返回给客户端的确认报文丢失了，客户端这边认为服务端没有和他建立连接，而服务端却以为已经和客户端建立了连接，并且可能向服务端已经开始向客户端发送数据，但客户端并不会接收这些数据，浪费了资源。如果是三次握手，不会出现双方连接还未完全建立成功就开始发送数据的情况。
- 如果服务端接收到了一个早已失效的来自客户端的连接请求报文，会向客户端发送确认报文同意建立TCP连接。但因为客户端并不需要向服务端发送数据，所以此次TCP连接没有意义并且浪费了资源。

源。

## 为什么TCP连接的时候是3次，关闭的时候却是4次？

因为需要确保通信双方都能通知对方释放连接，假设客服端发送完数据向服务端发送释放连接请求，当客服端并不知道，服务端是否已经发送完数据，所以此次断开的是客服端到服务端的单向连接，服务端返回给客户端确认报文后，服务端还能继续单向给客户端发送数据。当服务端发送完数据后还需要向客户端发送释放连接请求，客户端返回确认报文，TCP连接彻底关闭。所以断开TCP连接需要客户端和服务端分别通知对方并分别收到确认报文，一共需要四次。

## TIME\_WAIT和CLOSE\_WAIT的区别在哪？

默认客户端首先发起断开连接请求

- 从上图可以看出，CLOSE\_WAIT是被动关闭形成的，当客户端发送FIN报文，服务端返回ACK报文后进入CLOSE\_WAIT。
- TIME\_WAIT是主动关闭形成的，当第四次挥手完成后，客户端进入TIME\_WAIT状态。

## 为什么客户端发出第四次挥手的确认报文后要等2MSL的时间才能释放TCP连接？

MSL的意思是报文的最长寿命，可以从两方面考虑：

- 客户端发送第四次挥手中的报文后，再经过2MSL，可使本次TCP连接中的所有报文全部消失，不会出现在下一个TCP连接中。
- 考虑丢包问题，如果第四挥手发送的报文在传输过程中丢失了，那么服务端没收到确认ack报文就会重发第三次挥手的报文。如果客户端发送完第四次挥手的确认报文后直接关闭，而这次报文又恰好丢失，则会造成服务端无法正常关闭。

## 如果已经建立了连接，但是客户端突然出现故障了怎么办？

如果TCP连接已经建立，在通信过程中，客户端突然故障，那么服务端不会一直等下去，过一段时间就关闭连接了。具体原理是TCP有一个保活机制，主要用在服务器端，用于检测已建立TCP链接的客户端的状态，防止因客户端崩溃或者客户端网络不可达，而服务器端一直保持该TCP链接，占用服务器端的大量资源(因为Linux系统中可以创建的总TCP链接数是有限制的)。

保活机制原理：设置TCP保活机制的保活时间keepIdle，即在TCP链接超过该时间没有任何数据交互时，发送保活探测报文；设置保活探测报文的发送时间间隔keepInterval；设置保活探测报文的总发送次数keepCount。如果在keepCount次的保活探测报文均没有收到客户端的回应，则服务器端即关闭与客户端的TCP链接。

具体细节请看这篇博客[TCP通信过程中异常情况整理](#)。

## HTTP 与 HTTPS 的区别 \* \* \*

	HTTP	HTTPS
端口	80	443
安全性	无加密，安全性较差	有加密机制，安全性较高
资源消耗	较少	由于加密处理，资源消耗更多
是否需要证书	不需要	需要
协议	运行在TCP协议之上	运行在SSL协议之上，SSL运行在TCP协议之上

# 什么是对称加密与非对称加密 \* \*

- 对称加密

对称加密指加密和解密使用同一密钥，优点是运算速度快，缺点是如何安全将密钥传输给另一方。常见的对称加密算法有DES、AES等等。

- 非对称加密

非对称加密指的是加密和解密使用不同的密钥，一把公开的公钥，一把私有的私钥。公钥加密的信息只有私钥才能解密，私钥加密的信息只有公钥才能解密。优点解决了对称加密中存在的问题。缺点是运算速度较慢。常见的非对称加密算法有RSA、DSA、ECC等等。

非对称加密的工作流程：A生成一对非对称密钥，将公钥向所有人公开，B拿到A的公钥后使用A的公钥对信息加密后发送给A，经过加密的信息只有A手中的私钥能解密。这样B可以通过这种方式将自己的公钥加密后发送给A，两方建立起通信，可以通过对方的公钥加密要发送的信息，接收方用自己的私钥解密信息。

## HTTPS的加密过程 \* \* \*

上面已经介绍了对称加密和非对称加密的优缺点，HTTPS是将两者结合起来，使用的对称加密和非对称加密的混合加密算法。具体做法就是使用非对称加密来传输对称密钥来保证安全性，使用对称加密来保证通信的效率。

简化的工作流程：服务端生成一对非对称密钥，将公钥发给客户端。客户端生成对称密钥，用服务端发来的公钥进行加密，加密后发给服务端。服务端收到后用私钥进行解密，得到客户端发送的对称密钥。通信双方就可以通过对称密钥进行高效地通信了。

但是仔细想想这其中存在一个很大地问题，就是客户端最开始如何判断收到的这个公钥就是来自服务端而不是其他人冒充的？

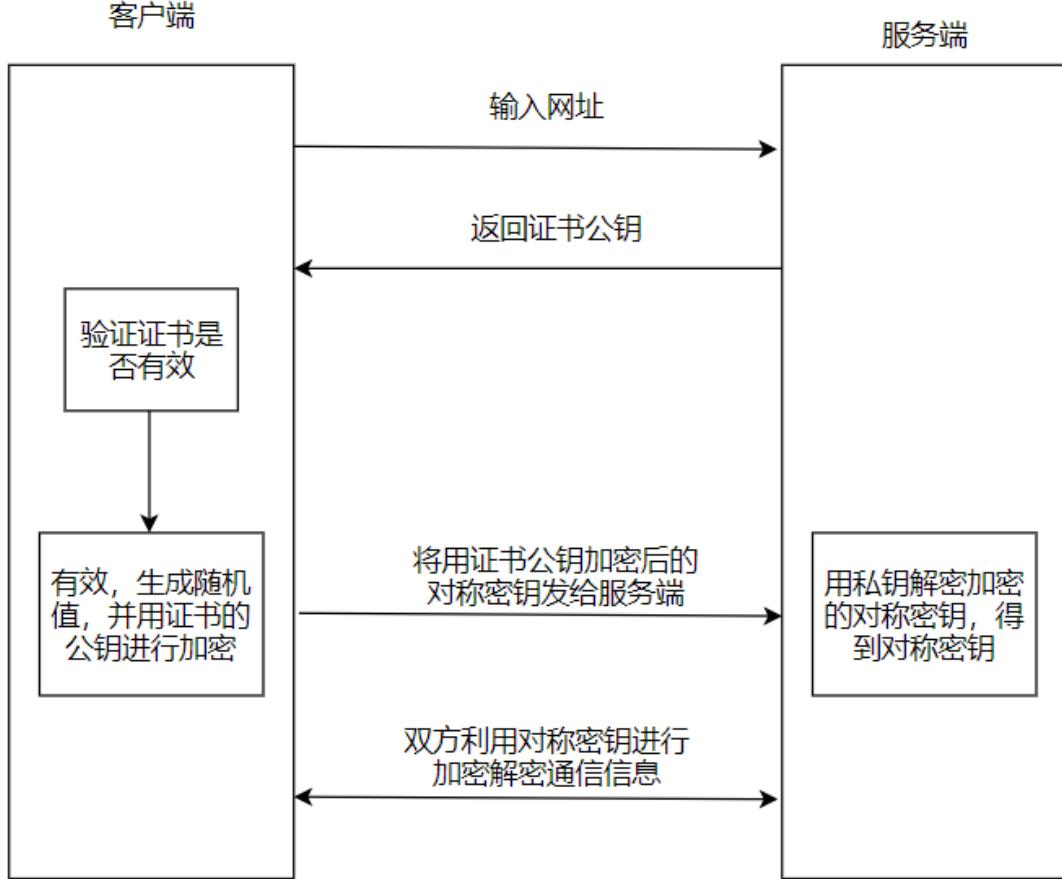
这就需要证书上场了，服务端会向一个权威机构申请一个证书来证明自己的身份，到时候将证书（证书中包含了公钥）发给客户端就可以了，客户端收到证书后既证明了服务端的身份又拿到了公钥就可以进行下一步操作了。

HTTPS的加密过程：

1. 客户端向服务端发起第一次握手请求，告诉服务端客户端所支持的SSL的指定版本、加密算法及密钥长度等信息。
2. 服务端将自己的公钥发给数字证书认证机构，数字证书认证机构利用自己的私钥对服务器的公钥进行数字签名，并给服务器颁发公钥证书。
3. 服务端将证书发给客户端。
4. 客户端利用数字认证机构的公钥，向数字证书认证机构验证公钥证书上的数字签名，确认服务器公开密钥的真实性。
5. 客户端使用服务端的公开密钥加密自己生成的对称密钥，发给服务端。
6. 服务端收到后利用私钥解密信息，获得客户端发来的对称密钥。
7. 通信双方可用对称密钥来加密解密信息。

上述流程存在的一个问题就是客户端哪里来的数字认证机构的公钥，其实，在很多浏览器开发时，会内置常用数字证书认证机构的公钥。

流程图如下：



## 常用HTTP状态码 \* \* \*

这也一个面试经常问的题目,背下来就行了.

状态码	类别
1XX	信息性状态码
2XX	成功状态码
3XX	重定向状态码
4XX	客户端错误状态码
5XX	服务端错误状态码

### 常见的HTTP状态码

#### 1XX

- 100 Continue: 表示正常，客户端可以继续发送请求
- 101 Switching Protocols: 切换协议，服务器根据客户端的请求切换协议。

#### 2XX

- 200 OK: 请求成功
- 201 Created: 已创建，表示成功请求并创建了新的资源
- 202 Accepted: 已接受，已接受请求，但未处理完成。
- 204 No Content: 无内容，服务器成功处理，但未返回内容。
- 205 Reset Content: 重置内容，服务器处理成功，客户端应重置文档视图。

- 206 Partial Content: 表示客户端进行了范围请求, 响应报文应包含Content-Range指定范围的实体内容

3XX

- 301 Moved Permanently: 永久性重定向
- 302 Found: 临时重定向
- 303 See Other: 和301功能类似, 但要求客户端采用get方法获取资源
- 304 Not Modified: 所请求的资源未修改, 服务器返回此状态码时, 不会返回任何资源。
- 305 Use Proxy: 所请求的资源必须通过代理访问
- 307 Temporary Redirect: 临时重定向, 与302类似, 要求使用get请求重定向。

4XX

- 400 Bad Request: 客户端请求的语法错误, 服务器无法理解。
- 401 Unauthorized: 表示发送的请求需要有认证信息。
- 403 Forbidden: 服务器理解用户的请求, 但是拒绝执行该请求
- 404 Not Found: 服务器无法根据客户端的请求找到资源。
- 405 Method Not Allowed: 客户端请求中的方法被禁止
- 406 Not Acceptable: 服务器无法根据客户端请求的内容特性完成请求
- 408 Request Time-out: 服务器等待客户端发送的请求时间过长, 超时

5XX

- 500 Internal Server Error: 服务器内部错误, 无法完成请求
- 501 Not Implemented: 服务器不支持请求的功能, 无法完成请求

## 常见的HTTP方法 \* \* \*

方法	作用
GET	获取资源
POST	传输实体主体
PUT	上传文件
DELETE	删除文件
HEAD	和GET方法类似, 但只回报文首部, 不回报文实体主体部分
PATCH	对资源进行部分修改
OPTIONS	查询指定的URL支持的方法
CONNECT	要求用隧道协议连接代理
TRACE	服务器会将通信路径返回给客户端

为了方便记忆, 可以将PUT、DELETE、POST、GET理解为客户端对服务端的增删改查。

- PUT: 上传文件, 向服务器添加数据, 可以看作增
- DELETE: 删除文件
- POST: 传输数据, 向服务器提交数据, 对服务器数据进行更新。
- GET: 获取资源, 查询服务器资源

## GET和POST区别 \* \* \*

- 作用  
    GET用于获取资源， POST用于传输实体主体
- 参数位置  
    GET的参数放在URL中， POST的参数存储在实体主体中，并且GET方法提交的请求的URL中的数据做多是2048字节， POST请求没有大小限制。
- 安全性  
    GET方法因为参数放在URL中， 安全性相对于POST较差一些
- 幂等性  
    GET方法是具有幂等性的， 而POST方法不具有幂等性。这里幂等性指客户端连续发出多次请求， 收到的结果都是一样的.

## HTTP 1.0、 HTTP 1.1及HTTP 2.0的主要区别是什么 \* \*

### HTTP 1.0和HTTP 1.1的区别

- 长连接  
    HTTP 1.1支持长连接和请求的流水线操作。长连接是指不在需要每次请求都重新建立一次连接， HTTP 1.0默认使用短连接， 每次请求都要重新建立一次TCP连接， 资源消耗较大。请求的流水线操作是指客户端在收到HTTP的响应报文之前可以先发送新的请求报文， 不支持请求的流水线操作需要等到收到HTTP的响应报文后才能继续发送新的请求报文。
- 缓存处理  
    在HTTP 1.0中主要使用header中的If-Modified-Since,Expires作为缓存判断的标准， HTTP 1.1引入了Entity tag, If-Unmodified-Since, If-Match等更多可供选择的缓存头来控制缓存策略。
- 错误状态码  
    在HTTP 1.1新增了24个错误状态响应码
- HOST域  
    在HTTP 1.0 中认为每台服务器都会绑定唯一的IP地址， 所以， 请求中的URL并没有传递主机名。但后来一台服务器上可能存在多个虚拟机， 它们共享一个IP地址， 所以HTTP 1.1中请求消息和响应消息都应该支持Host域。
- 带宽优化及网络连接的使用  
    在HTTP 1.0中会存在浪费带宽的现象， 主要是因为不支持断点续传功能， 客户端只是需要某个对象的一部分， 服务端却将整个对象都传了过来。在HTTP 1.1中请求头引入了range头域， 它支持只请求资源的某个部分， 返回的状态码为206。

### HTTP 2.0的新特性

- 新的二进制格式： HTTP 1.x的解析是基于文本， HTTP 2.0的解析采用二进制， 实现方便， 健壮性更好。
- 多路复用： 每一个request对应一个id， 一个连接上可以有多个request， 每个连接的request可以随机混在一起， 这样接收方可以根据request的id将request归属到各自不同的服务端请求里。
- header压缩： 在HTTP 1.x中， header携带大量信息，并且每次都需要重新发送， HTTP 2.0采用编码的方式减小了header的大小， 同时通信双方各自缓存一份header fields表， 避免了header的重复传输。
- 服务端推送： 客户端在请求一个资源时， 会把相关资源一起发给客户端， 这样客户端就不需要再次发起请求。

## Session、Cookie和Token的主要区别 \* \* \*

HTTP协议是无状态的，即服务器无法判断用户身份。Session和Cookie可以用来进行身份辨认。

- Cookie

Cookie是保存在客户端一个小数据块，其中包含了用户信息。当客户端向服务端发起请求，服务端会像客户端浏览器发送一个Cookie，客户端会把Cookie存起来，当下次客户端再次请求服务端时，会携带上这个Cookie，服务端会通过这个Cookie来确定身份。

- Session

Session是通过Cookie实现的，和Cookie不同的是，Session是存在服务端的。当客户端浏览器第一次访问服务器时，服务器会为浏览器创建一个sessionid，将sessionid放到Cookie中，存在客户端浏览器。比如浏览器访问的是购物网站，将一本《图解HTTP》放到了购物车，当浏览器再次访问服务器时，服务器会取出Cookie中的sessionid，并根据sessionid获取会话中的存储的信息，确认浏览器的身份是上次将《图解HTTP》放入到购物车那个用户。

- Token

客户端在浏览器第一次访问服务端时，服务端生成的一串字符串作为Token发给客户端浏览器，下次浏览器在访问服务端时携带token即可无需验证用户名和密码，省下来大量的资源开销。看到这里很多人感觉这不是和sessionid作用一样吗？其实是不一样的，但是本文章主要针对面试，知识点很多，篇幅有限，几句话也解释不清楚，大家可以看看这篇文章，我觉得说的非常清楚了。

[cookie、session与token的真正区别](#)

下面为了方便记忆，做了一个表格进行对比。

	存放位置	占用空间	安全性	应用场景
Cookie	客户端浏览器	小	较低	一般存放配置信息
Session	服务端	多	较高	存放较为重要的信息

## 如果客户端禁止 cookie 能实现 session 还能用吗？ \*

可以，Session的作用是在服务端来保持状态，通过sessionid来进行确认身份，但sessionid一般是通过Cookie来进行传递的。如果Cookie被禁用了，可以通过在URL中传递sessionid。

## 在浏览器中输入url地址到显示主页的过程 \* \* \*

面试超高频的一道题，一般能说清楚流程就可以。

1. 对输入到浏览器的url进行DNS解析，将域名转换为IP地址。
2. 和目的服务器建立TCP连接
3. 向目的服务器发送HTTP请求
4. 服务器处理请求并返回HTTP报文
5. 浏览器解析并渲染页面

## Servlet是线程安全的吗 \*

Servlet不是线程安全的，多线程的读写会导致数据不同步的问题。

关注公众号“路人zhang”，获取更多面试技巧及资料

关注知乎“路人zhang”，聊聊码农那些事



# MySQL数据库

## 什么是MySQL? \*

百度百科上的解释：MySQL是一种开放源代码的关系型数据库管理系统（RDBMS），使用最常用的数据库管理语言--结构化查询语言（SQL）进行数据库管理。MySQL是开放源代码的，因此任何人都可以在General Public License的许可下下载并根据个性化的需要对其进行修改。

## MySQL常用的存储引擎有什么？它们有什么区别？ \* \* \*

- InnoDB

InnoDB是MySQL的默认存储引擎，支持事务、行锁和外键等操作。

- MyISAM

MyISAM是MySQL5.1版本前的默认存储引擎，MyISAM的并发性比较差，不支持事务和外键等操作，默认的锁的粒度为表级锁。

	InnoDB	MyISAM
外键	支持	不支持
事务	支持	不支持
锁	支持表锁和行锁	支持表锁
可恢复性	根据事务日志进行恢复	无事务日志
表结构	数据和索引是集中存储的，.ibd 和.frm	数据和索引是分开存储的，数据.MYD，索引.MYI
查询性能	一般情况相比于MyISAM较差	一般情况相比于InnoDB较差
索引	聚簇索引	非聚簇索引

# 数据库的三大范式 \* \*

- 第一范式：确保每列保持原子性，数据表中的所有字段值都是不可分解的原子值。
- 第二范式：确保表中的每列都和主键相关
- 第三范式：确保每列都和主键列直接相关而不是间接相关

# MySQL的数据类型有哪些 \* \*

- 整数

TINYINT、SMALLINT、MEDIUMINT、INT、BIGINT分别占用8、16、24、32、64位存储空间。值得注意的是，INT(10)中的10只是表示显示字符的个数，并无实际意义。一般和UNSIGNED ZEROFILL配合使用才有实际意义，例如，数据类型INT(3)，属性为UNSIGNED ZEROFILL，如果插入的数据为3的话，实际存储的数据为003。

- 浮点数

FLOAT、DOUBLE及DECIMAL为浮点数类型，DECIMAL是利用字符串进行处理的，能存储精确的小数。相比于FLOAT和DOUBLE，DECIMAL的效率更低些。FLOAT、DOUBLE及DECIMAL都可以指定列宽，例如FLOAT(5,2)表示一共5位，两位存储小数部分，三位存储整数部分。

- 字符串

字符串常用的主要有CHAR和VARCHAR，VARCHAR主要用于存储可变长字符串，相比于定长的CHAR更节省空间。CHAR是定长的，根据定义的字符串长度分配空间。

应用场景：对于经常变更的数据使用CHAR更好，CHAR不容易产生碎片。对于非常短的列也是使用CHAR更好些，CHAR相比于VARCHAR在效率上更高些。一般避免使用TEXT/BLOB等类型，因为查询时会使用临时表，造成严重的性能开销。

- 日期

比较常用的有year、time、date、datetime、timestamp等，datetime保存从1000年到9999年的时间，精度位秒，使用8字节的存储空间，与时区无关。timestamp和UNIX的时间戳相同，保存从1970年1月1日午夜到2038年的时间，精度到秒，使用四个字节的存储空间，并且与时区相关。

应用场景：尽量使用timestamp，相比于datetime它有着更高的空间效率。

# 索引 \* \* \*

## 什么是索引？

百度百科的解释：索引是对数据库表的一列或者多列的值进行排序一种结构，使用索引可以快速访问数据表中的特定信息。

## 索引的优缺点？

优点：

- 大大加快数据检索的速度。
- 将随机I/O变成顺序I/O(因为B+树的叶子节点是连接在一起的)
- 加速表与表之间的连接

缺点：

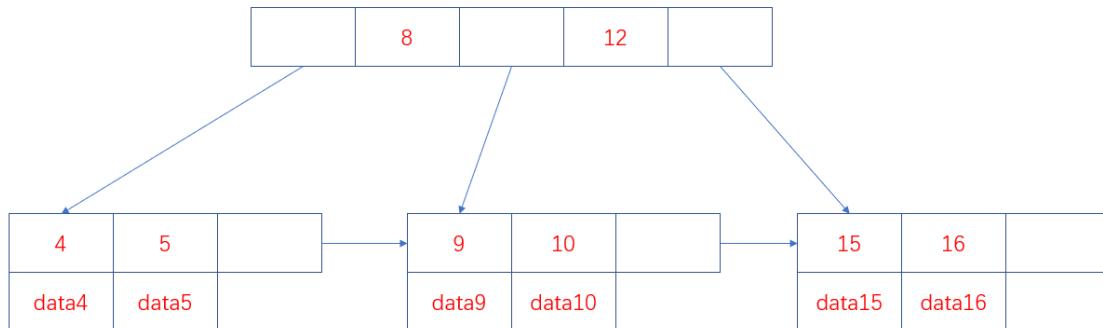
- 从空间角度考虑，建立索引需要占用物理空间
- 从时间角度考虑，创建和维护索引都需要花费时间，例如对数据进行增删改的时候都需要维护索引。

## 索引的数据结构？

索引的数据结构主要有B+树和哈希表，对应的索引分别为B+树索引和哈希索引。InnoDB引擎的索引类型有B+树索引和哈希索引，默认的索引类型为B+树索引。

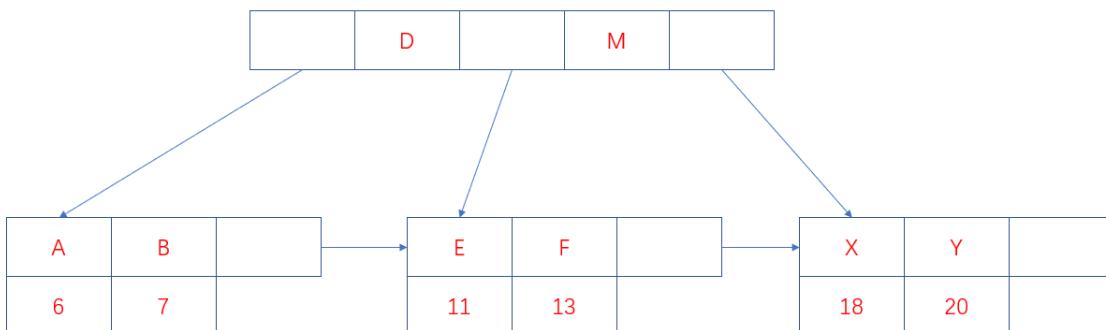
- B+树索引

熟悉数据结构的同学都知道，B+树、平衡二叉树、红黑树都是经典的数据结构。在B+树中，所有的记录节点都是按照键值大小的顺序放在叶子节点上，如下图。



从上图可以看出，因为B+树具有有序性，并且所有的数据都存放在叶子节点，所以查找的效率非常高，并且支持排序和范围查找。

B+树的索引又可以分为主索引和辅助索引。其中主索引为聚簇索引，辅助索引为非聚簇索引。聚簇索引是以主键作为B+树索引的键值所构成的B+树索引，聚簇索引的叶子节点存储着完整的数据记录；非聚簇索引是以非主键的列作为B+树索引的键值所构成的B+树索引，非聚簇索引的叶子节点存储着主键值。所以使用非聚簇索引进行查询时，会先找到主键值，然后到根据聚簇索引找到主键对应的数据域。上图中叶子节点存储的是数据记录，为聚簇索引的结构图，非聚簇索引的结构图如下：



上图中的字母为数据的非主键的列值，假设要查询该列值为B的信息，则需先找到主键7，在到聚簇索引中查询主键7所对应的数据域。

- 哈希索引

哈希索引是基于哈希表实现的，对于每一行数据，存储引擎会对索引列通过哈希算法进行哈希计算得到哈希码，并且哈希算法要尽量保证不同的列值计算出的哈希码值是不同的，将哈希码的值作为哈希表的key值，将指向数据行的指针作为哈希表的value值。这样查找一个数据的时间复杂度就是 $O(1)$ ，一般多用于精确查找。

## Hash索引和B+树的区别？

因为两者数据结构上的差异导致它们的使用场景也不同，哈希索引一般多用于精确的等值查找，B+索引则多用于除了精确的等值查找外的其他查找。在大多数情况下，会选择使用B+树索引。

- 哈希索引不支持排序，因为哈希表是无序的。
- 哈希索引不支持范围查找。

- 哈希索引不支持模糊查询及多列索引的最左前缀匹配。
- 因为哈希表中会存在哈希冲突，所以哈希索引的性能是不稳定的，而B+树索引的性能是相对稳定的，每次查询都是从根节点到叶子节点

## 索引的类型有哪些？

MySQL主要的索引类型主要有FULLTEXT, HASH, BTREE, RTREE。

- FULLTEXT

FULLTEXT即全文索引，MyISAM存储引擎和InnoDB存储引擎在MySQL5.6.4以上版本支持全文索引，一般用于查找文本中的关键字，而不是直接比较是否相等，多在CHAR, VARCHAR, TEXT等数据类型上创建全文索引。全文索引主要是用来解决WHERE name LIKE "%zhang%"等针对文本的模糊查询效率低的问题。

- HASH

HASH即哈希索引，哈希索引多用于等值查询，时间复杂度为 $O(1)$ ，效率非常高，但不支持排序、范围查询及模糊查询等。

- BTREE

BTREE即B+树索引，INnoDB存储引擎默认的索引，支持排序、分组、范围查询、模糊查询等，并且性能稳定。

- RTREE

RTREE即空间数据索引，多用于地理数据的存储，相比于其他索引，空间数据索引的优势在于范围查找

## 索引的种类有哪些？

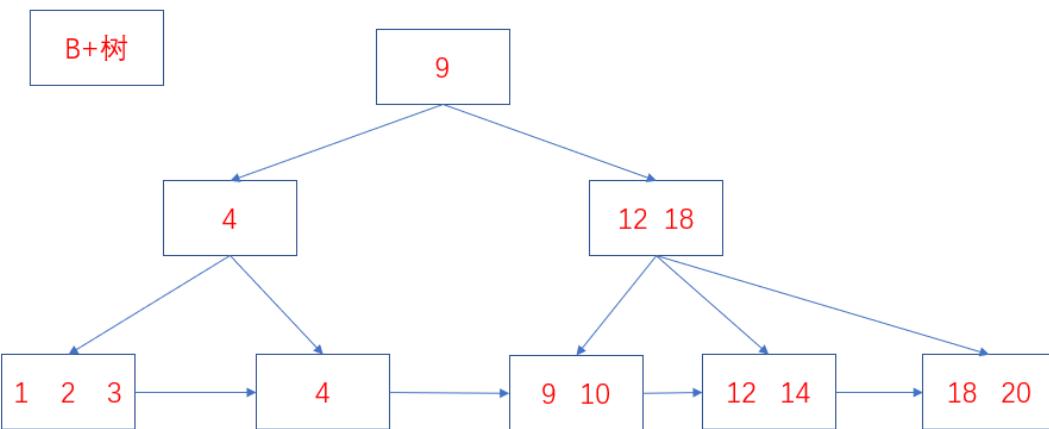
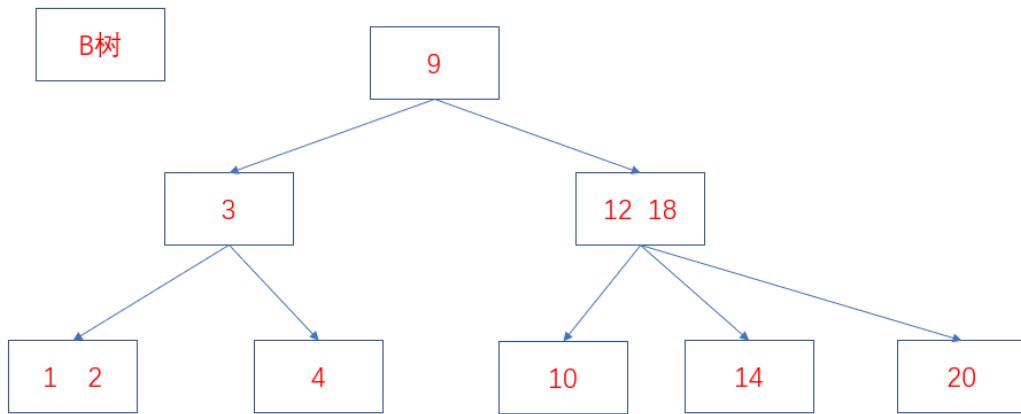
- 主键索引：数据列不允许重复，不能为NULL，一个表只能有一个主键索引
- 组合索引：由多个列值组成的索引。
- 唯一索引：数据列不允许重复，可以为NULL，索引列的值必须唯一的，如果是组合索引，则列值的组合必须唯一。
- 全文索引：对文本的内容进行搜索。
- 普通索引：基本的索引类型，可以为NULL

## B树和B+树的区别？

B树和B+树最主要的区别主要有两点：

- B树中的内部节点和叶子节点均存放键和值，而B+树的内部节点只有键没有值，叶子节点存放所有的键和值。
- B + 树的叶子节点是通过相连在一起的，方便顺序检索。

两者的结构图如下。



## 数据库为什么使用B+树而不是B树？

- B树适用于随机检索，而B+树适用于随机检索和顺序检索
- B+树的空间利用率更高，因为B树每个节点要存储键和值，而B+树的内部节点只存储键，这样B+树的一个节点就可以存储更多的索引，从而使树的高度变低，减少了I/O次数，使得数据检索速度更快。
- B+树的叶子节点都是连接在一起的，所以范围查找，顺序查找更加方便
- B+树的性能更加稳定，因为在B+树中，每次查询都是从根节点到叶子节点，而在B树中，要查询的值可能不在叶子节点，在内部节点就已经找到。

那在什么情况适合使用B树呢，因为B树的内部节点也可以存储值，所以可以把一些频繁访问的值放在距离根节点比较近的地方，这样就可以提高查询效率。综上所述，B+树的性能更加适合作为数据库的索引。

## 什么是聚簇索引，什么是非聚簇索引？

聚簇索引和非聚簇索引最主要的区别是**数据和索引是否分开存储**。

- 聚簇索引：将数据和索引放到一起存储，索引结构的叶子节点保留了数据行。
- 非聚簇索引：将数据和索引分开存储，索引叶子节点存储的是指向数据行的地址。

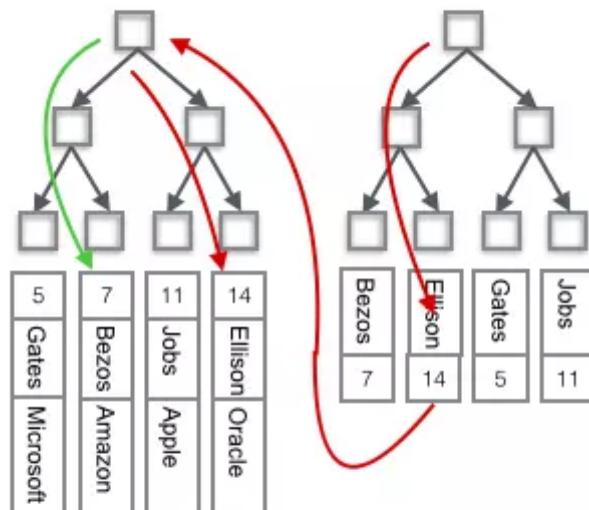
在InnoDB存储引擎中，默认的索引为B+树索引，利用主键创建的索引为主索引，也是聚簇索引，在主索引之上创建的索引为辅助索引，也是非聚簇索引。为什么说辅助索引是在主索引之上创建的呢，因为辅助索引中的叶子节点存储的是主键。

在MyISAM存储引擎中，默认的索引也是B+树索引，但主索引和辅助索引都是非聚簇索引，也就是说索引结构的叶子节点存储的都是一个指向数据行的地址。并且使用辅助索引检索无需访问主键的索引。

可以从非常经典的两张图看看它们的区别(图片来源于网络)：

主键索引

辅助键索引



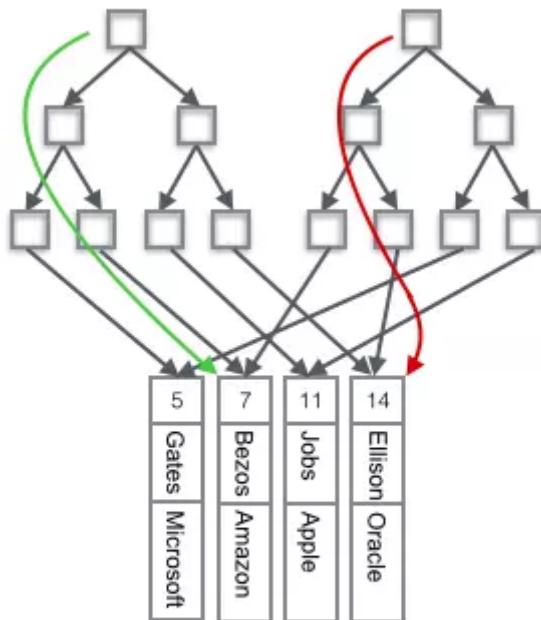
→ 主索引检索过程

→ 辅助索引检索过程

InnoDB (聚簇) 表分布

主键索引

辅助键索引



MyISAM (非聚簇) 表分布

## 非聚簇索引一定会进行回表查询吗？

上面是说了非聚簇索引的叶子节点存储的是主键，也就是说要先通过非聚簇索引找到主键，再通过聚簇索引找到主键所对应的数据，后面这个再通过聚簇索引找到主键对应的数据的过程就是回表查询，那么非聚簇索引就一定会进行回表查询吗？

答案是不一定的，这里涉及到一个索引覆盖的问题，如果查询的数据再辅助索引上完全能获取到便不需要回表查询。例如有一张表存储着个人信息包括id、name、age等字段。假设聚簇索引是以ID为键值构建的索引，非聚簇索引是以name为键值构建的索引，`select id, name from user where name = 'zhangsan'`; 这个查询便不需要进行回表查询因为，通过非聚簇索引已经能全部检索出数据，这就是索引覆盖的情况。如果查询语句是这样，`select id, name, age from user where name = 'zhangsan'`; 则需要进行回表查询，因为通过非聚簇索引不能检索出age的值。那应该如何解决那呢？只需要将索引覆盖即可，建立age和name的联合索引再使用`select id, name, age from user where name = 'zhangsan'`; 进行查询即可。

所以通过索引覆盖能解决非聚簇索引回表查询的问题。

## 索引的使用场景有哪些？

- 对于中大型表建立索引非常有效，对于非常小的表，一般全部表扫描速度更快些。
- 对于超大型的表，建立和维护索引的代价也会变高，这时可以考虑分区技术。
- 如何表的增删改非常多，而查询需求非常少的话，那就没有必要建立索引了，因为维护索引也是需要代价的。
- 一般不会出现再where条件中的字段就没有必要建立索引了。
- 多个字段经常被查询的话可以考虑联合索引。
- 字段多且字段值没有重复的时候考虑唯一索引。
- 字段多且有重复的时候考虑普通索引。

## 索引的设计原则？

- 最适合索引的列是在where后面出现的列或者连接句子中指定的列，而不是出现在SELECT关键字后面的选择列表中的列。
- 索引列的基数越大，索引的效果越好，换句话说就是索引列的区分度越高，索引的效果越好。比如使用性别这种区分度很低的列作为索引，效果就会很差，因为列的基数最多也就是三种，大多不是男性就是女性。
- 尽量使用短索引，对于较长的字符串进行索引时应该指定一个较短的前缀长度，因为较小的索引涉及到的磁盘I/O较少，并且索引高速缓存中的块可以容纳更多的键值，会使得查询速度更快。
- 尽量利用最左前缀。
- 不要过度索引，每个索引都需要额外的物理空间，维护也需要花费时间，所以索引不是越多越好。

## 如何对索引进行优化？

对索引的优化其实最关键的就是要符合索引的设计原则和应用场景，将不符合要求的索引优化成符合索引设计原则和应用场景的索引。

除了索引的设计原则和应用场景那几点外，还可以从以下两方面考虑。

- 在进行查询时，索引列不能是表达式的一部分，也不能是函数的参数，因为这样无法使用索引。例如`select * from table_name where a + 1 = 2`
- 将区分度最高的索引放在前面
- 尽量少使用`select*`

索引的使用场景、索引的设计原则和如何对索引进行优化可以看成一个问题。

## 如何创建/删除索引？

创建索引：

- 使用CREATE INDEX语句

```
CREATE INDEX index_name ON table_name (column_list);
```

- 在CREATE TABLE时创建

```
CREATE TABLE user(
    id INT PRIMARY KEY,
    information TEXT,
    FULLTEXT KEY (information)
);
```

- 使用ALTER TABLE创建索引

```
ALTER TABLE table_name ADD INDEX index_name (column_list);
```

删除索引：

- 删除主键索引

```
alter table 表名 drop primary key
```

- 删除其他索引

```
alter table 表名 drop key 索引名
```

## 使用索引查询时性能一定会提升吗？

不一定，前面在索引的使用场景和索引的设计原则中已经提到了如何合理地使用索引，因为创建和维护索引需要花费空间和时间上的代价，如果不合理地使用索引反而会使查询性能下降。

## 什么是前缀索引？

前缀索引是指对文本或者字符串的前几个字符建立索引，这样索引的长度更短，查询速度更快。

使用场景：前缀的区分度比较高的情况下。

建立前缀索引的方式

```
ALTER TABLE table_name ADD KEY(column_name(prefix_length));
```

这里面有个prefix\_length参数很难确定，这个参数就是前缀长度的意思。通常可以使用以下方法进行确定，先计算全列的区分度

```
SELECT COUNT(DISTINCT column_name) / COUNT(*) FROM table_name;
```

然后在计算前缀长度为多少时和全列的区分度最相似。

```
SELECT COUNT(DISTINCT LEFT(column_name, prefix_length)) / COUNT(*) FROM table_name;
```

不断地调整prefix\_length的值，直到和全列计算出区分度相近。

## 什么是最左匹配原则？

最左匹配原则：从最左边为起点开始连续匹配，遇到范围查询 (<、>、between、like) 会停止匹配。

例如建立索引(a,b,c)，大家可以猜测以下几种情况是否用到了索引。

- 第一种

```
select * from table_name where a = 1 and b = 2 and c = 3
select * from table_name where b = 2 and a = 1 and c = 3
```

上面两次查询过程中所有值都用到了索引，where后面字段调换不会影响查询结果，因为MySQL中的优化器会自动优化查询顺序。

- 第二种

```
select * from table_name where a = 1
select * from table_name where a = 1 and b = 2
select * from table_name where a = 1 and b = 2 and c = 3
```

答案是三个查询语句都用到了索引，因为三个语句都是从最左开始匹配的。

- 第三种

```
select * from table_name where b = 1
select * from table_name where b = 1 and c = 2
```

答案是这两个查询语句都没有用到索引，因为不是从最左边开始匹配的

- 第四种

```
select * from table_name where a = 1 and c = 2
```

这个查询语句只有a列用到了索引，c列没有用到索引，因为中间跳过了b列，不是从最左开始连续匹配的。

- 第五种

```
select * from table_name where a = 1 and b < 3 and c < 1
```

这个查询中只有a列和b列使用到了索引，而c列没有使用索引，因为根据最左匹配查询原则，遇到范围查询会停止。

- 第六种

```
select * from table_name where a like 'ab%';
select * from table_name where a like '%ab'
select * from table_name where a like '%ab%'
```

对于列为字符串的情况，只有前缀匹配可以使用索引，中缀匹配和后缀匹配只能进行全表扫描。

## 索引在什么情况下会失效？

在上面介绍了几种不符合最左匹配原则的情况会导致索引失效，除此之外，以下这几种情况也会导致索引失效。

- 条件中有or，例如 `select * from table_name where a = 1 or b = 3`
- 在索引上进行计算会导致索引失效，例如 `select * from table_name where a + 1 = 2`
- 在索引的类型上进行数据类型的隐形转换，会导致索引失效，例如字符串一定要加引号，假设 `select * from table_name where a = '1'` 会使用到索引，如果写成 `select * from table_name where a = 1` 则会导致索引失效。
- 在索引中使用函数会导致索引失效，例如 `select * from table_name where abs(a) = 1`
- 在使用like查询时以%开头会导致索引失效
- 索引上使用!、=、<>进行判断时会导致索引失效，例如 `select * from table_name where a != 1`
- 索引字段上使用 is null/is not null判断时会导致索引失效，例如 `select * from table_name where a is null`

# 数据库的事务 \* \* \*

## 什么是数据库的事务？

百度百科的解释：数据库事务(transaction)是访问并可能操作各种数据项的一个数据库操作序列，这些操作要么全部执行，要么全部不执行，是一个不可分割的工作单位。事务由事务开始与事务结束之间执行的全部数据库操作组成。

## 事务的四大特性是什么？

- 原子性：原子性是指包含事务的操作要么全部执行成功，要么全部失败回滚。
- 一致性：一致性指事务在执行前后状态是一致的。
- 隔离性：一个事务所进行的修改在最终提交之前，对其他事务是不可见的。
- 持久性：数据一旦提交，其所作的修改将永久地保存到数据库中。

## 数据库的并发一致性问题

当多个事务并发执行时，可能会出现以下问题：

- 脏读：事务A更新了数据，但还没有提交，这时事务B读取到事务A更新后的数据，然后事务A回滚了，事务B读取到的数据就成为脏数据了。
- 不可重复读：事务A对数据进行多次读取，事务B在事务A多次读取的过程中执行了更新操作并提交了，导致事务A多次读取到的数据并不一致。
- 幻读：事务A在读取数据后，事务B向事务A读取的数据中插入了几条数据，事务A再次读取数据时发现多了几条数据，和之前读取的数据不一致。
- 丢失修改：事务A和事务B都对同一个数据进行修改，事务A先修改，事务B随后修改，事务B的修改覆盖了事务A的修改。

不可重复度和幻读看起来比较像，它们主要的区别是：在不可重复读中，发现数据不一致主要是数据被更新了。在幻读中，发现数据不一致主要是数据增多或者减少了。

## 数据库的隔离级别有哪些？

- 未提交读：一个事务在提交前，它的修改对其他事务也是可见的。
- 提交读：一个事务提交之后，它的修改才能被其他事务看到。
- 可重复读：在同一个事务中多次读取到的数据是一致的。
- 串行化：需要加锁实现，会强制事务串行执行。

数据库的隔离级别分别可以解决数据库的脏读、不可重复读、幻读等问题。

隔离级别	脏读	不可重复读	幻读
未提交读	允许	允许	允许
提交读	不允许	允许	允许
可重复读	不允许	不允许	允许
串行化	不允许	不允许	不允许

MySQL的默认隔离级别是可重复读。

## 隔离级别是如何实现的？

事务的隔离机制主要是依靠锁机制和MVCC(多版本并发控制)实现的，提交读和可重复读可以通过MVCC实现，串行化可以通过锁机制实现。

## 什么是MVCC？

MVCC(multiple version concurrent control)是一种控制并发的方法，主要用来提高数据库的并发性能。

在了解MVCC时应该先了解当前读和快照读。

- 当前读：读取的是数据库的最新版本，并且在读取时要保证其他事务不会修改该当前记录，所以会对读取的记录加锁。
- 快照读：不加锁读取操作即为快照读，使用MVCC来读取快照中的数据，避免加锁带来的性能损耗。

可以看到MVCC的作用就是在不加锁的情况下，解决数据库读写冲突问题，并且解决脏读、幻读、不可重复读等问题，但是不能解决丢失修改问题。

MVCC的实现原理：

- 版本号

系统版本号：是一个自增的ID，每开启一个事务，系统版本号都会递增。

事务版本号：事务版本号就是事务开始时的系统版本号，可以通过事务版本号的大小判断事务的时间顺序。

- 行记录隐藏的列

DB\_ROW\_ID：所需空间6byte，隐含的自增ID，用来生成聚簇索引，如果数据表没有指定聚簇索引，InnoDB会利用这个隐藏ID创建聚簇索引。

DB\_TRX\_ID：所需空间6byte，最近修改的事务ID，记录创建这条记录或最后一次修改这条记录的事务ID。

DB\_ROLL\_PTR：所需空间7byte，回滚指针，指向这条记录的上一个版本。

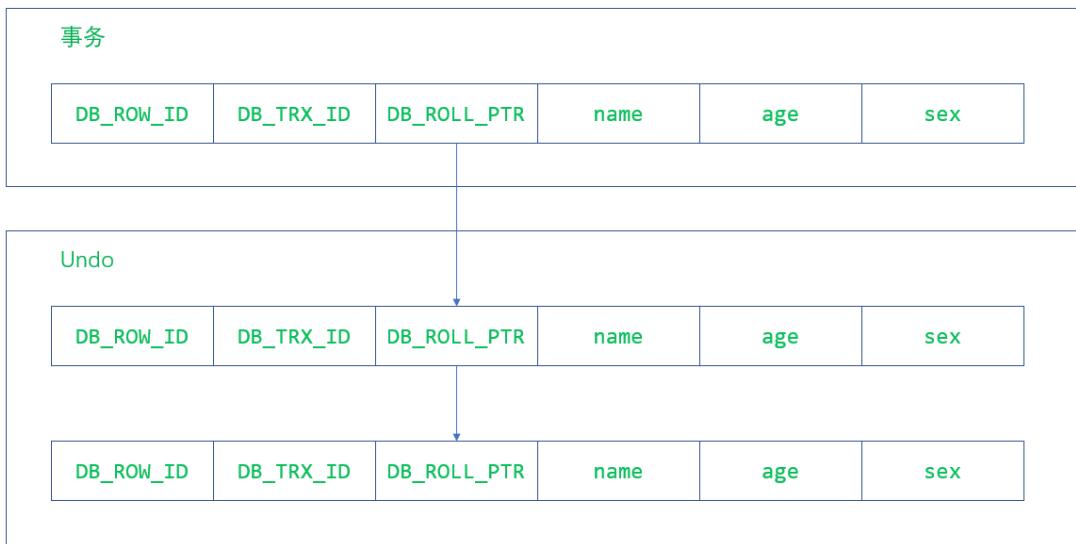
它们大致长这样，省略了具体字段的值。·

某条记录

DB_ROW_ID	DB_TRX_ID	DB_ROLL_PTR	name	age	sex
-----------	-----------	-------------	------	-----	-----

- undo日志

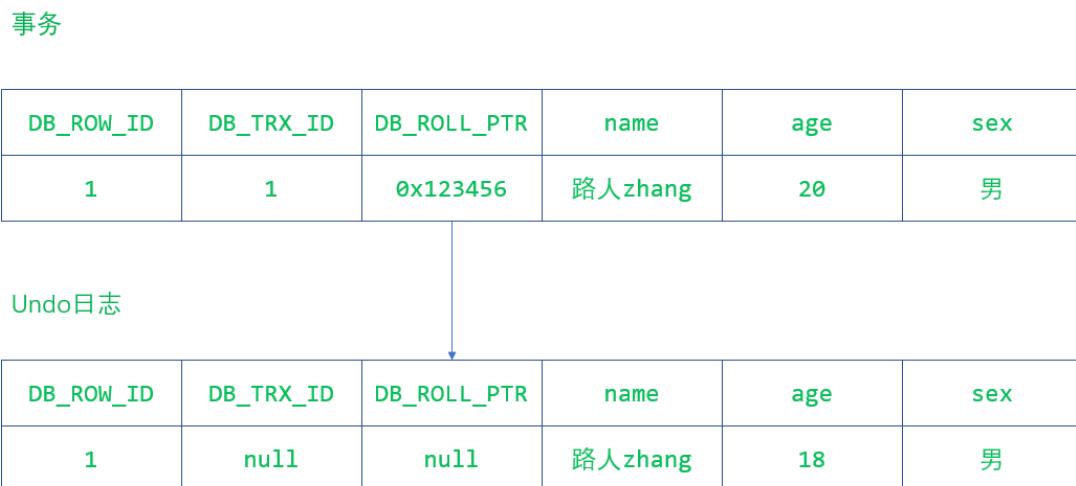
MVCC做使用到的快照会存储在Undo日志中，该日志通过回滚指针将一个一个数据行的所有快照连接起来。它们大致长这样。



举一个简单的例子说明下，比如最开始的某条记录长这样

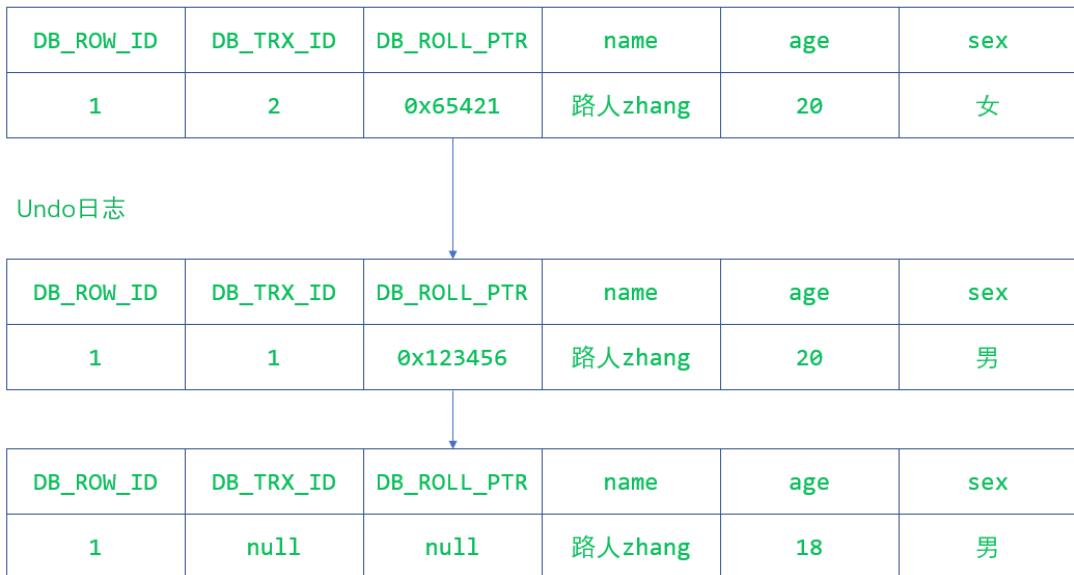
DB_ROW_ID	DB_TRX_ID	DB_ROLL_PTR	name	age	sex
1	null	null	路人zhang	18	男

现在来了一个事务对他的年龄字段进行了修改，变成了这样



现在又来了一个事务2对它的性别进行了修改，它又变成了这样

## 事务2



从上面的分析可以看出，事务对同一记录的修改，记录的各个会在Undo日志中连接成一个线性表，在表头的就是最新的旧纪录。

在重复读的隔离级别下，InnoDB的工作流程：

- SELECT

作为查询的结果要满足两个条件：

1. 当前事务所要查询的数据行快照的创建版本号必须小于当前事务的版本号，这样做的目的是保证当前事务读取的数据行的快照要么是在当前事务开始前就已经存在的，要么就是当前事务自身插入或者修改过的。
2. 当前事务所要读取的数据行快照的删除版本号必须是大于当前事务的版本号，如果是小于等于的话，表示该数据行快照已经被删除，不能读取。

- INSERT

将当前系统版本号作为数据行快照的创建版本号。

- DELETE

将当前系统版本号作为数据行快照的删除版本号。

- UPDATE

保存当前系统版本号为更新前的数据行快照创建行版本号，并保存当前系统版本号为更新后的数据行快照的删除版本号，其实就是，先删除再插入即为更新。

总结一下，MVCC的作用就是在避免加锁的情况下最大限度解决读写并发冲突的问题，它可以实现提交读和可重复度两个隔离级。

## 数据库的锁 \* \* \*

### 什么是数据库的锁？

当数据库有并发事务的时候，保证数据访问顺序的机制称为锁机制。

## 数据库的锁与隔离级别的关系？

隔离级别	实现方式
未提交读	总是读取最新的数据，无需加锁
提交读	读取数据时加共享锁，读取数据后释放共享锁
可重复读	读取数据时加共享锁，事务结束后释放共享锁
串行化	锁定整个范围的键，一直持有锁直到事务结束

## 数据库锁的类型有哪些？

按照锁的粒度可以将MySQL锁分为三种：

MySQL锁类别	资源开销	加锁速度	是否会出现死锁	锁的粒度	并发度
表级锁	小	快	不会	大	低
行级锁	大	慢	会	小	高
页面锁	一般	一般	不会	一般	一般

MyISAM默认采用表级锁，InnoDB默认采用行级锁。

从锁的类别上区别可以分为共享锁和排他锁

- 共享锁：共享锁又称读锁，简写为S锁，一个事务对一个数据对象加了S锁，可以对这个数据对象进行读取操作，但不能进行更新操作。并且在加锁期间其他事务只能对这个数据对象加S锁，不能加X锁。
- 排他锁：排他锁又称为写锁，简写为X锁，一个事务对一个数据对象加了X锁，可以对这个对象进行读取和更新操作，加锁期间，其他事务不能对该数据对象进行加X锁或S锁。

它们的兼容情况如下（不太会用excel，图太丑了）：

请求锁模式 是否兼容 当前锁模式	读锁	写锁
读锁	是	否
写锁	否	否

## MySQL中InnoDB引擎的行锁模式及其是如何实现的？

### 行锁模式

在存在行锁和表锁的情况下，一个事务想对某个表加X锁时，需要先检查是否有其他事务对这个表加了锁或对这个表的某一行加了锁，对表的每一行都进行检测一次这是非常低效率的，为了解决这种问题，实现多粒度锁机制，InnoDB还有两种内部使用的意向锁，两种意向锁都是表锁。

- 意向共享锁：简称IS锁，一个事务打算给数据行加共享锁前必须先获得该表的IS锁。
- 意向排他锁：简称IX锁，一个事务打算给数据行加排他锁前必须先获得该表的IX锁。

有了意向锁，一个事务想对某个表加X锁，只需要检查是否有其他事务对这个表加了X/IX/S/IS锁即可。

锁的兼容性如下：

请求锁模式 是否兼容 当前锁模式	X	IX	S	IS
X	否	否	否	否
IX	否	是	否	是
S	否	否	是	是
IS	否	是	是	是

行锁实现方式：InnoDB的行锁是通过给索引上的索引项加锁实现的，如果没有索引，InnoDB将通过隐藏的聚簇索引来对记录进行加锁。

InnoDB行锁主要分三种情况：

- Record lock：对索引项加锁
- Gap lock：对索引之间的“间隙”、第一条记录前的“间隙”或最后一条后的间隙加锁。
- Next-key lock：前两种放入组合，对记录及前面的间隙加锁。

InnoDB行锁的特性：如果不通过索引条件检索数据，那么InnoDB将对表中所有记录加锁，实际产生的效果和表锁是一样的。

MVCC不能解决幻读问题，在可重复读隔离级别下，使用MVCC+Next-Key Locks可以解决幻读问题。

## 什么是数据库的乐观锁和悲观锁，如何实现？

乐观锁：系统假设数据的更新在大多数时候是不会产生冲突的，所以数据库只在更新操作提交的时候对数据检测冲突，如果存在冲突，则数据更新失败。

乐观锁实现方式：一般通过版本号和CAS算法实现。

悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。通俗讲就是每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁。

悲观锁的实现方式：通过数据库的锁机制实现，对查询语句添加for update。

## 什么是死锁？如何避免？

死锁是指两个或者两个以上进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象。在MySQL中，MyISAM是一次获得所需的全部锁，要么全部满足，要么等待，所以不会出现死锁。在InnoDB存储引擎中，除了单个SQL组成的事务外，锁都是逐步获得的，所以存在死锁问题。

如何避免MySQL发生死锁或锁冲突：

- 如果不同的程序并发存取多个表，尽量以相同的顺序访问表。
- 在程序以批量方式处理数据的时候，如果已经对数据排序，尽量保证每个线程按照固定的顺序来处理记录。
- 在事务中，如果需要更新记录，应直接申请足够级别的排他锁，而不应该先申请共享锁，更新时再申请排他锁，因为在当前用户申请排他锁时，其他事务可能已经获得了相同记录的共享锁，从而造

成锁冲突或者死锁。

- 尽量使用较低的隔离级别
- 尽量使用索引访问数据，使加锁更加准确，从而减少锁冲突的机会
- 合理选择事务的大小，小事务发生锁冲突的概率更低
- 尽量用相等的条件访问数据，可以避免Next-Key锁对并发插入的影响。
- 不要申请超过实际需要的锁级别，查询时尽量不要显示加锁
- 对于一些特定的事务，可以表锁来提高处理速度或减少死锁的概率。

## SQL语句基础知识

### SQL语句主要分为哪几类 \*

- 数据定义语言DDL (Data Definition Language) : 主要有CREATE, DROP, ALTER等对逻辑结构有操作的，包括表结构、视图和索引。
- 数据库查询语言DQL (Data Query Language) : 主要以SELECT为主
- 数据操纵语言DML (Data Manipulation Language) : 主要包括INSERT, UPDATE, DELETE
- 数据控制功能DCL (Data Control Language) : 主要是权限控制能操作，包括GRANT, REVOKE, COMMIT, ROLLBACK等。

### SQL约束有哪些? \*\*

- 主键约束：主键为在表中存在一列或者多列的组合，能唯一标识表中的每一行。一个表只有一个主键，并且主键约束的列不能为空。
- 外键约束：外键约束是指用于在两个表之间建立关系，需要指定引用主表的那一列。只有主表的主键可以被从表用作外键，被约束的从表的列可以不是主键，所以创建外键约束需要先定义主表的主键，然后定义从表的外键。
- 唯一约束：确保表中的一列数据没有相同的值，一个表可以定义多个唯一约束。
- 默认约束：在插入新数据时，如果该行没有指定数据，系统将默认值赋给该行，如果没有设置没默认值，则为NULL。
- Check约束：Check会通过逻辑表达式来判断数据的有效性，用来限制输入一列或者多列的值的范围。在列更新数据时，输入的内容必须满足Check约束的条件。

### 什么是子查询? \*\*

子查询：把一个查询的结果在另一个查询中使用

子查询可以分为以下几类：

- 标量子查询：指子查询返回的是一个值，可以使用 =,>,<,>=,<=,<>等操作符对子查询标量结果进行比较，一般子查询会放在比较式的右侧。

```
SELECT * FROM user WHERE age = (SELECT max(age) from user) //查询年纪最大的人
```

- 列子查询：指子查询的结果是n行一列，一般应用于对表的某个字段进行查询返回。可以使用IN、ANY、SOME和ALL等操作符，不能直接使用

```
SELECT num1 FROM table1 WHERE num1 > ANY (SELECT num2 FROM table2)
```

- 行子查询：指子查询返回的结果一行n列

```
SELECT * FROM user WHERE (age,sex) = (SELECT age,sex FROM user WHERE name='zhangsan')
```

- 表子查询：指子查询是n行n列的一个数据表

```
SELECT * FROM student WHERE (name,age,sex) IN (SELECT name,age,sex FROM class1) //在学生表中找到班级在1班的学生
```

## 了解MySQL的几种连接查询吗？ \* \* \*

MySQL的连接查询主要可以分为外连接，内连接，交叉连接

- 外连接

外连接主要分为左外连接(LEFT JOIN)、右外连接(RIGHT JOIN)、全外连接。

左外连接：显示左表中所有的数据及右表中符合条件的数据，右表中不符合条件的数据为null。



右外连接：显示左表中所有的数据及右表中符合条件的数据，右表中不符合条件的数据为null。



MySQL中不支持全外连接。

- 内连接：只显示符合条件的数据



- 交叉连接：使用笛卡尔积的一种连接。

笛卡尔积，百度百科的解释：两个集合 $X$ 和 $Y$ 的笛卡尔积表示为 $X \times Y$ ，第一个对象是 $X$ 的成员而第二个对象是 $Y$ 的所有可能有序对的其中一个成员。例如： $A=\{a,b\}$ ,  $B=\{0,1,2\}$ ,  $A \times B = \{(a,0), (a,1), (a,2), (b,0), (b,1), (b,2)\}$

举例如下：有两张表分为L表和R表。

L表

A	B
a1	b1
a2	b2
a3	b3

R表

B	C
b1	c1
b2	c2
b4	c3

- 左外连接：`select L.* , R.* from L left join R on L.b=R.b`

A	B	B	C
a1	b1	b1	c1
a2	b2	b2	c2
a3	b3	null	null

- 右外连接：`select L.* , R.* from L right join R on L.b=R.b`

B	C	A	B
b1	c1	a1	b1
b2	c2	a2	b2
b4	c3	null	null

- 内连接：`select L.* , R.* from L inner join R on L.b=R.b`

A	B	B	C
a1	b1	b1	c1
a2	b2	b2	c2

- 交叉连接: `select L.* , R.* from L, R`

A	B	B	C
a1	b1	b1	c1
a1	b1	b2	c2
a1	b1	b4	c3
a2	b2	b1	c1
a2	b2	b2	c2
a2	b2	b4	c3
a3	b3	b1	c1
a3	b3	b2	c2
a3	b3	b4	c3

### mysql中in和exists的区别? \* \*

in和exists一般用于子查询。

- 使用exists时会先进行外表查询，将查询到的每行数据带入到内表查询中看是否满足条件；使用in一般会先进行内表查询获取结果集，然后对外表查询匹配结果集，返回数据。
- in在内表查询或者外表查询过程中都会用到索引。
- exists仅在内表查询时会用到索引
- 一般来说，当子查询的结果集比较大，外表较小使用exist效率更高；当子查询寻得结果集较小，外表较大时，使用in效率更高。
- 对于not in和not exists，not exists效率比not in的效率高，与子查询的结果集无关，因为not in对于内外表都进行了全表扫描，没有使用到索引。not exists的子查询中可以用到表上的索引。

### varchar和char的区别? \* \* \*

- varchar表示变长，char表示长度固定。当所插入的字符超过他们的长度时，在严格模式下，会拒绝插入并提示错误信息，在一般模式下，会截取后插入。如char(5)，无论插入的字符长度是多少，长度都是5，插入字符长度小于5，则用空格补充。对于varchar(5)，如果插入的字符长度小于5，则存储的字符长度就是插入字符的长度，不会填充。
- 存储容量不同，对于char来说，最多能存放的字符个数为255。对于varchar，最多能存放的字符个数是65532。
- 存储速度不同，char长度固定，存储速度会比varchar快一些，但在空间上会占用额外的空间，属于一种空间换时间的策略。而varchar空间利用率会高些，但存储速度慢，属于一种时间换空间的策略。

### MySQL中int(10)和char(10)和varchar(10)的区别? \* \* \*

int(10)中的10表示的是显示数据的长度，而char(10)和varchar(10)表示的是存储数据的大小。

### drop、 delete和truncate的区别? \* \*

	<b>drop</b>	<b>delete</b>	<b>truncate</b>
速度	快	逐行删除，慢	较快
类型	DDL	DML	DDL
回滚	不可回滚	可回滚	不可回滚
删除内容	删除整个表，数据行、索引都会被删除	表结构还在，删除表的一部分或全部数据	表结构还在，删除表的全部数据

一般来讲，删除整个表，使用drop，删除表的部分数据使用delete，保留表结构删除表的全部数据使用truncate。

## **UNION和UNION ALL的区别?** \* \*

union和union all的作用都是将两个结果集合并到一起。

- union会对结果去重并排序，union all直接返回合并后的结果，不去重也不进行排序。
- union all的性能比union性能好。

## **什么是临时表，什么时候会使用到临时表，什么时候删除临时表?** \*

MySQL在执行SQL语句的时候会临时创建一些存储中间结果集的表，这种表被称为临时表，临时表只对当前连接可见，在连接关闭后，临时表会被删除并释放空间。

临时表主要分为内存临时表和磁盘临时表两种。内存临时表使用的是MEMORY存储引擎，磁盘临时表使用的是MyISAM存储引擎。

一般在以下几种情况中会使用到临时表：

- FROM中的子查询
- DISTINCT查询并加上ORDER BY
- ORDER BY和GROUP BY的子句不一样时会产生临时表
- 使用UNION查询会产生临时表

## **大表数据查询如何进行优化?** \* \* \*

- 索引优化
- SQL语句优化
- 水平拆分
- 垂直拆分
- 建立中间表
- 使用缓存技术
- 固定长度的表访问起来更快
- 越小的列访问越快

## **了解慢日志查询吗？统计过慢查询吗？对慢查询如何优化？** \* \* \*

慢查询一般用于记录执行时间超过某个临界值的SQL语句的日志。

相关参数：

- slow\_query\_log：是否开启慢日志查询，1表示开启，0表示关闭。
- slow\_query\_log\_file：MySQL数据库慢查询日志存储路径。
- long\_query\_time：慢查询阈值，当SQL语句查询时间大于阈值，会被记录在日志上。
- log\_queries\_not\_using\_indexes：未使用索引的查询会被记录到慢查询日志中。
- log\_output：日志存储方式。“FILE”表示将日志存入文件。“TABLE”表示将日志存入数据库。

如何对慢查询进行优化?

- 分析语句的执行计划，查看SQL语句的索引是否命中
- 优化数据库的结构，将字段很多的表分解成多个表，或者考虑建立中间表。
- 优化LIMIT分页。

## 为什么要设置主键? \* \*

主键是唯一区分表中每一行的唯一标识，如果没有主键，更新或者删除表中特定的行会很困难，因为不能唯一准确地标识某一行。

## 主键一般用自增ID还是UUID? \* \*

使用自增ID的好处：

- 字段长度较uuid会小很多。
- 数据库自动编号，按顺序存放，利于检索
- 无需担心主键重复问题

使用自增ID的缺点：

- 因为是自增，在某些业务场景下，容易被其他人查到业务量。
- 发生数据迁移时，或者表合并时会非常麻烦
- 在高并发的场景下，竞争自增锁会降低数据库的吞吐能力

UUID：通用唯一标识码，UUID是基于当前时间、计数器和硬件标识等数据计算生成的。

使用UUID的优点：

- 唯一标识，不会考虑重复问题，在数据拆分、合并时也能达到全局的唯一性。
- 可以在应用层生成，提高数据库的吞吐能力。
- 无需担心业务量泄露的问题。

使用UUID的缺点：

- 因为UUID是随机生成的，所以会发生随机IO，影响插入速度，并且会造成硬盘的使用率较低。
- UUID占用空间较大，建立的索引越多，造成的影响越大。
- UUID之间比较大小较自增ID慢不少，影响查询速度。

最后说下结论，一般情况MySQL推荐使用自增ID。因为在MySQL的InnoDB存储引擎中，主键索引是一种聚簇索引，主键索引的B+树的叶子节点按照顺序存储了主键值及数据，如果主键索引是自增ID，只需要按顺序往后排列即可，如果是UUID，ID是随机生成的，在数据插入时会造成大量的数据移动，产生大量的内存碎片，造成插入性能的下降。

## 字段为什么要设置成not null? \* \*

首先说一点，NULL和空值是不一样的，空值是不占用空间的，而NULL是占用空间的，所以字段设为NOT NULL后仍然可以插入空值。

字段设置成not null主要有以下几点原因：

- NULL值会影响一些函数的统计，如count，遇到NULL值，这条记录不会统计在内。
- B树不存储NULL，所以索引用不到NULL，会造成第一点中说的统计不到的问题。
- NOT IN子查询在有NULL值的情况下返回的结果都是空值。

例如user表如下

<b>id</b>	<b>username</b>
0	zhangsan
1	lisi
2	null

```
select * from `user` where username NOT IN (select username from `user` where id != 0), 这条查询语句应该查到zhangsan这条数据，但是结果显示为null。
```

- MySQL在进行比较的时候，NULL会参与字段的比较，因为NULL是一种比较特殊的数据类型，数据库在处理时需要进行特数处理，增加了数据库处理记录的复杂性。

## 如何优化查询过程中的数据访问? \* \* \*

从减少数据访问方面考虑：

- 正确使用索引，尽量做到索引覆盖
- 优化SQL执行计划

从返回更少的数据方面考虑：

- 数据分页处理
- 只返回需要的字段

从减少服务器CPU开销方面考虑：

- 合理使用排序
- 减少比较的操作
- 复杂运算在客户端处理

从增加资源方面考虑：

- 客户端多进程并行访问
- 数据库并行处理

## 如何优化长难的查询语句? \* \*

- 将一个大的查询分解为多个小的查询
- 分解关联查询，使缓存的效率更高

## 如何优化LIMIT分页? \* \*

- 在LIMIT偏移量较大的时候，查询效率会变低，可以记录每次取出的最大ID，下次查询时可以利用ID进行查询
- 建立复合索引

## 如何优化UNION查询 \* \*

如果不需对结果集进行去重或者排序建议使用UNION ALL，会好一些。

## 如何优化WHERE子句 \* \* \*

- 不要在where子句中使用!=和<>进行不等于判断，这样会导致放弃索引进行全表扫描。
- 不要在where子句中使用null或空值判断，尽量设置字段为not null。
- 尽量使用union all代替or
- 在where和order by涉及的列建立索引
- 尽量减少使用in或者not in，会进行全表扫描

- 在where子句中使用参数会导致全表扫描
- 避免在where子句中对字段及进行表达式或者函数操作会导致存储引擎放弃索引进而全表扫描

## SQL语句执行的很慢原因是什么? \* \* \*

- 如果SQL语句只是偶尔执行很慢, 可能是执行的时候遇到了锁, 也可能是redo log日志写满了, 要将redo log中的数据同步到磁盘中去。
- 如果SQL语句一直都很慢, 可能是字段上没有索引或者字段有索引但是没用上索引。

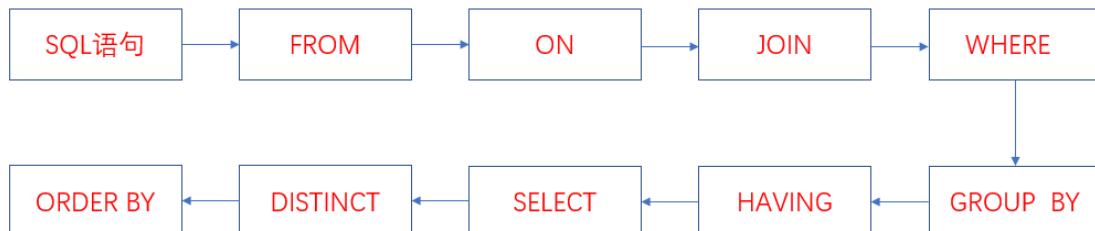
## SQL语句的执行顺序? \*

```

SELECT DISTINCT
    select_list
FROM
    left_table
LEFT JOIN
    right_table ON join_condition
WHERE
    where_condition
GROUP BY
    group_by_list
HAVING
    having_condition
ORDER BY
    order_by_condition

```

执行顺序如下:



- FROM: 对SQL语句执行查询时, 首先对关键字两边的表以笛卡尔积的形式执行连接, 并产生一个虚表V1。虚表就是视图, 数据会来自多张表的执行结果。
- ON: 对FROM连接的结果进行ON过滤, 并创建虚表V2
- JOIN: 将ON过滤后的左表添加进来, 并创建新的虚拟表V3
- WHERE: 对虚拟表V3进行WHERE筛选, 创建虚拟表V4
- GROUP BY: 对V4中的记录进行分组操作, 创建虚拟表V5
- HAVING: 对V5进行过滤, 创建虚拟表V6
- SELECT: 将V6中的结果按照SELECT进行筛选, 创建虚拟表V7
- DISTINCT: 对V7表中的结果进行去重操作, 创建虚拟表V8, 如果使用了GROUP BY子句则无需使用DISTINCT, 因为分组的时候是将列中唯一的值分成一组, 并且每组只返回一行记录, 所有的记录都是不同的。
- ORDER BY: 对V8表中的结果进行排序。

# 数据库优化

## 大表如何优化? \* \* \*

- 限定数据的范围：避免不带任何限制数据范围条件的查询语句。
- 读写分离：主库负责写，从库负责读。
- 垂直分表：将一个表按照字段分成多个表，每个表存储其中一部分字段。
- 水平分表：在同一个数据库内，把一个表的数据按照一定规则拆分到多个表中。
- 对单表进行优化：对表中的字段、索引、查询SQL进行优化。
- 添加缓存

## 什么是垂直分表、垂直分库、水平分表、水平分库? \* \* \*

垂直分表：将一个表按照字段分成多个表，每个表存储其中一部分字段。一般会将常用的字段放到一个表中，将不常用的字段放到另一个表中。

垂直分表的优势：

- 避免IO竞争减少锁表的概率。因为大的字段效率更低，第一数据量大，需要的读取时间长。第二，大字段占用的空间更大，单页内存存储的行数变少，会使得IO操作增多。
- 可以更好地提升热门数据的查询效率。

垂直分库：按照业务对表进行分类，部署到不同的数据库上面，不同的数据库可以放到不同的服务器上面。

垂直分库的优势：

- 降低业务中的耦合，方便对不同的业务进行分级管理。
- 可以提升IO、数据库连接数、解决单机硬件资源的瓶颈问题。

垂直拆分（分库、分表）的缺点：

- 主键出现冗余，需要管理冗余列
- 事务的处理变得复杂
- 仍然存在单表数据量过大的问题

水平分表：在同一个数据库内，把同一个表的数据按照一定规则拆分到多个表中。

水平分表的优势：

- 解决了单表数据量过大的问题
- 避免IO竞争并减少锁表的概率

水平分库：把同一个表的数据按照一定规则拆分到不同的数据库中，不同的数据库可以放到不同的服务器上。

水平分库的优势：

- 解决了单库大数据量的瓶颈问题
- IO冲突减少，锁的竞争减少，某个数据库出现问题不影响其他数据库（可用性），提高了系统的稳定性和可用性

水平拆分（分表、分库）的缺点：

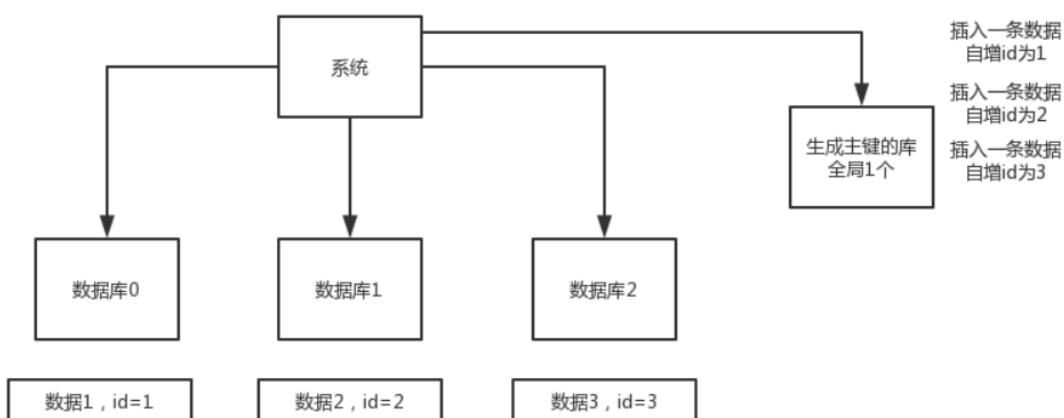
- 分片事务一致性难以解决
- 跨节点JOIN性能差，逻辑会变得复杂
- 数据扩展难度大，不易维护

在系统设计时应根据业务耦合来确定垂直分库和垂直分表的方案，在数据访问压力不是特别大时应考虑缓存、读写分离等方法，若数据量很大，或持续增长可考虑水平分库分表，水平拆分所涉及的逻辑比较复杂，常见的方案有客户端架构和恶代理架构。

## 分库分表后，ID键如何处理？ \* \* \*

分库分表后不能每个表的ID都是从1开始，所以需要一个全局ID，设置全局ID主要有以下几种方法：

- UUID：优点：本地生成ID，不需要远程调用；全局唯一不重复。缺点：占用空间大，不适合作为索引。
- 数据库自增ID：在分库分表后使用数据库自增ID，需要一个专门用于生成主键的库，每次服务接收到请求，先向这个库中插入一条没有意义的数据，获取一个数据库自增的ID，利用这个ID去分库分表中写数据。优点：简单易实现。缺点：在高并发下存在瓶颈。系统结构如下图（图片来源于网络）



- Redis生成ID：优点：不依赖数据库，性能比较好。缺点：引入新的组件会使得系统复杂度增加
- Twitter的snowflake算法：是一个64位的long型的ID，其中有1bit是不用的，41bit作为毫秒数，10bit作为工作机器ID，12bit作为序列号。
  - 1bit：第一个bit默认为0，因为二进制中第一个bit为1的话为负数，但是ID不能为负数。
  - 41bit：表示的是时间戳，单位是毫秒。
  - 10bit：记录工作机器ID，其中5个bit表示机房ID，5个bit表示机器ID。
  - 12bit：用来记录同一毫秒内产生的不同ID。
- 美团的Leaf分布式ID生成系统，[美团点评分布式ID生成系统](#)

## MySQL的复制原理及流程？如何实现主从复制？ \* \* \*

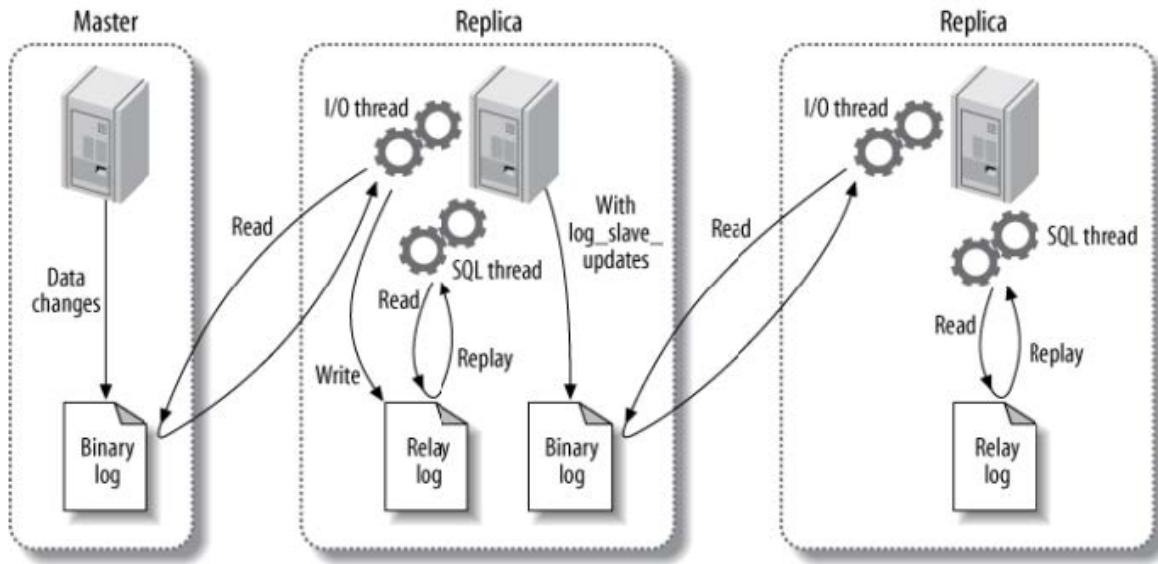
MySQL复制：为保证主服务器和从服务器的数据一致性，在向主服务器插入数据后，从服务器会自动将主服务器中修改的数据同步过来。

主从复制的原理：

主从复制主要有三个线程：binlog线程，I/O线程，SQL线程。

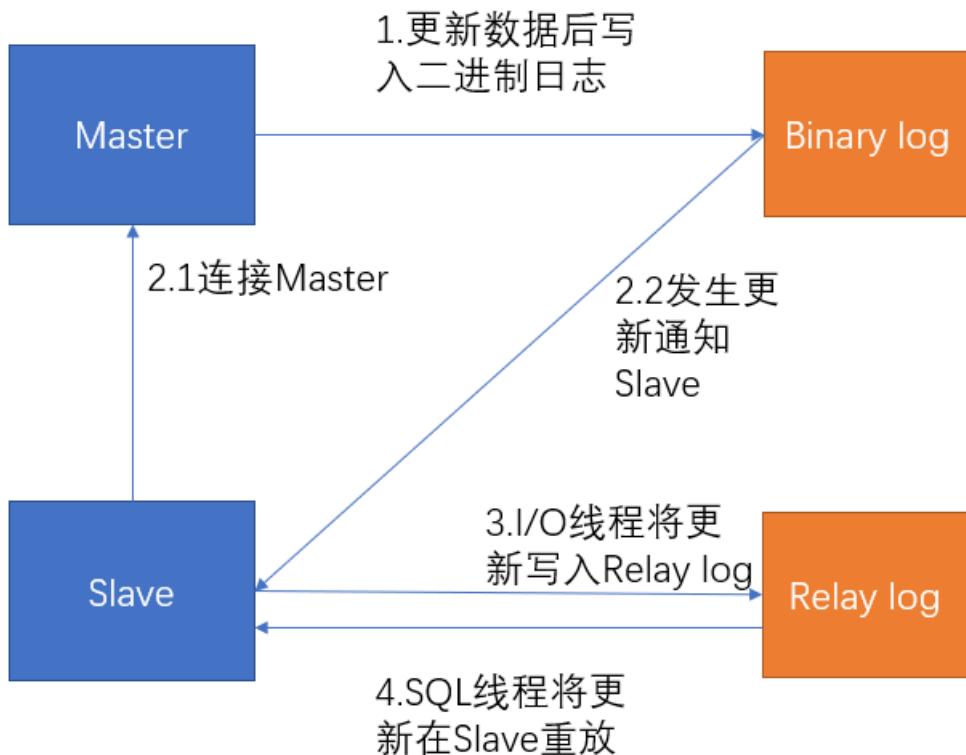
- binlog线程：负责将主服务器上的数据更改写入到二进制日志（Binary log）中。
- I/O线程：负责从主服务器上读取二进制日志（Binary log），并写入从服务器的中继日志（Relay log）中。
- SQL线程：负责读取中继日志，解析出主服务器中已经执行的数据更改并在从服务器中重放

复制过程如下（图片来源于网络）：



1. Master在每个事务更新数据完成之前，将操作记录写入到binlog中。
2. Slave从库连接Master主库，并且Master有多少个Slave就会创建多少个binlog dump线程。当Master节点的binlog发生变化时，binlog dump会通知所有的Slave，并将相应的binlog发送给Slave。
3. I/O线程接收到binlog内容后，将其写入到中继日志（Relay log）中。
4. SQL线程读取中继日志，并在从服务器中重放。

这里补充一个通俗易懂的图。



主从复制的作用：

- 高可用和故障转移
- 负载均衡
- 数据备份
- 升级测试

## 了解读写分离吗? \* \* \*

读写分离主要依赖于主从复制，主从复制为读写分离服务。

读写分离的优势：

- 主服务器负责写，从服务器负责读，缓解了锁的竞争
- 从服务器可以使用MyISAM，提升查询性能及节约系统开销
- 增加冗余，提高可用性

关注公众号“路人zhang”，获取更多面试资料

关注知乎“路人zhang”，聊聊码农那些事

