大家好，我是大彬。最近在面试，看了很多面经，饱受Java八股文的折磨。所以我将常见的Java并发编程常见面试题总结了一下，汇集成**面试手册**。这些题目都是我自己整理的，不是网上复制粘贴来的，希望对小伙伴们有所帮助！

此外，我建了一个**互联网求职交流群，群里会分享各大互联网公司校招资讯、面试和笔试题目，有需要的小伙伴可以扫码进群。**

群二维码失效的话，可以添加我的微信号（i_am_dabin）或者扫下面的二维码，加我微信，我拉你进群~

Java核心知识

1. Java的特点

Java是一门面向对象的编程语言。面向对象和面向过程的区别参考下一个问题。

Java具有平台独立性和移植性。

- Java有一句口号：Write once, run anywhere，一次编写、到处运行。这也是Java的魅力所在。而实现这种特性的正是Java虚拟机JVM。已编译的Java程序可以在任何带有JVM的平台上运行。你可以在windows平台编写代码，然后拿到linux上运行。只要你在编写完代码后，将代码编译成.class文件，再把class文件打成Java包，这个jar包就可以在不同的平台上运行了。

Java具有稳健性。

- Java是一个强类型语言，它允许扩展编译时检查潜在类型不匹配问题的功能。Java要求显式的方法声明，它不支持C风格的隐式声明。这些严格的要求保证编译程序能捕捉调用错误，这就导致更可靠的程序。
- 异常处理是Java中使得程序更稳健的另一个特征。异常是某种类似于错误的异常条件出现的信号。使用try/catch/finally语句，程序员可以找到出错的处理代码，这就简化了出错处理和恢复的任务。

2. Java 与 C++ 的区别

- Java 是纯粹的面向对象语言，所有的对象都继承自 java.lang.Object，C++ 兼容 C ，不但支持面向对象也支持面向过程。
- Java 通过虚拟机从而实现跨平台特性， 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。

> What are the main differences between Java and C++?

3. 面向对象和面向过程的区别?

面向对象和面向过程是一种软件开发思想。

- 面向过程就是分析出解决问题所需要的步骤，然后用函数按这些步骤实现，使用的时候依次调用就可以了。
- 面向对象是把构成问题事务分解成各个对象，分别设计这些对象，然后将他们组装成有完整功能的系统。面向过程只用函数实现，面向对象是用类实现各个功能模块。

以五子棋为例（来源网络），面向过程的设计思路就是首先分析问题的步骤： 1、开始游戏，2、黑子先走，3、绘制画面，4、判断输赢，5、轮到白子，6、绘制画面，7、判断输赢，8、返回步骤2，9、输出最后结果。 把上面每个步骤用分别的函数来实现，问题就解决了。

而面向对象的设计则是从另外的思路来解决问题。整个五子棋可以分为：

1. 黑白双方
2. 棋盘系统，负责绘制画面
3. 规则系统，负责判定诸如犯规、输赢等。

黑白双方负责接受用户的输入，并告知棋盘系统棋子布局发生变化，棋盘系统接收到了棋子的变化的信息就负责在屏幕上面显示出这种变化，同时利用规则系统来对棋局进行判定。

4. JKD和JRE的区别?

JDK和JRE是Java开发和运行工具，其中JDK包含了JRE，而JRE是可以独立安装的。

JDK：Java Development Kit，JAVA语言的软件工具开发包，是整个JAVA开发的核心，它包含了JAVA的运行（JVM+JAVA类库）环境和JAVA工具。

JRE：Java Runtime Environment，Java运行环境，包含JVM标准实现及Java核心类库。JRE是Java运行环境，并不是一个开发环境，所以没有包含任何开发工具（如编译器和调试器）。

JRE是运行基于Java语言编写的程序所不可缺少的运行环境。也是通过它，Java的开发者才得以将自己开发的程序发布到用户手中，让用户使用。

5. 面向对象有哪些特性?

面向对象四大特性：封装，继承，多态，抽象

- 封装就是将类的信息隐藏在类内部，不允许外部程序直接访问，而是通过该类的方法实现对隐藏信息的操作和访问。 良好的封装能够减少耦合。
- 继承是从已有的类中派生出新的类，新的类继承父类的属性和行为，并能扩展新的能力，大大增加程序的重用性和易维护性。在Java中是单继承的，也就是说一个子类只有一个父类。
- 多态是同一个行为具有多个不同表现形式的能力。在不修改程序代码的情况下改变程序运行时绑定的代码。实现多态的三要素：继承、重写、父类引用指向子类对象。 静态多态性：通过重载实现，相同的方法有不同的参数列表，可以根据参数的不同，做出不同的处理。 动态多态性：在子类中重写父类的方法。运行期间判断所引用对象的实际类型，根据其实际类型调用相应的方法。
- 抽象。把客观事物用代码抽象出来。

6. Java的基本数据类型有哪些?

- byte，8bit
- char，16bit
- short，16bit
- int，32bit

- float，32bit
- long，64bit
- double，64bit
- boolean，只有两个值：true、false，可以使用用 1 bit 来存储

简单类型	boolean	byte	char	short	Int	long	float	double
二进制位数	1	8	16	16	32	64	32	64
包装类	Boolean	Byte	Character	Short	Integer	Long	Float	Double

7. 什么是值传递和引用传递?

- 值传递是对基本型变量而言的,传递的是该变量的一个副本，改变副本不影响原变量。

  引用传递一般是对于对象型变量而言的,传递的是该对象地址的一个副本, 并不是原对象本身 。所以对引用对象进行操作会同时改变原对象。

8. 自动装箱和拆箱

Java中基础数据类型与它们对应的包装类见下表：

原始类型	包装类型
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

装箱：将基础类型转化为包装类型。

拆箱：将包装类型转化为基础类型。

当基础类型与它们的包装类有如下几种情况时，编译器会**自动**帮我们进行装箱或拆箱：

- 赋值操作（装箱或拆箱）
- 进行加减乘除混合运算 （拆箱）
- 进行>,<,==比较运算（拆箱）
- 调用equals进行比较（装箱）
- ArrayList、HashMap等集合类添加基础类型数据时（装箱）

示例代码：

```
Integer x = 1; // 装箱 调用 Integer.valueOf(1)
int y = x; // 拆箱 调用了 X.intValue()
```

下面看一道常见的面试题：

```
public void testAutoBox() {
     int a = 100;
     Integer b = 100;
     System.out.println(a == b);

     Integer c = 100;
     Integer d = 100;
     System.out.println(c == d);

     Integer e = 200;
     Integer f = 200;
     System.out.println(e == f);
}
```

输出：

```
true
true
false
```

为什么第三个输出是false？看看 Integer 类的源码就知道啦。

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

`Integer e = 200;` 会调用 调用`Integer.valueOf(200)`。而从Integer的valueOf()源码可以看到，这里的实现并不是简单的new Integer，而是用IntegerCache做一个cache。

```
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;
    }
    ...
}
```

这是IntegerCache静态代码块中的一段，默认Integer cache 的下限是-128，上限默认127。当赋值100给Integer时，刚好在这个范围内，所以从cache中取对应的Integer并返回，所以二次返回的是同一个对象，所以==比较是相等的，当赋值200给Integer时，不在cache 的范围内，所以会new Integer并返回，当然==比较的结果是不相等的。

9. String 为什么不可变?

先看下Java8 String类的源码：

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];

    /** Cache the hash code for the string */
    private int hash; // Default to 0
}
```

String类是final的，它的所有成员变量也都是final的。为什么是final的?

1. **线程安全。**同一个字符串实例可以被多个线程共享，因为字符串不可变，本身就是线程安全的。
2. **支持hash映射和缓存。**因为String的hash值经常会使用到，比如作为 Map 的键，不可变的特性使得 hash 值也不会变，不需要重新计算。
3. **字符串常量池优化。**String对象创建之后，会缓存到字符串常量池中，下次需要创建同样的对象时，可以直接返回缓存的引用。

10. String, StringBuffer 和 StringBuilder区别

1. 可变性

- String 不可变
- StringBuffer 和 StringBuilder 可变

2. 线程安全

- String 不可变，因此是线程安全的
- StringBuilder 不是线程安全的
- StringBuffer 是线程安全的，内部使用 synchronized 进行同步

11. String 类的常用方法有哪些?

- indexOf()：返回指定字符的索引。
- charAt()：返回指定索引处的字符。
- replace()：字符串替换。
- 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。
- getBytes()：返回字符串的 byte 类型数组。
- length()：返回字符串长度。
- toLowerCase()：将字符串转成小写字母。

- toUpperCase()：将字符串转成大写字符。
- substring()：截取字符串。
- equals()：字符串比较。

12. new String("dabin")会创建几个对象?

使用这种方式会创建两个字符串对象（前提是字符串常量池中没有 "dabin" 这个字符串对象）。

- "dabin" 属于字符串字面量，因此编译时期会在字符串常量池中创建一个字符串对象，指向这个 "dabin" 字符串字面量；
- 使用 new 的方式会在堆中创建一个字符串对象。

13. 什么是字符串常量池?

字符串常量池（String Pool）保存着所有字符串字面量，这些字面量在编译时期就确定。字符串常量池位于堆内存中，专门用来存储字符串常量。在创建字符串时，JVM首先会检查字符串常量池，如果该字符串已经存在池中，则返回其引用，如果不存在，则创建此字符串并放入池中，并返回其引用。

14. object常用方法有哪些?

Java面试经常会出现的一道题目，Object的常用方法。下面给大家整理一下。

object常用方法有：toString()、equals()、hashCode()、clone()等。

toString

默认输出对象地址。

```
public class Person {
    private int age;
    private String name;

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

    public static void main(String[] args) {
        System.out.println(new Person(18, "程序员大彬").toString());
    }
    //output
    //me.tyson.java.core.Person@4554617c
}
```

可以重写toString方法，按照重写逻辑输出对象值。

```
public class Person {
    private int age;
    private String name;

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

    @Override
    public String toString() {
```

```
        return name + ":" + age;
    }

    public static void main(String[] args) {
        System.out.println(new Person(18, "程序员大彬").toString());
    }
    //output
    //程序员大彬:18
}
```

equals

默认比较两个引用变量是否指向同一个对象（内存地址）。

```
public class Person {
    private int age;
    private String name;

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

    public static void main(String[] args) {
        String name = "程序员大彬";
        Person p1 = new Person(18, name);
        Person p2 = new Person(18, name);

        System.out.println(p1.equals(p2));
    }
    //output
    //false
}
```

可以重写equals方法，按照age和name是否相等来判断：

```
public class Person {
    private int age;
    private String name;

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

    @Override
    public boolean equals(Object o) {
        if (o instanceof Person) {
            Person p = (Person) o;
            return age == p.age && name.equals(p.name);
        }
        return false;
    }

    public static void main(String[] args) {
        String name = "程序员大彬";
        Person p1 = new Person(18, name);
```

```
        Person p2 = new Person(18, name);

        System.out.println(p1.equals(p2));
    }
    //output
    //true
}
```

hashCode

将与对象相关的信息映射成一个哈希值，默认的实现hashCode值是根据内存地址换算出来。

```
public class Cat {
    public static void main(String[] args) {
        System.out.println(new Cat().hashCode());
    }
    //out
    //1349277854
}
```

clone

java赋值是复制对象引用，如果我们想要得到一个对象的副本，使用赋值操作是无法达到目的的。
Object对象有个clone()方法，实现了对

象中各个属性的复制，但它的可见范围是protected的。

```
protected native Object clone() throws CloneNotSupportedException;
```

所以实体类使用克隆的前提是：

- 实现Cloneable接口，这是一个标记接口，自身没有方法，这应该是一种约定。调用clone方法时，会判断有没有实现Cloneable接口，没有实现Cloneable的话会抛异常CloneNotSupportedException。
- 覆盖clone()方法，可见性提升为public。

```
public class Cat implements Cloneable {
    private String name;

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public static void main(String[] args) throws CloneNotSupportedException {
        Cat c = new Cat();
        c.name = "程序员大彬";
        Cat cloneCat = (Cat) c.clone();
        c.name = "大彬";
        System.out.println(cloneCat.name);
    }
    //output
    //程序员大彬
}
```

getClass

返回此 Object 的运行时类，常用于java反射机制。

```
public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public static void main(String[] args) {
        Person p = new Person("程序员大彬");
        Class clz = p.getClass();
        System.out.println(clz);
        //获取类名
        System.out.println(clz.getName());
    }
    /**
     * class com.tyson.basic.Person
     * com.tyson.basic.Person
     */
}
```

wait

当前线程调用对象的wait()方法之后，当前线程会释放对象锁，进入等待状态。等待其他线程调用此对象的notify()/notifyAll()唤醒或者等待超时时间wait(long timeout)自动唤醒。线程需要获取obj对象锁之后才能调用 obj.wait()。

notity

obj.notify()唤醒在此对象上等待的单个线程，选择是任意性的。notifyAll()唤醒在此对象上等待的所有线程。

15. 深拷贝和浅拷贝?

浅拷贝：拷贝对象和原始对象的引用类型引用同一个对象。

以下例子，Cat对象里面有个Person对象，调用clone之后，克隆对象和原对象的Person引用的是同一个对象，这就是浅拷贝。

```
public class Cat implements Cloneable {
    private String name;
    private Person owner;

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public static void main(String[] args) throws CloneNotSupportedException {
        Cat c = new Cat();
        Person p = new Person(18, "程序员大彬");
        c.owner = p;

        Cat cloneCat = (Cat) c.clone();
        p.setName("大彬");
        System.out.println(cloneCat.owner.getName());
```

```
    }
    //output
    //大彬
}
```

深拷贝：拷贝对象和原始对象的引用类型引用不同的对象。

以下例子，在clone函数中不仅调用了super.clone，而且调用Person对象的clone方法（Person也要实现Cloneable接口并重写clone方法），从而实现了深拷贝。可以看到，拷贝对象的值不会受到原对象的影响。

```
public class Cat implements Cloneable {
    private String name;
    private Person owner;

    @Override
    protected Object clone() throws CloneNotSupportedException {
        Cat c = null;
        c = (Cat) super.clone();
        c.owner = (Person) owner.clone();//拷贝Person对象
        return c;
    }

    public static void main(String[] args) throws CloneNotSupportedException {
        Cat c = new Cat();
        Person p = new Person(18, "程序员大彬");
        c.owner = p;

        Cat cloneCat = (Cat) c.clone();
        p.setName("大彬");
        System.out.println(cloneCat.owner.getName());
    }
    //output
    //程序员大彬
}
```

16. 两个对象的hashCode()相同，则 equals()是否也一定为 true?

equals与hashcode的关系：

1. 如果两个对象调用equals比较返回true，那么它们的hashCode值一定要相同；
2. 如果两个对象的hashCode相同，它们并不一定相同。

hashcode方法主要是用来提升对象比较的效率，先进行hashcode()的比较，如果不相同，那就不必在进行equals的比较，这样就大大减少了equals比较的次数，当比较对象的数量很大的时候能提升效率。

之所以重写equals()要重写hashcode()，是为了保证equals()方法返回true的情况下hashcode值也要一致，如果重写了equals()没有重写hashcode()，就会出现两个对象相等但hashcode()不相等的情况。这样，当用其中的一个对象作为键保存到hashMap、hashTable或hashSet中，再以另一个对象作为键值去查找他们的时候，则会查找不到。

17. Java创建对象有几种方式?

Java创建对象有以下几种方式：

- 用new语句创建对象。

- 使用反射，使用Class.newInstance()创建对象。
- 调用对象的clone()方法。
- 运用反序列化手段，调用java.io.ObjectInputStream对象的readObject()方法。

18. 说说类实例化的顺序

Java中类实例化顺序：

1. 静态属性，静态代码块。
2. 普通属性，普通代码块。
3. 构造方法。

```
public class LifeCycle {
    // 静态属性
    private static String staticField = getStaticField();

    // 静态代码块
    static {
        System.out.println(staticField);
        System.out.println("静态代码块初始化");
    }

    // 普通属性
    private String field = getField();

    // 普通代码块
    {
        System.out.println(field);
        System.out.println("普通代码块初始化");
    }

    // 构造方法
    public LifeCycle() {
        System.out.println("构造方法初始化");
    }

    // 静态方法
    public static String getStaticField() {
        String statiFiled = "静态属性初始化";
        return statiFiled;
    }

    // 普通方法
    public String getField() {
        String filed = "普通属性初始化";
        return filed;
    }

    public static void main(String[] argc) {
        new LifeCycle();
    }

    /**
     *       静态属性初始化
     *       静态代码块初始化
     *       普通属性初始化
     *       普通代码块初始化
```

```
     *       构造方法初始化
     */
}
```

19. equals和==有什么区别?

- 对于基本数据类型，==比较的是他们的值。基本数据类型没有equal方法；
- 对于复合数据类型，==比较的是它们的存放地址(是否是同一个对象)。equals()默认比较地址值，重写的话按照重写逻辑去比较。

20. 常见的关键字有哪些?

static

static可以用来修饰类的成员方法、类的成员变量。

静态变量

static变量也称作静态变量，静态变量和非静态变量的区别是：静态变量被所有的对象所共享，在内存中只有一个副本，它当且仅当在类初次加载时会被初始化。而非静态变量是对象所拥有的，在创建对象的时候被初始化，存在多个副本，各个对象拥有的副本互不影响。

以下例子，age为非静态变量，则p1打印结果是：`Name:zhangsan, Age:10`；若age使用static修饰，则p1打印结果是：`Name:zhangsan, Age:12`，因为static变量在内存只有一个副本。

```
public class Person {
    String name;
    int age;

    public String toString() {
        return "Name:" + name + ", Age:" + age;
    }

    public static void main(String[] args) {
        Person p1 = new Person();
        p1.name = "zhangsan";
        p1.age = 10;
        Person p2 = new Person();
        p2.name = "lisi";
        p2.age = 12;
        System.out.println(p1);
        System.out.println(p2);
    }
    /**Output
     * Name:zhangsan, Age:10
     * Name:lisi, Age:12
     *///~
}
```

静态方法

static方法一般称作静态方法。静态方法不依赖于任何对象就可以进行访问，通过类名即可调用静态方法。

```
public class Utils {
    public static void print(String s) {
        System.out.println("hello world: " + s);
    }

    public static void main(String[] args) {
        Utils.print("程序员大彬");
    }
}
```

静态代码块

静态代码块只会在类加载的时候执行一次。以下例子，startDate和endDate在类加载的时候进行赋值。

```
class Person  {
    private Date birthDate;
    private static Date startDate, endDate;
    static{
        startDate = Date.valueOf("2008");
        endDate = Date.valueOf("2021");
    }

    public Person(Date birthDate) {
        this.birthDate = birthDate;
    }
}
```

静态内部类

在静态方法里，使用非静态内部类依赖于外部类的实例，也就是说需要先创建外部类实例，才能用这个实例去创建非静态内部类。而静态内部类不需要。

```
public class OuterClass {
    class InnerClass {
    }
    static class StaticInnerClass {
    }
    public static void main(String[] args) {
        // 在静态方法里，不能直接使用OuterClass.this去创建InnerClass的实例
        // 需要先创建OuterClass的实例o，然后通过o创建InnerClass的实例
        // InnerClass innerClass = new InnerClass();
        OuterClass outerClass = new OuterClass();
        InnerClass innerClass = outerClass.new InnerClass();
        StaticInnerClass staticInnerClass = new StaticInnerClass();

        outerClass.test();
    }

    public void nonStaticMethod() {
        InnerClass innerClass = new InnerClass();
        System.out.println("nonStaticMethod...");
    }
}
```

final

1. **基本数据**类型用final修饰，则不能修改，是常量；**对象引用**用final修饰，则引用只能指向该对象，不能指向别的对象，但是对象本身可以修改。
2. final修饰的方法不能被子类重写
3. final修饰的类不能被继承。

this

`this.属性名称`指访问类中的成员变量，可以用来区分成员变量和局部变量。如下代码所示，`this.name`访问类Person当前实例的变量。

```
/**
 * @description:
 * @author: 程序员大彬
 * @time: 2021-08-17 00:29
 */
public class Person {
    String name;
    int age;

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

`this.方法名称`用来访问本类的方法。以下代码中，`this.born()`调用类 Person 的当前实例的方法。

```
/**
 * @description:
 * @author: 程序员大彬
 * @time: 2021-08-17 00:29
 */
public class Person {
    String name;
    int age;

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

    void born() {
    }
}
```

super

super 关键字用于在子类中访问父类的变量和方法。

```
class A {
    protected String name = "大彬";

    public void getName() {
        System.out.println("父类:" + name);
    }
```

```
}

public class B extends A {
    @Override
    public void getName() {
        System.out.println(super.name);
        super.getName();
    }

    public static void main(String[] args) {
        B b = new B();
        b.getName();
    }
    /**
     * 大彬
     * 父类:大彬
     */
}
```

在子类B中，我们重写了父类的getName()方法，如果在重写的getName()方法中我们要调用父类的相同方法，必须要通过super关键字显式指出。

21. final, finally, finalize 的区别

- final 用于修饰属性,方法和类, 分别表示属性不能被重新赋值, 方法不可被覆盖, 类不可被继承。
- finally 是异常处理语句结构的一部分，一般以ty-catch-finally出现，finally代码块表示总是被执行。
- finalize 是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()方法，回收垃圾，JVM并不保证此方法总被调用。

22. final关键字的作用?

- final 修饰的类不能被继承。
- final 修饰的方法不能被重写。
- final 修饰的变量叫常量，常量必须初始化，初始化之后值就不能被修改。

23. 方法重载和重写的区别?

同个类中的多个方法可以有相同的方法名称，但是有不同的参数列表，这就称为方法重载。参数列表又叫参数签名，包括参数的类型、参数的个数、参数的顺序，只要有一个不同就叫做参数列表不同。

重载是面向对象的一个基本特性。

```
public class OverrideTest {
    void setPerson() { }

    void setPerson(String name) {
        //set name
    }

    void setPerson(String name, int age) {
        //set name and age
    }
}
```

方法的重写描述的是父类和子类之间的。当父类的功能无法满足子类的需求，可以在子类对方法进行重写。方法重写时， 方法名与形参列表必须一致。

如下代码，Person为父类，Student为子类，在Student中重写了dailyTask方法。

```
public class Person {
    private String name;

    public void dailyTask() {
        System.out.println("work eat sleep");
    }
}


public class Student extends Person {
    @Override
    public void dailyTask() {
        System.out.println("study eat sleep");
    }
}
```

24. 接口与抽象类区别?

- 语法层面上 1）抽象类可以有方法实现，而接口的方法中只能是抽象方法； 2）抽象类中的成员变量可以是各种类型的，接口中的成员变量只能是public static final类型； 3）接口中不能含有静态代码块以及静态方法，而抽象类可以有静态代码块和静态方法； 4）一个类只能继承一个抽象类，而一个类却可以实现多个接口。
- 设计层面上的区别 1）抽象层次不同。抽象类是对整个类整体进行抽象，包括属性、行为，但是接口只是对类行为进行抽象。继承抽象类是一种"是不是"的关系，而接口实现则是 "有没有"的关系。如果一个类继承了某个抽象类，则子类必定是抽象类的种类，而接口实现则是具备不具备的关系，比如鸟是否能飞。 2） 继承抽象类的是具有相似特点的类，而实现接口的却可以不同的类。

  门和警报的例子：

  ```
  class AlarmDoor extends Door implements Alarm {
      //code
  }

  class BMWCar extends Car implements Alarm {
      //code
  }
  ```

25. 常见的Exception有哪些?

常见的RuntimeException：

1. ClassCastException //类型转换异常
2. IndexOutOfBoundsException //数组越界异常
3. NullPointerException //空指针
4. ArrayStoreException //数组存储异常
5. NumberFormatException //数字格式化异常
6. ArithmeticException //数学运算异常

unchecked Exception：

1. NoSuchFieldException //反射异常，没有对应的字段

2. ClassNotFoundException //类没有找到异常
3. IllegalAccessException //安全权限异常，可能是反射时调用了private方法

26. Error和Exception的区别?

Error：JVM 无法解决的严重问题，如栈溢出（StackOverflowError）、内存溢出（OOM）等。程序无法处理的错误。

Exception：其它因编程错误或偶然的外在因素导致的一般性问题。可以在代码中进行处理。如：空指针异常、数组下标越界等。

27. 运行时异常和非运行时异常（checked）的区别?

unchecked exception包括RuntimeException和Error类，其他所有异常称为检查（checked）异常。

1. RuntimeException由程序错误导致，应该修正程序避免这类异常发生。
2. checked Exception由具体的环境（读取的文件不存在或文件为空或sql异常）导致的异常。必须进行处理，不然编译不通过，可以catch或者throws。

28. throw和throws的区别?

- **throw**：用于抛出一个具体的异常对象。
- **throws**：用在方法签名中，用于声明该方法可能抛出的异常。子类方法抛出的异常范围更加小，或者根本不抛异常。

29. BIO/NIO/AIO区别的区别?

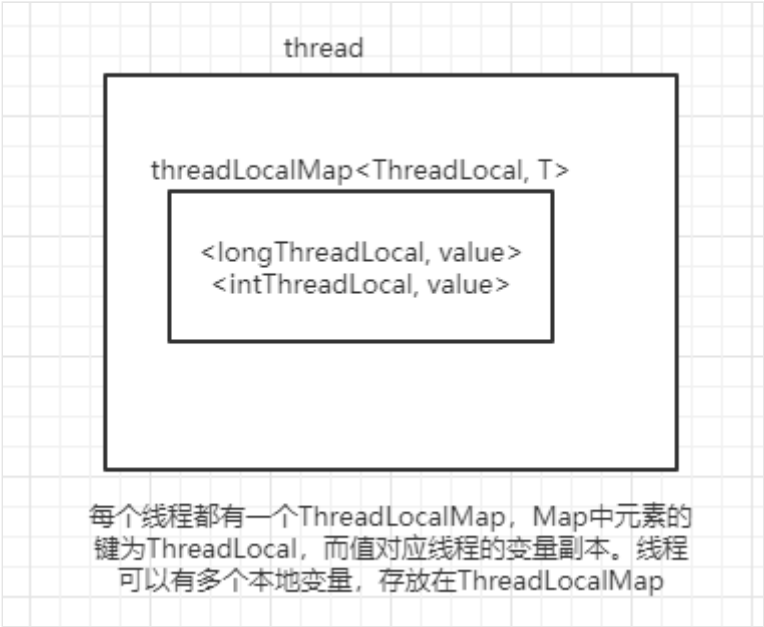
同步阻塞IO : 用户进程发起一个IO操作以后，必须等待IO操作的真正完成后，才能继续运行。

同步非阻塞IO: 客户端与服务器通过Channel连接，采用多路复用器轮询注册的Channel。提高吞吐量和可靠性。用户进程发起一个IO操作以后，可做其它事情，但用户进程需要轮询IO操作是否完成，这样造成不必要的CPU资源浪费。

异步非阻塞IO: 非阻塞异步通信模式，NIO的升级版，采用异步通道实现异步通信，其read和write方法均是异步方法。用户进程发起一个IO操作，然后立即返回，等IO操作真正的完成以后，应用程序会得到IO操作完成的通知。类似Future模式。

30. ThreadLocal的实现原理?

每个线程都有一个ThreadLocalMap(ThreadLocal内部类)，Map中元素的键为ThreadLocal，而值对应线程的变量副本。



调用threadLocal.set()-->调用getMap(Thread)-->返回当前线程的ThreadLocalMap<ThreadLocal, value>-->map.set(this, value)，this是ThreadLocal

```
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}

ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}

void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

调用get()-->调用getMap(Thread)-->返回当前线程的ThreadLocalMap<ThreadLocal, value>-->map.getEntry(this)，返回value

```
    public T get() {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null) {
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null) {
                @SuppressWarnings("unchecked")
                T result = (T)e.value;
                return result;
            }
        }
        return setInitialValue();
    }
```

threadLocals的类型ThreadLocalMap的键为ThreadLocal对象，因为每个线程中可有多个threadLocal变量，如longLocal和stringLocal。

```
public class ThreadLocalDemo {
    ThreadLocal<Long> longLocal = new ThreadLocal<>();

    public void set() {
        longLocal.set(Thread.currentThread().getId());
    }
    public Long get() {
        return longLocal.get();
    }

    public static void main(String[] args) throws InterruptedException {
        ThreadLocalDemo threadLocalDemo = new ThreadLocalDemo();
        threadLocalDemo.set();
        System.out.println(threadLocalDemo.get());

        Thread thread = new Thread(() -> {
            threadLocalDemo.set();
            System.out.println(threadLocalDemo.get());
        }
        );

        thread.start();
        thread.join();

        System.out.println(threadLocalDemo.get());
    }
}
```

ThreadLocal 并不是用来解决共享资源的多线程访问的问题，因为每个线程中的资源只是副本，并不共享。因此ThreadLocal适合作为线程上下文变量，简化线程内传参。

31. 什么是fail fast?

fast-fail是Java集合的一种错误机制。当多个线程对同一个集合进行操作时，就有可能会产生fast-fail事件。 例如：当线程a正通过iterator遍历集合时，另一个线程b修改了集合的内容，此时modCount（记录集合操作过程的修改次数）会加1，不等于expectedModCount，那么线程a访问集合的时候，就会抛出ConcurrentModificationException，产生fast-fail事件。边遍历边修改集合也会产生fast-fail事件。

解决方法：

- 使用Colletions.synchronizedList方法或在修改集合内容的地方加上synchronized。这样的话，增删集合内容的同步锁会阻塞遍历操作，影响性能。
- 使用CopyOnWriteArrayList来替换ArrayList。在对CopyOnWriteArrayList进行修改操作的时候，会拷贝一个新的数组，对新的数组进行操作，操作完成后再把引用移到新的数组。

32. 守护线程是什么?

- 守护线程是运行在后台的一种特殊进程。
- 它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。
- 在 Java 中垃圾回收线程就是特殊的守护线程。

33. Java支持多继承吗?

java中，**类不支持**多继承。**接口才支持**多继承。接口的作用是拓展对象功能。当一个子接口继承了多个父接口时，说明子接口拓展了多个功能。当一个类实现该接口时，就拓展了多个的功能。

Java不支持多继承的原因（来源网络）：

- 安全性的考虑，如果子类继承的多个父类里面有相同的方法或者属性，子类将不知道具体要继承哪个。
- Java提供了接口和内部类以达到实现多继承功能，弥补单继承的缺陷。

34. 如何实现对象克隆?

- 实现 Cloneable 接口，重写 clone() 方法。这种方式是浅拷贝，即如果类中属性有自定义引用类型，只拷贝引用，不拷贝引用指向的对象。如果对象的属性的Class 也实现 Cloneable 接口，那么在克隆对象时也会克隆属性，即深拷贝。
- 结合序列化(JDK java.io.Serializable 接口、JSON格式、XML格式等)，深拷贝。
- 通过`org.apache.commons`中的工具类`BeanUtils`和`PropertyUtils`进行对象复制。

35. 同步和异步的区别?

同步：发出一个调用时，在没有得到结果之前，该调用就不返回。

异步：在调用发出后，被调用者返回结果之后会通知调用者，或通过回调函数处理这个调用。

36. 阻塞和非阻塞的区别?

阻塞和非阻塞关注的是线程的状态。

阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会恢复运行。

非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

> 举个例子，理解下同步、阻塞、异步、非阻塞的区别：
>
> 同步就是烧开水，要自己来看开没开；异步就是水开了，然后水壶响了通知你水开了（回调通知）。阻塞是烧开水的过程中，你不能干其他事情，必须在旁边等着；非阻塞是烧开水的过程里可以干其他事情。

37. Java8的新特性有哪些?

- Lambda 表达式：Lambda允许把函数作为一个方法的参数
- Stream API ： 新添加的Stream API（java.util.stream） 把真正的函数式编程风格引入到Java中
- 默认方法：默认方法就是一个在接口里面有了一个实现的方法。
- Optional 类 ： Optional 类已经成为 Java 8 类库的一部分，用来解决空指针异常。
- Date Time API ： 加强对日期与时间的处理。

> Java8 新特性总结

38. 什么是序列化和反序列化?

序列化：把内存中的对象转换为字节序列的过程。

反序列化：把字节序列恢复为Java对象的过程。

39. 如何实现序列化

实现Serializable接口即可。序列化的时候（如objectOutputStream.writeObject(user)），会判断user是否实现了Serializable（obj instanceof Serializable），如果对象没有实现Serializable接口，在序列化的时候会抛出NotSerializableException异常。

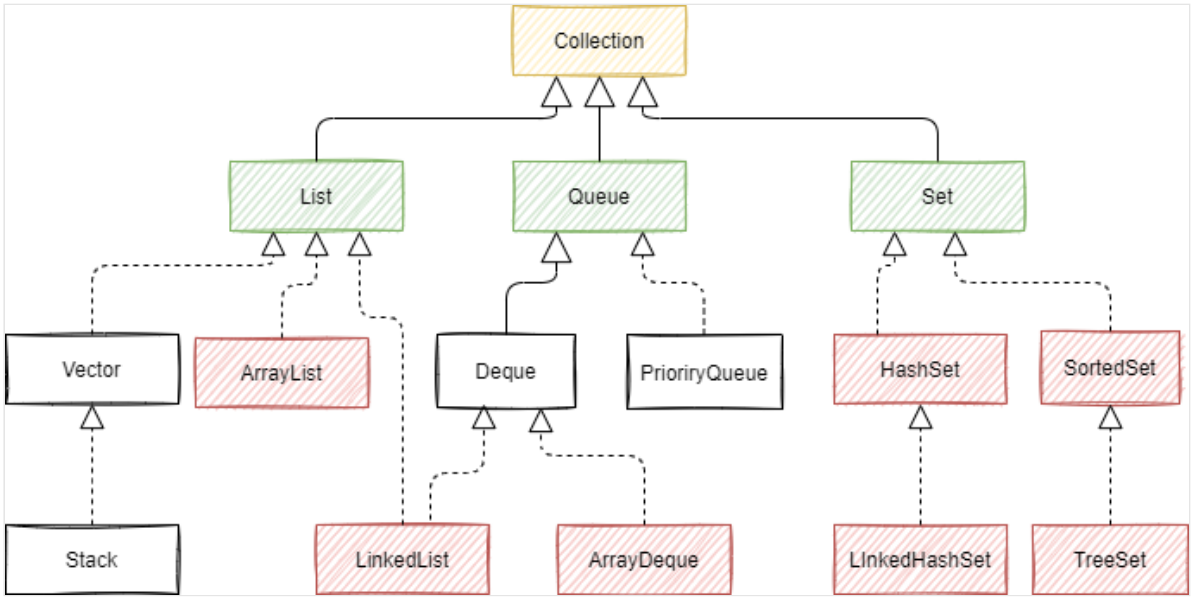
> 本文已经收录到github仓库，此仓库用于分享Java相关知识总结，包括Java基础、MySQL、Spring Boot、MyBatis、Redis、RabbitMQ、计算机网络、数据结构与算法等等，欢迎大家提pr和star！
>
> github地址：https://github.com/Tyson0314/Java-learning
>
> 如果github访问不了，可以访问gitee仓库。
>
> gitee地址：https://gitee.com/tysondai/Java-learning

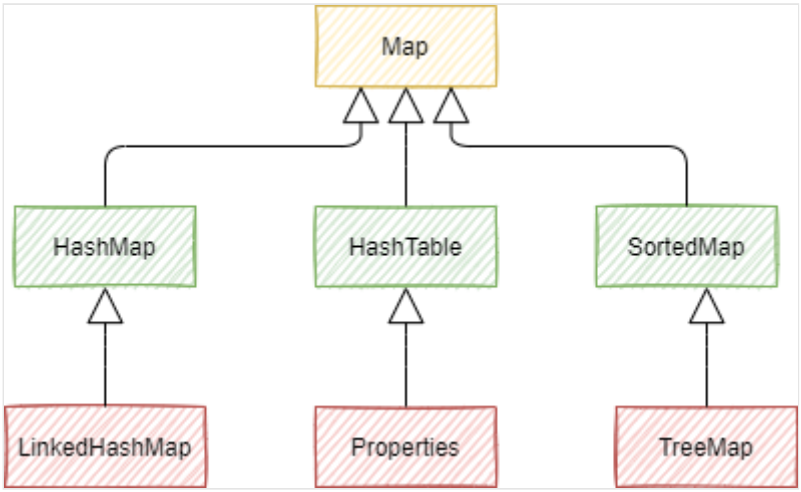
Java集合

1. 常见的集合有哪些?

Java集合类主要由两个接口**Collection**和**Map**派生出来的，Collection有三个子接口：List、Set、Queue。

Java集合框架图如下：





List代表了有序可重复集合，可直接根据元素的索引来访问；Set代表无序不可重复集合，只能根据元素本身来访问；Queue是队列集合。Map代表的是存储key-value对的集合，可根据元素的key来访问value。

集合体系中常用的实现类有ArrayList、LinkedList、HashSet、TreeSet、HashMap、TreeMap等实现类。

2. List 、Set和Map 的区别

- List 以索引来存取元素，有序的，元素是允许重复的，可以插入多个null；
- Set 不能存放重复元素，无序的，只允许一个null；
- Map 保存键值对映射；
- List 底层实现有数组、链表两种方式；Set、Map 容器有基于哈希存储和红黑树两种方式实现；
- Set 基于 Map 实现，Set 里的元素值就是 Map的键值。

3. ArrayList 了解吗?

`ArrayList` 的底层是动态数组，它的容量能动态增长。在添加大量元素前，应用可以使用 `ensureCapacity` 操作增加 `ArrayList` 实例的容量。ArrayList 继承了 AbstractList ，并实现了 List 接口。

4. ArrayList 的扩容机制?

ArrayList扩容的本质就是计算出新的扩容数组的size后实例化，并将原有数组内容复制到新数组中去。**默认情况下，新的容量会是原容量的1.5倍。**以JDK1.8为例说明:

```
public boolean add(E e) {
    //判断是否可以容纳e，若能，则直接添加在末尾；若不能，则进行扩容，然后再把e添加在末尾
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    //将e添加到数组末尾
    elementData[size++] = e;
    return true;
    }

// 每次在add()一个元素时，arraylist都需要对这个list的容量进行一个判断。通过ensureCapacityInternal()方法确保当前ArrayList维护的数组具有存储新元素的能力，经过处理之后将元素存储在数组elementData的尾部

private void ensureCapacityInternal(int minCapacity) {
      ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));
}

private static int calculateCapacity(Object[] elementData, int minCapacity) {
        //如果传入的是个空数组则最小容量取默认容量与minCapacity之间的最大值
        if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
            return Math.max(DEFAULT_CAPACITY, minCapacity);
        }
        return minCapacity;
    }

  private void ensureExplicitCapacity(int minCapacity) {
        modCount++;
        // 若ArrayList已有的存储能力满足最低存储要求，则返回add直接添加元素；如果最低要求的存储能力>ArrayList已有的存储能力，这就表示ArrayList的存储能力不足，因此需要调用 grow();方法进行扩容
        if (minCapacity - elementData.length > 0)
            grow(minCapacity);
    }


private void grow(int minCapacity) {
        // 获取elementData数组的内存空间长度
        int oldCapacity = elementData.length;
        // 扩容至原来的1.5倍
        int newCapacity = oldCapacity + (oldCapacity >> 1);
        //校验容量是否够
        if (newCapacity - minCapacity < 0)
            newCapacity = minCapacity;
        //若预设值大于默认的最大值，检查是否溢出
        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity);
```

```
        // 调用Arrays.copyOf方法将elementData数组指向新的内存空间
         //并将elementData的数据复制到新的内存空间
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
```

5. 怎么在遍历 ArrayList 时移除一个元素?

foreach删除会导致快速失败问题，可以使用迭代器的 remove() 方法。

```
Iterator itr = list.iterator();
while(itr.hasNext()) {
      if(itr.next().equals("jay") {
        itr.remove();
      }
}
```

6. Arraylist 和 Vector 的区别

1. ArrayList在内存不够时默认是扩展50% + 1个，Vector是默认扩展1倍。
2. Vector属于线程安全级别的，但是大多数情况下不使用Vector，因为操作Vector效率比较低。

7. Arraylist 与 LinkedList 区别

1. ArrayList基于动态数组实现；LinkedList基于链表实现。
2. 对于随机index访问的get和set方法，ArrayList的速度要优于LinkedList。因为ArrayList直接通过数组下标直接找到元素；LinkedList要移动指针遍历每个元素直到找到为止。
3. 新增和删除元素，LinkedList的速度要优于ArrayList。因为ArrayList在新增和删除元素时，可能扩容和复制数组；LinkedList实例化对象需要时间外，只需要修改指针即可。

8. HashMap

HashMap 使用数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的， 链表长度大于8（TREEIFY_THRESHOLD）时，会把链表转换为红黑树，红黑树节点个数小于6（UNTREEIFY_THRESHOLD）时才转化为链表，防止频繁的转化。

8.1. 解决hash冲突的办法有哪些？HashMap用的哪种?

解决Hash冲突方法有:开放定址法、再哈希法、链地址法。HashMap中采用的是 链地址法 。

- 开放定址法基本思想就是，如果`p=H(key)`出现冲突时，则以`p`为基础，再次hash，`p1=H(p)`,如果p1再次出现冲突，则以p1为基础，以此类推，直到找到一个不冲突的哈希地址`pi`。 因此开放定址法所需要的hash表的长度要大于等于所需要存放的元素，而且因为存在再次hash，所以只能在删除的节点上做标记，而不能真正删除节点。
- 再哈希法提供多个不同的hash函数，当`R1=H1(key1)`发生冲突时，再计算`R2=H2(key1)`，直到没有冲突为止。 这样做虽然不易产生堆集，但增加了计算的时间。
- 链地址法将哈希值相同的元素构成一个同义词的单链表,并将单链表的头指针存放在哈希表的第i个单元中，查找、插入和删除主要在同义词链表中进行。链表法适用于经常进行插入和删除的情况。

8.2. 使用的hash算法?

Hash算法：取key的hashCode值、高位运算、取模运算。

```
h=key.hashCode() //第一步 取hashCode值
h^(h>>>16)  //第二步 高位参与运算，减少冲突
return h&(length-1);  //第三步 取模运算
```

在JDK1.8的实现中，优化了高位运算的算法，通过hashCode()的高16位异或低16位实现的：这么做可以在数组比较小的时候，也能保证考虑到高低位都参与到Hash的计算中，可以减少冲突，同时不会有太大的开销。

8.3. 扩容过程?

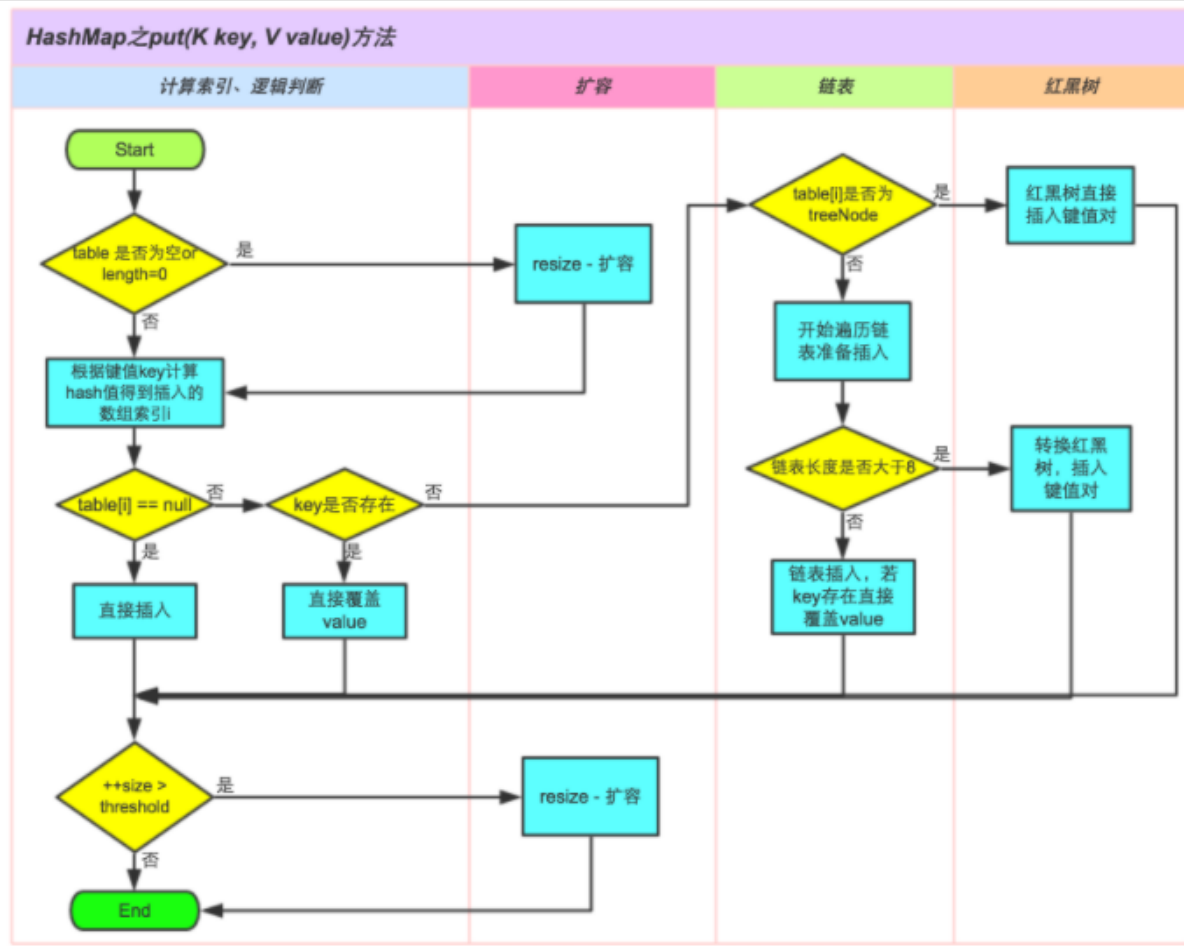
1.8扩容机制：当元素个数大于threshold时，会进行扩容，使用2倍容量的数组代替原有数组。采用尾插入的方式将原数组元素拷贝到新数组。1.8扩容之后链表元素相对位置没有变化，而1.7扩容之后链表元素会倒置。

1.7链表新节点采用的是头插法，这样在线程一扩容迁移元素时，会将元素顺序改变，导致两个线程中出现元素的相互指向而形成循环链表，1.8采用了尾插法，避免了这种情况的发生。

原数组的元素在重新计算hash之后，因为数组容量n变为2倍，那么n-1的mask范围在高位多1bit。在元素拷贝过程不需要重新计算元素在数组中的位置，只需要看看原来的hash值新增的那个bit是1还是0，是0的话索引没变，是1的话索引变成“原索引+oldCap”（根据`e.hash & (oldCap - 1) == 0`判断）。这样可以省去重新计算hash值的时间，而且由于新增的1bit是0还是1可以认为是随机的，因此resize的过程会均匀的把之前的冲突的节点分散到新的bucket。

8.4. put方法流程?

1. 如果table没有初始化就先进行初始化过程
2. 使用hash算法计算key的索引
3. 判断索引处有没有存在元素，没有就直接插入
4. 如果索引处存在元素，则遍历插入，有两种情况，一种是链表形式就直接遍历到尾端插入，一种是红黑树就按照红黑树结构插入
5. 链表的数量大于阈值8，就要转换成红黑树的结构
6. 添加成功后会检查是否需要扩容



8.5. 红黑树的特点?

- 每个节点或者是黑色，或者是红色。
- 根节点是黑色。
- 每个叶子节点（NIL）是黑色。
- 如果一个节点是红色的，则它的子节点必须是黑色的。
- 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

8.6. 为什么使用红黑树而不使用AVL树?

ConcurrentHashMap 在put的时候会加锁，使用红黑树插入速度更快，可以减少等待锁释放的时间。红黑树是对AVL树的优化，只要求部分平衡，用非严格的平衡来换取增删节点时候旋转次数的降低，提高了插入和删除的性能。

8.7. 在解决 hash 冲突的时候，为什么选择先用链表，再转红黑树?

因为红黑树需要进行左旋，右旋，变色这些操作来保持平衡，而单链表不需要。当元素小于 8 个的时候，链表结构可以保证查询性能。当元素大于 8 个的时候， 红黑树搜索时间复杂度是 O(logn)，而链表是 O(n)，此时需要红黑树来加快查询速度，但是插入和删除节点的效率变慢了。如果一开始就用红黑树结构，元素太少，插入和删除节点的效率又比较慢，浪费性能。

8.8. HashMap 的长度为什么是 2 的幂次方?

Hash 值的范围值比较大，使用之前需要先对数组的长度取模运算，得到的余数才是元素存放的位置也就是对应的数组下标。这个数组下标的计算方法是`(n - 1) & hash`。将HashMap的长度定为2 的幂次方，这样就可以使用`(n - 1)&hash`位运算代替%取余的操作，提高性能。

8.9. HashMap默认加载因子是多少？为什么是 0.75?

先看下HashMap的默认构造函数：

```
int threshold;              // 容纳键值对的最大值
final float loadFactor;     // 负载因子
int modCount;
int size;
```

Node[] table的初始化长度length为16，默认的loadFactor是0.75，0.75是对空间和时间效率的一个平衡选择，根据泊松分布，loadFactor 取0.75碰撞最小。一般不会修改，除非在时间和空间比较特殊的情况下：

- 如果内存空间很多而又对时间效率要求很高，可以降低负载因子Load factor的值 。
- 如果内存空间紧张而对时间效率要求不高，可以增加负载因子loadFactor的值，这个值可以大于1。

8.10. 一般用什么作为HashMap的key?

一般用Integer、String 这种不可变类当 HashMap 当 key。String类比较常用。

- 因为 String 是不可变的，所以在它创建的时候 hashcode 就被缓存了，不需要重新计算。这就是 HashMap 中的key经常使用字符串的原因。
- 获取对象的时候要用到 equals() 和 hashCode() 方法，而Integer、String这些类都已经重写了 hashCode() 以及 equals() 方法，不需要自己去重写这两个方法。

8.11. HashMap为什么线程不安全?

- 多线程下扩容死循环。JDK1.7中的 HashMap 使用头插法插入元素，在多线程的环境下，扩容的时候有可能导致**环形链表**的出现，形成死循环。
- 在JDK1.8中，在多线程环境下，会发生**数据覆盖**的情况。

8.12. HashMap和HashTable的区别?

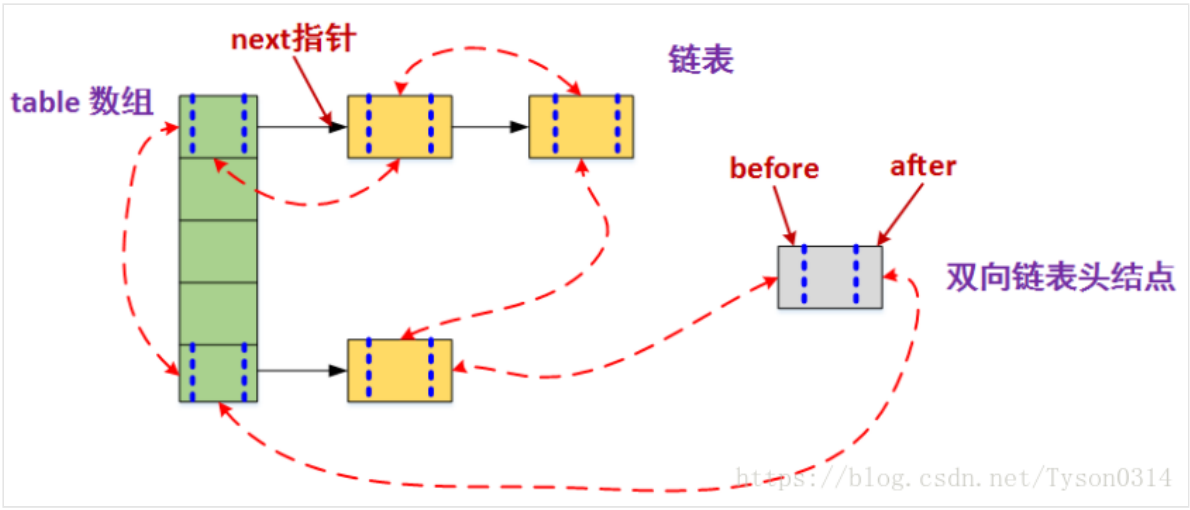
HashMap和Hashtable都实现了Map接口。

1. HashMap可以接受为null的key和value，key为null的键值对放在下标为0的头结点的链表中，而Hashtable则不行。
2. HashMap是非线程安全的，HashTable是线程安全的。Jdk1.5提供了ConcurrentHashMap，它是HashTable的替代。
3. Hashtable很多方法是同步方法，在单线程环境下它比HashMap要慢。
4. 哈希值的使用不同，HashTable直接使用对象的hashCode。而HashMap重新计算hash值。

9. LinkedHashMap底层原理?

HashMap是无序的，迭代HashMap所得到元素的顺序并不是它们最初放到HashMap的顺序，即不能保持它们的插入顺序。

LinkedHashMap继承于HashMap，是HashMap和LinkedList的融合体，具备两者的特性。每次put操作都会将entry插入到双向链表的尾部。

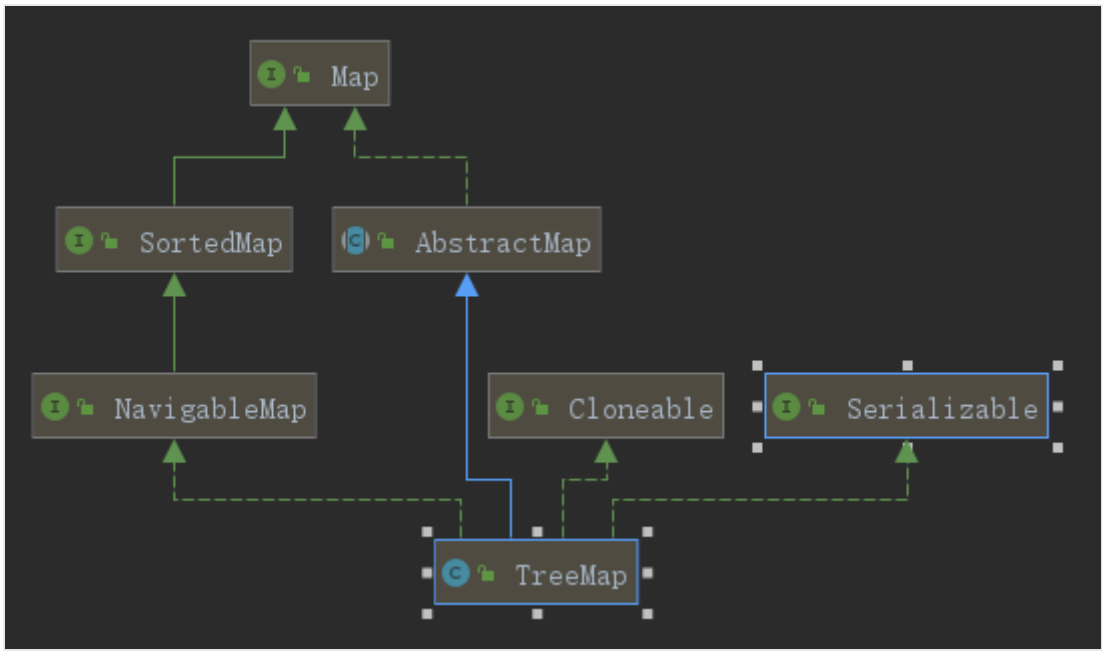


10. 讲一下TreeMap?

TreeMap是一个能比较元素大小的Map集合，会对传入的key进行了大小排序。可以使用元素的自然顺序，也可以使用集合中自定义的比较器来进行排序。

```
public class TreeMap<K,V>
    extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable, java.io.Serializable {
}
```

TreeMap 的继承结构：



TreeMap的特点：

1. TreeMap是有序的key-value集合，通过红黑树实现。根据键的自然顺序进行排序或根据提供的Comparator进行排序。
2. TreeMap继承了AbstractMap，实现了NavigableMap接口，支持一系列的导航方法，给定具体搜索目标，可以返回最接近的匹配项。如floorEntry()、ceilingEntry()分别返回小于等于、大于等于给定键关联的Map.Entry()对象，不存在则返回null。lowerKey()、floorKey、ceilingKey、higherKey()只返回关联的key。

11. HashSet底层原理?

HashSet 基于 HashMap 实现。放入HashSet中的元素实际上由HashMap的key来保存，而HashMap的value则存储了一个静态的Object对象。

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable {
    static final long serialVersionUID = -5024744406713321676L;

    private transient HashMap<E,Object> map; //基于HashMap实现
    //...
}
```

12. HashSet、LinkedHashSet 和 TreeSet 的区别?

`HashSet` 是 `Set` 接口的主要实现类， `HashSet` 的底层是 `HashMap`，线程不安全的，可以存储 null 值；

`LinkedHashSet` 是 `HashSet` 的子类，能够按照添加的顺序遍历；

`TreeSet` 底层使用红黑树，能够按照添加元素的顺序进行遍历，排序的方式可以自定义。

13. 什么是fail fast?

fast-fail是Java集合的一种错误机制。当多个线程对同一个集合进行操作时，就有可能会产生fast-fail事件。例如：当线程a正通过iterator遍历集合时，另一个线程b修改了集合的内容，此时modCount（记录集合操作过程的修改次数）会加1，不等于expectedModCount，那么线程a访问集合的时候，就会抛出ConcurrentModificationException，产生fast-fail事件。边遍历边修改集合也会产生fast-fail事件。

解决方法：

- 使用Colletions.synchronizedList方法或在修改集合内容的地方加上synchronized。这样的话，增删集合内容的同步锁会阻塞遍历操作，影响性能。
- 使用CopyOnWriteArrayList来替换ArrayList。在对CopyOnWriteArrayList进行修改操作的时候，会拷贝一个新的数组，对新的数组进行操作，操作完成后再把引用移到新的数组。

14. 什么是fail safe?

采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。java.util.concurrent包下的容器都是安全失败，可以在多线程下并发使用，并发修改。

原理：由于迭代时是对原集合的拷贝进行遍历，所以在遍历过程中对原集合所作的修改并不能被迭代器检测到，所以不会触发Concurrent Modification Exception。

缺点：基于拷贝内容的优点是避免了Concurrent Modification Exception，但同样地，迭代器并不能访问到修改后的内容，即：迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的。

15. 讲一下ArrayDeque?

ArrayDeque实现了双端队列，内部使用循环数组实现，默认大小为16。它的特点有：

1. 在两端添加、删除元素的效率较高
2. 根据元素内容查找和删除的效率比较低。
3. 没有索引位置的概念，不能根据索引位置进行操作。

ArrayDeque和LinkedList都实现了Deque接口，如果只需要从两端进行操作，ArrayDeque效率更高一些。如果同时需要根据索引位置进行操作，或者经常需要在中间进行插入和删除（LinkedList有相应的api，如add(int index, E e)），则应该选LinkedList。

ArrayDeque和LinkedList都是线程不安全的，可以使用Collections工具类中synchronizedXxx()转换成线程同步。

16. 哪些集合类是线程安全的？哪些不安全？

线性安全的集合类：

- Vector：比ArrayList多了同步机制。
- Hashtable。
- ConcurrentHashMap：是一种高效并且线程安全的集合。
- Stack：栈，也是线程安全的，继承于Vector。

线性不安全的集合类：

- Hashmap
- Arraylist
- LinkedList
- HashSet
- TreeSet
- TreeMap

17. 迭代器 Iterator 是什么？

Iterator模式用同一种逻辑来遍历集合。它可以把访问逻辑从不同类型的集合类中抽象出来，不需要了解集合内部实现便可以遍历集合元素，统一使用 Iterator 提供的接口去遍历。它的特点是更加安全，因为它可以保证，在当前遍历的集合元素被更改的时候，就会抛出 ConcurrentModificationException 异常。

```
public interface Collection<E> extends Iterable<E> {
    Iterator<E> iterator();
}
```

主要有三个方法：hasNext()、next()和remove()。

18. Iterator 和 ListIterator 有什么区别？

ListIterator 是 Iterator的增强版。

- ListIterator遍历可以是逆向的，因为有previous()和hasPrevious()方法，而Iterator不可以。
- ListIterator有add()方法，可以向List添加对象，而Iterator却不能。
- ListIterator可以定位当前的索引位置，因为有nextIndex()和previousIndex()方法，而Iterator不可以。
- ListIterator可以实现对象的修改，set()方法可以实现。lierator仅能遍历，不能修改。
- ListIterator只能用于遍历List及其子类，Iterator可用来遍历所有集合。

19. 并发容器

JDK 提供的这些容器大部分在 `java.util.concurrent` 包中。

- **ConcurrentHashMap:** 线程安全的 HashMap
- **CopyOnWriteArrayList:** 线程安全的 List，在读多写少的场合性能非常好，远远好于 Vector.

- **ConcurrentLinkedQueue:** 高效的并发队列，使用链表实现。可以看做一个线程安全的 LinkedList，这是一个非阻塞队列。
- **BlockingQueue:** 阻塞队列接口，JDK 内部通过链表、数组等方式实现了这个接口。非常适合用于作为数据共享的通道。
- **ConcurrentSkipListMap:** 跳表的实现。使用跳表的数据结构进行快速查找。

19.1. ConcurrentHashMap

多线程环境下，使用Hashmap进行put操作会引起死循环，应该使用支持多线程的ConcurrentHashMap。

JDK1.8 ConcurrentHashMap取消了segment分段锁，而采用CAS和synchronized来保证并发安全。数据结构采用数组+链表/红黑二叉树。synchronized只锁定当前链表或红黑二叉树的首节点，相比1.7锁定HashEntry数组，锁粒度更小，支持更高的并发量。当链表长度过长时，Node会转换成TreeNode，提高查找速度。

19.1.1. put执行流程?

在put的时候需要锁住Segment，保证并发安全。调用get的时候不加锁，因为node数组成员val和指针next是用volatile修饰的，更改后的值会立刻刷新到主存中，保证了可见性，node数组table也用volatile修饰，保证在运行过程对其他线程具有可见性。

```
transient volatile Node<K,V>[] table;

static class Node<K,V> implements Map.Entry<K,V> {
    volatile V val;
    volatile Node<K,V> next;
}
```

put 操作流程：

1. 如果table没有初始化就先进行初始化过程
2. 使用hash算法计算key的位置
3. 如果这个位置为空则直接CAS插入，如果不为空的话，则取出这个节点来
4. 如果取出来的节点的hash值是MOVED(-1)的话，则表示当前正在对这个数组进行扩容，复制到新的数组，则当前线程也去帮助复制
5. 如果这个节点，不为空，也不在扩容，则通过synchronized来加锁，进行添加操作，这里有两种情况，一种是链表形式就直接遍历到尾端插入或者覆盖掉相同的key，一种是红黑树就按照红黑树结构插入
6. 链表的数量大于阈值8，就会转换成红黑树的结构或者进行扩容（table长度小于64）
7. 添加成功后会检查是否需要扩容

19.1.2. 怎么扩容?

数组扩容transfer方法中会设置一个步长，表示一个线程处理的数组长度，最小值是16。在一个步长范围内只有一个线程会对其进行复制移动操作。

19.1.3. ConcurrentHashMap 和 Hashtable 的区别?

1. Hashtable通过使用synchronized修饰方法的方式来实现多线程同步，因此，Hashtable的同步会锁住整个数组。在高并发的情况下，性能会非常差。ConcurrentHashMap采用了更细粒度的锁来提高在并发情况下的效率。注：Synchronized容器（同步容器）也是通过synchronized关键字来实现线程安全，在使用的时候会对所有的数据加锁。
2. Hashtable默认的大小为11，当达到阈值后，每次按照下面的公式对容量进行扩充：newCapacity = oldCapacity * 2 + 1。ConcurrentHashMap默认大小是16，扩容时容量扩大为原来的2倍。

19.2. CopyOnWrite

写时复制。当我们往容器添加元素时，不直接往容器添加，而是先将当前容器进行复制，复制出一个新的容器，然后往新的容器添加元素，添加完元素之后，再将原容器的引用指向新容器。这样做的好处就是可以对CopyOnWrite容器进行并发的读而不需要加锁，因为当前容器不会被修改。

```
    public boolean add(E e) {
        final ReentrantLock lock = this.lock;
        lock.lock(); //add方法需要加锁
        try {
            Object[] elements = getArray();
            int len = elements.length;
            Object[] newElements = Arrays.copyOf(elements, len + 1); //复制新数组
            newElements[len] = e;
            setArray(newElements); //原容器的引用指向新容器
            return true;
        } finally {
            lock.unlock();
        }
    }
```

从JDK1.5开始Java并发包里提供了两个使用CopyOnWrite机制实现的并发容器，它们是CopyOnWriteArrayList和CopyOnWriteArraySet。

CopyOnWriteArrayList中add方法添加的时候是需要加锁的，保证同步，避免了多线程写的时候复制出多个副本。读的时候不需要加锁，如果读的时候有其他线程正在向CopyOnWriteArrayList添加数据，还是可以读到旧的数据。

缺点:

- 内存占用问题。由于CopyOnWrite的写时复制机制，在进行写操作的时候，内存里会同时驻扎两个对象的内存。
- CopyOnWrite容器不能保证数据的实时一致性，可能读取到旧数据。

19.3. ConcurrentLinkedQueue

非阻塞队列。高效的并发队列，使用链表实现。可以看做一个线程安全的 LinkedList，通过 CAS 操作实现。

如果对队列加锁的成本较高则适合使用无锁的 ConcurrentLinkedQueue 来替代。适合在对性能要求相对较高，同时有多个线程对队列进行读写的场景。

非阻塞队列中的几种主要方法： add(E e) : 将元素e插入到队列末尾，如果插入成功，则返回true；如果插入失败（即队列已满），则会抛出异常； remove() ： 移除队首元素，若移除成功，则返回true；如果移除失败（队列为空），则会抛出异常； offer(E e) ： 将元素e插入到队列末尾，如果插入成功，则返回true；如果插入失败（即队列已满），则返回false； poll() ： 移除并获取队首元素，若成功，则返回队首元素；否则返回null； peek() ： 获取队首元素，若成功，则返回队首元素；否则返回null

对于非阻塞队列，一般情况下建议使用offer、poll和peek三个方法，不建议使用add和remove方法。因为使用offer、poll和peek三个方法可以通过返回值判断操作成功与否，而使用add和remove方法却不能达到这样的效果。

19.4. 阻塞队列

阻塞队列是java.util.concurrent包下重要的数据结构，BlockingQueue提供了线程安全的队列访问方式：当阻塞队列进行插入数据时，如果队列已满，线程将会阻塞等待直到队列非满；从阻塞队列取数据时，如果队列已空，线程将会阻塞等待直到队列非空。并发包下很多高级同步类的实现都是基于BlockingQueue实现的。BlockingQueue 适合用于作为数据共享的通道。

使用阻塞算法的队列可以用一个锁（入队和出队用同一把锁）或两个锁（入队和出队用不同的锁）等方式来实现。

阻塞队列和一般的队列的区别就在于：

1. 多线程支持，多个线程可以安全的访问队列
2. 阻塞操作，当队列为空的时候，消费线程会阻塞等待队列不为空；当队列满了的时候，生产线程就会阻塞直到队列不满。

方法

方法\处理方式	抛出异常	返回特殊值	一直阻塞	超时退出
插入方法	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除方法	remove()	poll()	take()	poll(time,unit)
检查方法	element()	peek()	不可用	不可用

19.4.1. JDK提供的阻塞队列

JDK 7 提供了7个阻塞队列，如下

1、**ArrayBlockingQueue**

有界阻塞队列，底层采用数组实现。ArrayBlockingQueue 一旦创建，容量不能改变。其并发控制采用可重入锁来控制，不管是插入操作还是读取操作，都需要获取到锁才能进行操作。此队列按照先进先出（FIFO）的原则对元素进行排序。默认情况下不能保证线程访问队列的公平性，参数`fair`可用于设置线程是否公平访问队列。为了保证公平性，通常会降低吞吐量。

```
private static ArrayBlockingQueue<Integer> blockingQueue = new
ArrayBlockingQueue<Integer>(10,true);//fair
```

2、**LinkedBlockingQueue**

LinkedBlockingQueue是一个用单向链表实现的有界阻塞队列，可以当做无界队列也可以当做有界队列来使用。通常在创建 LinkedBlockingQueue 对象时，会指定队列最大的容量。此队列的默认和最大长度为`Integer.MAX_VALUE`。此队列按照先进先出的原则对元素进行排序。与 ArrayBlockingQueue 相比起来具有更高的吞吐量。

3、**PriorityBlockingQueue**

支持优先级的**无界**阻塞队列。默认情况下元素采取自然顺序升序排列。也可以自定义类实现`compareTo()`方法来指定元素排序规则，或者初始化PriorityBlockingQueue时，指定构造参数`Comparator`来进行排序。

PriorityBlockingQueue 只能指定初始的队列大小，后面插入元素的时候，如果空间不够的话会**自动扩容。**

PriorityQueue 的线程安全版本。不可以插入 null 值，同时，插入队列的对象必须是可比较大小的（comparable），否则报 ClassCastException 异常。它的插入操作 put 方法不会 block，因为它是无界队列（take 方法在队列为空的时候会阻塞）。

4、**DelayQueue**

支持延时获取元素的无界阻塞队列。队列使用PriorityBlockingQueue来实现。队列中的元素必须实现Delayed接口，在创建元素时可以指定多久才能从队列中获取当前元素。只有在延迟期满时才能从队列中提取元素。

5、**SynchronousQueue**

不存储元素的阻塞队列，每一个put必须等待一个take操作，否则不能继续添加元素。支持公平访问队列。

SynchronousQueue可以看成是一个传球手，负责把生产者线程处理的数据直接传递给消费者线程。队列本身不存储任何元素，非常适合传递性场景。SynchronousQueue的吞吐量高于LinkedBlockingQueue和ArrayBlockingQueue。

6、**LinkedTransferQueue**

由链表结构组成的无界阻塞TransferQueue队列。相对于其他阻塞队列，多了tryTransfer和transfer方法。

transfer方法：如果当前有消费者正在等待接收元素（take或者待时间限制的poll方法），transfer可以把生产者传入的元素立刻传给消费者。如果没有消费者等待接收元素，则将元素放在队列的tail节点，并等到该元素被消费者消费了才返回。

tryTransfer方法：用来试探生产者传入的元素能否直接传给消费者。如果没有消费者在等待，则返回false。和上述方法的区别是该方法无论消费者是否接收，方法立即返回。而transfer方法是必须等到消费者消费了才返回。

19.4.2. 原理

JDK使用通知模式实现阻塞队列。所谓通知模式，就是当生产者往满的队列里添加元素时会阻塞生产者，当消费者消费了一个队列中的元素后，会通知生产者当前队列可用。

ArrayBlockingQueue使用Condition来实现：

```
private final Condition notEmpty;
private final Condition notFull;

public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.items = new Object[capacity];
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull =  lock.newCondition();
}

public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == 0) // 队列为空时，阻塞当前消费者
            notEmpty.await();
        return dequeue();
    } finally {
```

```
        lock.unlock();
    }
}

public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == items.length)
            notFull.await();
        enqueue(e);
    } finally {
        lock.unlock();
    }
}

private void enqueue(E x) {
    final Object[] items = this.items;
    items[putIndex] = x;
    if (++putIndex == items.length)
          putIndex = 0;
     count++;
     notEmpty.signal(); // 队列不为空时，通知消费者获取元素
}
```

20. 参考链接

http://www.importnew.com/20386.html

https://www.cnblogs.com/yangming1996/p/7997468.html

https://coolshell.cn/articles/9606.htm（HashMap 死循环）

Java并发

1. 线程池

线程池：一个管理线程的池子。

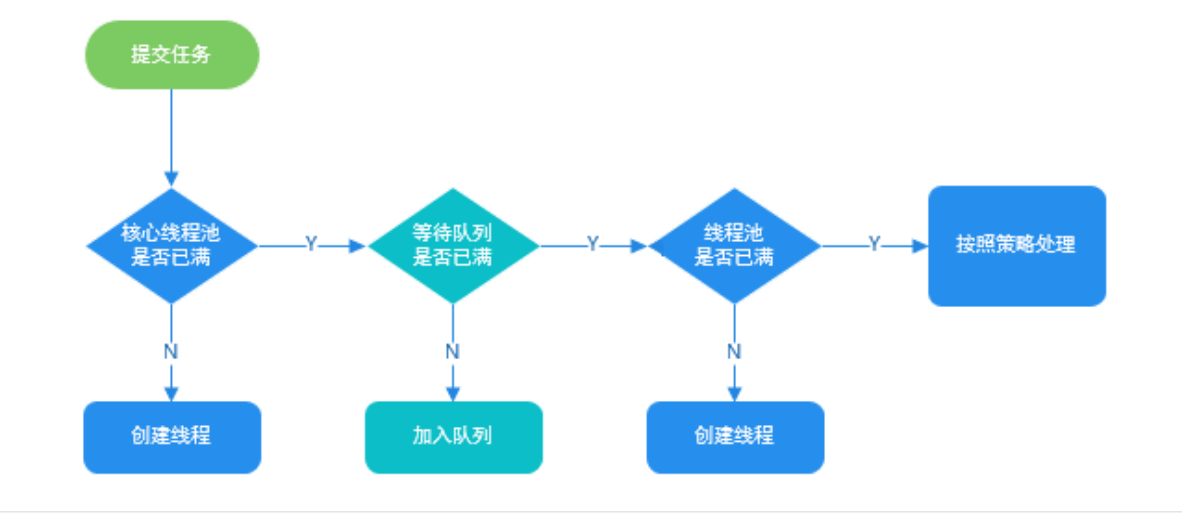
1.1. 为什么使用线程池?

- **降低资源消耗。**通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- **提高响应速度。**当任务到达时，任务可以不需要的等到线程创建就能立即执行。
- **提高线程的可管理性。**统一管理线程，避免系统创建大量同类线程而导致消耗完内存。

```
public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long
keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory
threadFactory, RejectedExecutionHandler handler);
```

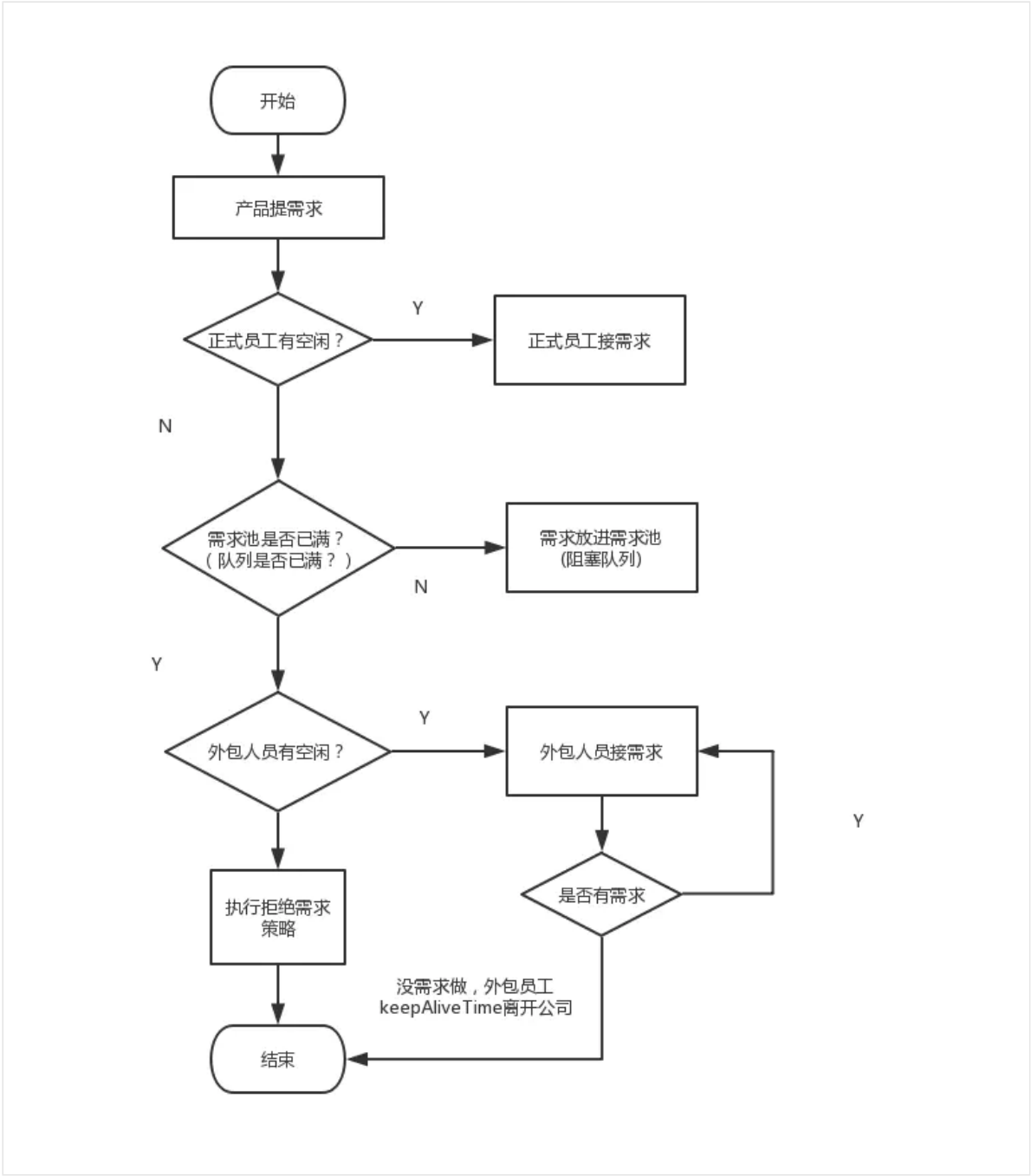
1.2. 线程池执行原理?

创建新的线程需要获取全局锁，通过这种设计可以尽量避免获取全局锁，当 ThreadPoolExecutor 完成预热之后（当前运行的线程数大于等于 corePoolSize），提交的大部分任务都会被放到 BlockingQueue。



为了形象描述线程池执行，打个比喻：

- 核心线程比作公司正式员工
- 非核心线程比作外包员工
- 阻塞队列比作需求池
- 提交任务比作提需求



1.3. 线程池参数有哪些?

ThreadPoolExecutor 的通用构造函数：

```
public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long
keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory
threadFactory, RejectedExecutionHandler handler);
```

- corePoolSize：当有新任务时，如果线程池中线程数没有达到线程池的基本大小，则会创建新的线程执行任务，否则将任务放入阻塞队列。当线程池中存活的线程数总是大于 corePoolSize 时，应该考虑调大 corePoolSize。
- maximumPoolSize：当阻塞队列填满时，如果线程池中线程数没有超过最大线程数，则会创建新的线程运行任务。否则根据拒绝策略处理新任务。非核心线程类似于临时借来的资源，这些线程在空闲时间超过 keepAliveTime 之后，就应该退出，避免资源浪费。
- BlockingQueue：存储等待运行的任务。

- keepAliveTime：**非核心线程**空闲后，保持存活的时间，此参数只对非核心线程有效。设置为0，表示多余的空闲线程会被立即终止。
- TimeUnit：时间单位

```
TimeUnit.DAYS
TimeUnit.HOURS
TimeUnit.MINUTES
TimeUnit.SECONDS
TimeUnit.MILLISECONDS
TimeUnit.MICROSECONDS
TimeUnit.NANOSECONDS
```

- ThreadFactory：每当线程池创建一个新的线程时，都是通过线程工厂方法来完成的。在 ThreadFactory 中只定义了一个方法 newThread，每当线程池需要创建新线程就会调用它。

```
public class MyThreadFactory implements ThreadFactory {
    private final String poolName;

    public MyThreadFactory(String poolName) {
        this.poolName = poolName;
    }

    public Thread newThread(Runnable runnable) {
        return new MyAppThread(runnable, poolName);//将线程池名字传递给构造函数，
用于区分不同线程池的线程
    }
}
```

- RejectedExecutionHandler：当队列和线程池都满了时，根据拒绝策略处理新任务。

```
AbortPolicy：默认的策略，直接抛出RejectedExecutionException
DiscardPolicy：不处理，直接丢弃
DiscardOldestPolicy：将等待队列队首的任务丢弃，并执行当前任务
CallerRunsPolicy：由调用线程处理该任务
```

1.4. 线程池大小怎么设置?

如果线程池线程数量太小，当有大量请求需要处理，系统响应比较慢影响体验，甚至会出现任务队列大量堆积任务导致OOM。

如果线程池线程数量过大，大量线程可能会同时在争取 CPU 资源，这样会导致大量的上下文切换（cpu给线程分配时间片，当线程的cpu时间片用完后保存状态，以便下次继续运行），从 而增加线程的执行时间，影响了整体执行效率。

CPU 密集型任务(N+1)： 这种任务消耗的主要是 CPU 资源，可以将线程数设置为 N（CPU 核心数）+1，比 CPU 核心数多出来的一个线程是为了防止某些原因导致的任务暂停（线程阻塞，如io操作，等待锁，线程sleep）而带来的影响。一旦某个线程被阻塞，释放了cpu资源，而在这种情况下多出来的一个线程就可以充分利用 CPU 的空闲时间。

I/O 密集型任务(2N)： 系统会用大部分的时间来处理 I/O 操作，而线程等待 I/O 操作会被阻塞，释放cpu资源，这时就可以将 CPU 交出给其它线程使用。因此在 I/O 密集型任务的应用中，我们可以多配置一些线程，具体的计算方法：最佳线程数 = CPU核心数 * (1/CPU利用率) = CPU核心数 * (1 + (I/O耗时/CPU耗时))，一般可设置为2N

1.5. 线程池的类型有哪些？适用场景？

常见的线程池有 FixedThreadPool、SingleThreadExecutor、CachedThreadPool 和 ScheduledThreadPool。这几个都是 ExecutorService （线程池）实例。

FixedThreadPool

固定线程数的线程池。任何时间点，最多只有 nThreads 个线程处于活动状态执行任务。

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS,
new LinkedBlockingQueue<Runnable>());
}
```

使用无界队列 LinkedBlockingQueue（队列容量为 Integer.MAX_VALUE），运行中的线程池不会拒绝任务，即不会调用RejectedExecutionHandler.rejectedExecution()方法。

maxThreadPoolSize 是无效参数，故将它的值设置为与 coreThreadPoolSize 一致。

keepAliveTime 也是无效参数，设置为0L，因为此线程池里所有线程都是核心线程，核心线程不会被回收（除非设置了executor.allowCoreThreadTimeOut(true)）。

适用场景：适用于处理CPU密集型的任务，确保CPU在长期被工作线程使用的情况下，尽可能的少的分配线程，即适用执行长期的任务。需要注意的是，FixedThreadPool 不会拒绝任务，**在任务比较多的时候会导致 OOM。**

SingleThreadExecutor

只有一个线程的线程池。

```
public static ExecutionService newSingleThreadExecutor() {
    return new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS, new
LinkedBlockingQueue<Runnable>());
}
```

使用无界队列 LinkedBlockingQueue。线程池只有一个运行的线程，新来的任务放入工作队列，线程处理完任务就循环从队列里获取任务执行。保证顺序的执行各个任务。

适用场景：适用于串行执行任务的场景，一个任务一个任务地执行。**在任务比较多的时候也是会导致 OOM。**

CachedThreadPool

根据需要创建新线程的线程池。

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS,
new SynchronousQueue<Runnable>());
}
```

如果主线程提交任务的速度高于线程处理任务的速度时，`CachedThreadPool` 会不断创建新的线程。极端情况下，这样会导致耗尽 cpu 和内存资源。

使用没有容量的SynchronousQueue作为线程池工作队列，当线程池有空闲线程时，`SynchronousQueue.offer(Runnable task)` 提交的任务会被空闲线程处理，否则会创建新的线程处理任务。

适用场景：用于并发执行大量短期的小任务。`CachedThreadPool`允许创建的线程数量为Integer.MAX_VALUE，**可能会创建大量线程，从而导致 OOM。**

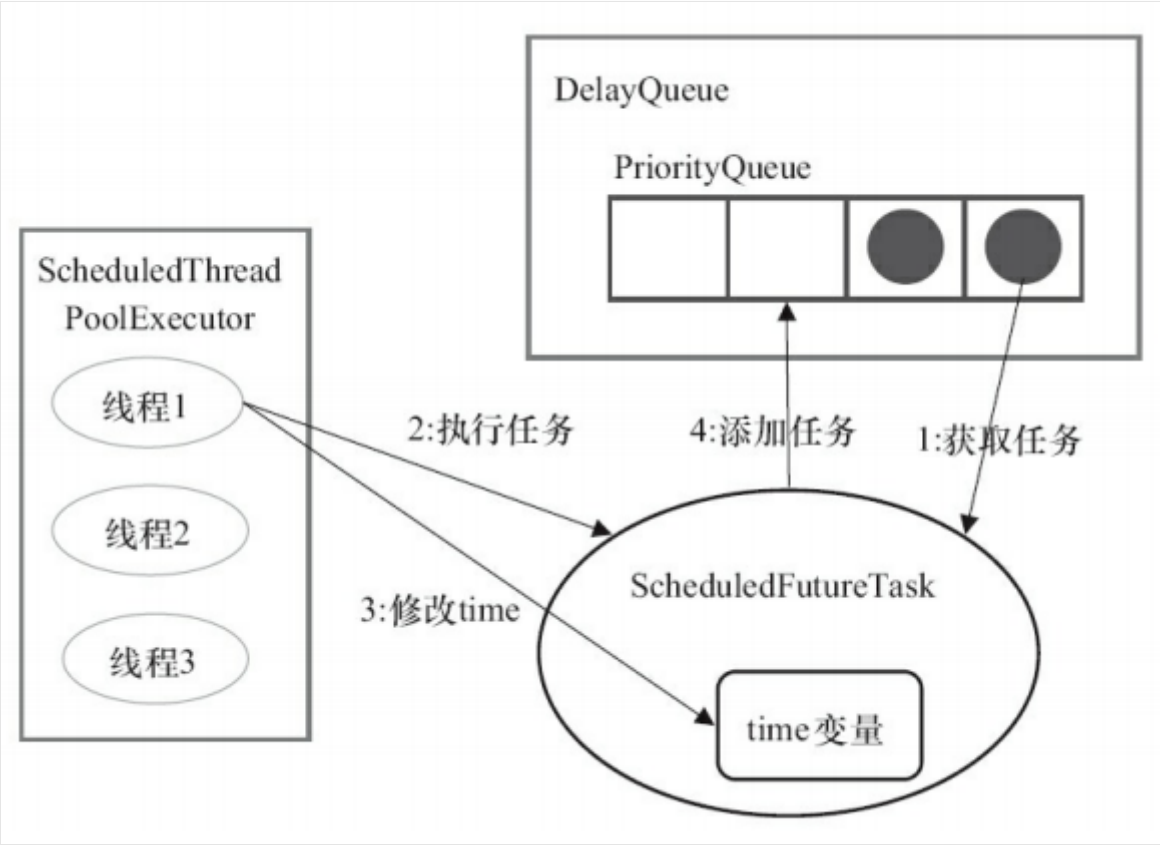
ScheduledThreadPoolExecutor

在给定的延迟后运行任务，或者定期执行任务。在实际项目中基本不会被用到，因为有其他方案选择比如`quartz`。

使用的任务队列`DelayQueue`封装了一个`PriorityQueue`，`PriorityQueue`会对队列中的任务进行排序，时间早的任务先被执行(即`ScheduledFutureTask`的`time`变量小的先执行)，如果time相同则先提交的任务会被先执行(`ScheduledFutureTask`的`squenceNumber`变量小的先执行)。

执行周期任务步骤：

1. 线程从`DelayQueue`中获取已到期的`ScheduledFutureTask（DelayQueue.take()）`。到期任务是指`ScheduledFutureTask`的 time 大于等于当前系统的时间；
2. 执行这个`ScheduledFutureTask`；
3. 修改`ScheduledFutureTask`的 time 变量为下次将要被执行的时间；
4. 把这个修改 time 之后的`ScheduledFutureTask`放回`DelayQueue`中（`DelayQueue.add()`）。



适用场景：周期性执行任务的场景，需要限制线程数量的场景。

2. 进程线程

进程是指一个内存中运行的应用程序，每个进程都有自己独立的一块内存空间，一个进程中可以启动多个线程。 线程是比进程更小的执行单位，它是在一个进程中独立的控制流，一个进程可以启动多个线程，每条线程并行执行不同的任务。

2.1. 线程的生命周期

初始(NEW)：线程被构建，还没有调用 start()。

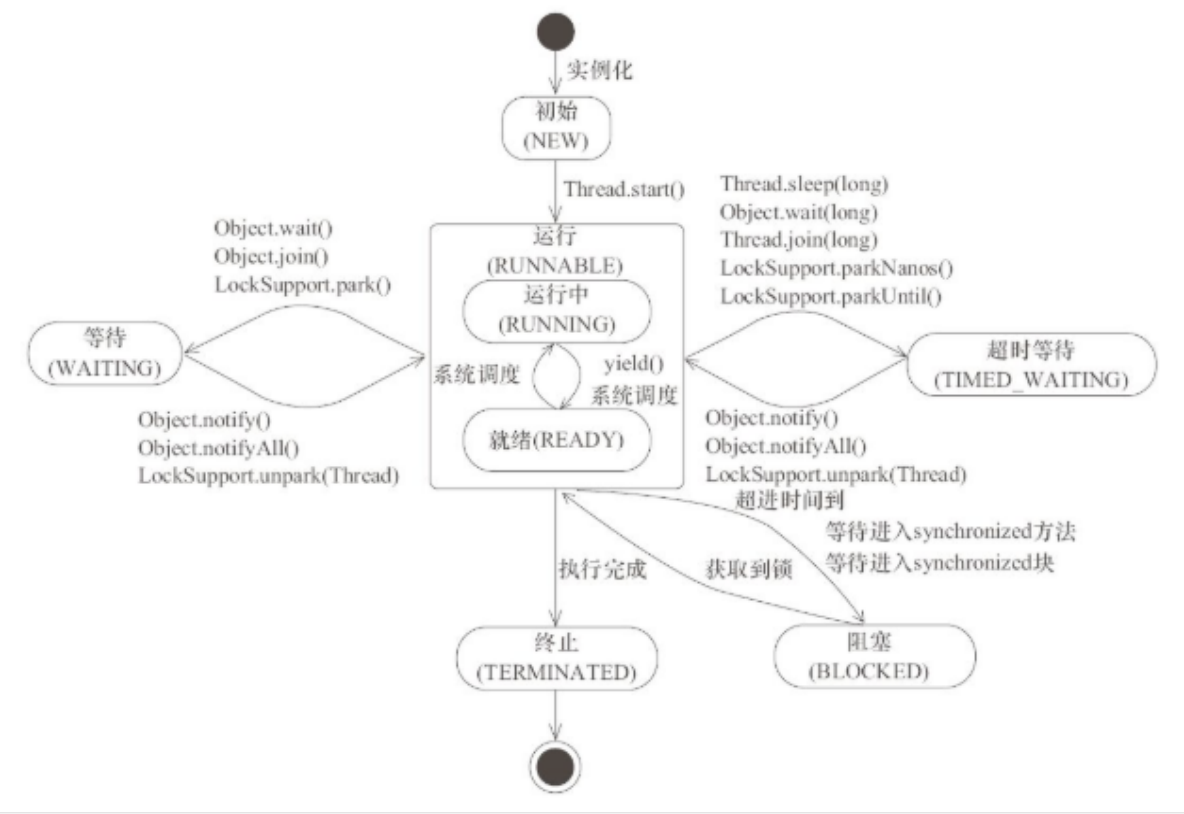
运行(RUNNABLE)：包括操作系统的就绪和运行两种状态。

阻塞(BLOCKED)：一般是被动的，在抢占资源中得不到资源，被动的挂起在内存，等待资源释放将其唤醒。线程被阻塞会释放CPU，不释放内存。

等待(WAITING)：进入该状态的线程需要等待其他线程做出一些特定动作（通知或中断）。

超时等待(TIMED_WAITING)：该状态不同于WAITING，它可以在指定的时间后自行返回。

终止(TERMINATED)：表示该线程已经执行完毕。



> 图片来源：Java并发编程的艺术

2.2. 讲一下线程中断?

线程中断即线程运行过程中被其他线程给打断了，它与 stop 最大的区别是：stop 是由系统强制终止线程，而线程中断则是给目标线程发送一个中断信号，如果目标线程没有接收线程中断的信号并结束线程，线程则不会终止，具体是否退出或者执行其他逻辑取决于目标线程。

线程中断三个重要的方法：

1、java.lang.Thread#interrupt

调用目标线程的interrupt()方法，给目标线程发一个中断信号，线程被打上中断标记。

2、java.lang.Thread#isInterrupted()

判断目标线程是否被中断，不会清除中断标记。

3、java.lang.Thread#interrupted

判断目标线程是否被中断，会清除中断标记。

```
private static void test2() {
    Thread thread = new Thread(() -> {
        while (true) {
            Thread.yield();

            // 响应中断
            if (Thread.currentThread().isInterrupted()) {
                System.out.println("Java技术栈线程被中断，程序退出。");
                return;
            }
        }
    });
    thread.start();
    thread.interrupt();
}
```

2.3. 创建线程有哪几种方式?

- 通过扩展Thread类来创建多线程
- 通过实现Runnable接口来创建多线程，可实现线程间的资源共享
- 实现Callable接口，通过FutureTask接口创建线程。
- 使用Executor框架来创建线程池。

继承 Thread 创建线程代码如下。run()方法是由jvm创建完操作系统级线程后回调的方法，不可以手动调用，手动调用相当于调用普通方法。

```
/**
 * @author: 程序员大彬
 * @time: 2021-09-11 10:15
 */
public class MyThread extends Thread {
    public MyThread() {
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread() + ":" + i);
        }
    }

    public static void main(String[] args) {
        MyThread mThread1 = new MyThread();
        MyThread mThread2 = new MyThread();
        MyThread myThread3 = new MyThread();
        mThread1.start();
        mThread2.start();
        myThread3.start();
    }
}
```

Runnable 创建线程代码:

```
/**
 * @author: 程序员大彬
 * @time: 2021-09-11 10:04
```

```
 */
public class RunnableTest {
    public static  void main(String[] args){
        Runnable1 r = new Runnable1();
        Thread thread = new Thread(r);
        thread.start();
        System.out.println("主线程：["+Thread.currentThread().getName()+"]");
    }
}

class Runnable1 implements Runnable{
    @Override
    public void run() {
        System.out.println("当前线程："+Thread.currentThread().getName());
    }
}
```

实现Runnable接口比继承Thread类所具有的优势：

1. 资源共享，适合多个相同的程序代码的线程去处理同一个资源
2. 可以避免java中的单继承的限制
3. 线程池只能放入实现Runable或Callable类线程，不能直接放入继承Thread的类

Callable 创建线程代码：

```
/**
 * @author: 程序员大彬
 * @time: 2021-09-11 10:21
 */
public class CallableTest {
    public static void main(String[] args) {
        Callable1 c = new Callable1();

        //异步计算的结果
        FutureTask<Integer> result = new FutureTask<>(c);

        new Thread(result).start();

        try {
            //等待任务完成，返回结果
            int sum = result.get();
            System.out.println(sum);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }

}

class Callable1 implements Callable<Integer> {

    @Override
    public Integer call() throws Exception {
        int sum = 0;

        for (int i = 0; i <= 100; i++) {
            sum += i;
```

```
        }
        return sum;
    }
}
```

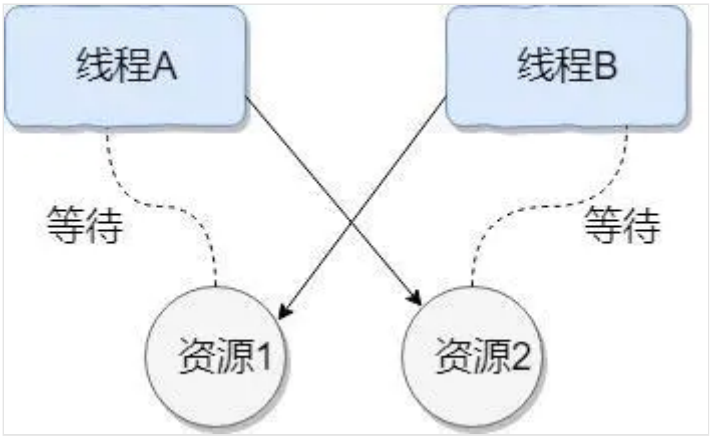
使用 Executor 创建线程代码：

```
/**
 * @author: 程序员大彬
 * @time: 2021-09-11 10:44
 */
public class ExecutorsTest {
    public static void main(String[] args) {
        //获取ExecutorService实例，生产禁用，需要手动创建线程池
        ExecutorService executorService = Executors.newCachedThreadPool();
        //提交任务
        executorService.submit(new RunnableDemo());
    }
}

class RunnableDemo implements Runnable {
    @Override
    public void run() {
        System.out.println("大彬");
    }
}
```

2.4. 什么是线程死锁?

多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

如下图所示，线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。



下面通过例子说明线程死锁，代码来自并发编程之美。

```
public class DeadLockDemo {
    private static Object resource1 = new Object();//资源 1
    private static Object resource2 = new Object();//资源 2

    public static void main(String[] args) {
        new Thread(() -> {
            synchronized (resource1) {
```

```
                System.out.println(Thread.currentThread() + "get resource1");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread() + "waiting get
resource2");
                synchronized (resource2) {
                    System.out.println(Thread.currentThread() + "get
resource2");
                }
            }
        }, "线程 1").start();

        new Thread(() -> {
            synchronized (resource2) {
                System.out.println(Thread.currentThread() + "get resource2");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread() + "waiting get
resource1");
                synchronized (resource1) {
                    System.out.println(Thread.currentThread() + "get
resource1");
                }
            }
        }, "线程 2").start();
    }
}
```

代码输出如下：

```
Thread[线程 1,5,main]get resource1
Thread[线程 2,5,main]get resource2
Thread[线程 1,5,main]waiting get resource2
Thread[线程 2,5,main]waiting get resource1
```

线程 A 通过 synchronized (resource1) 获得 resource1 的监视器锁，然后通过 Thread.sleep(1000);让线程 A 休眠 1s 为的是让线程 B 得到执行然后获取到 resource2 的监视器锁。线程 A 和线程 B 休眠结束了都开始企图请求获取对方的资源，然后这两个线程就会陷入互相等待的状态，这也就产生了死锁。

2.5. 线程死锁怎么产生？怎么避免？

死锁产生的四个必要条件：

- 互斥：一个资源每次只能被一个进程使用（资源独立）
- 请求与保持：一个进程因请求资源而阻塞时，对已获得的资源保持不放（不释放锁）
- 不剥夺：进程已获得的资源，在未使用之前，不能强行剥夺（抢夺资源）
- 循环等待：若干进程之间形成一种头尾相接的循环等待的资源关闭（死循环）

避免死锁的方法：

- 第一个条件 "互斥" 是不能破坏的，因为加锁就是为了保证互斥

- 一次性申请所有的资源，破坏 "占有且等待" 条件
- 占有部分资源的线程进一步申请其他资源时，如果申请不到，主动释放它占有的资源，破坏 "不可抢占" 条件
- 按序申请资源，破坏 "循环等待" 条件

2.6. 线程run和start的区别?

调用 start() 方法是用来启动线程的，轮到该线程执行时，会自动调用 run()；直接调用 run() 方法，无法达到启动多线程的目的，相当于主线程线性执行 Thread 对象的 run() 方法。 一个线程对线的 start() 方法只能调用一次，多次调用会抛出 java.lang.IllegalThreadStateException 异常；run() 方法没有限制。

2.7. 线程都有哪些方法?

join

Thread.join()，在main中创建了thread线程，在main中调用了thread.join()/thread.join(long millis)，main线程放弃cpu控制权，线程进入WAITING/TIMED_WAITING状态，等到thread线程执行完才继续执行main线程。

```
public final void join() throws InterruptedException {
    join(0);
}
```

yield

Thread.yield()，一定是当前线程调用此方法，当前线程放弃获取的CPU时间片，但不释放锁资源，由运行状态变为就绪状态，让OS再次选择线程。作用：让相同优先级的线程轮流执行，但并不保证一定会轮流执行。实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。Thread.yield()不会导致阻塞。该方法与sleep()类似，只是不能由用户指定暂停多长时间。

```
public static native void yield(); //static方法
```

sleep

Thread.sleep(long millis)，一定是当前线程调用此方法，当前线程进入TIMED_WAITING状态，让出cpu资源，但不释放对象锁，指定时间到后又恢复运行。作用：给其它线程执行机会的最佳方式。

```
public static native void sleep(long millis) throws
InterruptedException;//static方法
```

3. volatile底层原理

volatile是轻量级的同步机制，volatile保证变量对所有线程的可见性，不保证原子性。

1. 当对volatile变量进行写操作的时候，JVM会向处理器发送一条LOCK前缀的指令，将该变量所在缓存行的数据写回系统内存。
2. 由于缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己的缓存是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行置为无效状态，当处理器对这个数据进行修改操作的时候，会重新从系统内存中把数据读到处理器缓存中。

MESI（缓存一致性协议）：当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此当其他CPU需要读取这个变量时，就会从内存重新读取。

volatile关键字的两个作用：

1. 保证了不同线程对共享变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。
2. 禁止进行指令重排序。

指令重排序是JVM为了优化指令，提高程序运行效率，在不影响单线程程序执行结果的前提下，尽可能地提高并行度。Java编译器会在生成指令系列时在适当的位置会插入`内存屏障`指令来禁止处理器重排序。插入一个内存屏障，相当于告诉CPU和编译器先于这个命令的必须先执行，后于这个命令的必须后执行。对一个volatile字段进行写操作，Java内存模型将在写操作后插入一个写屏障指令，这个指令会把之前的写入值都刷新到内存。

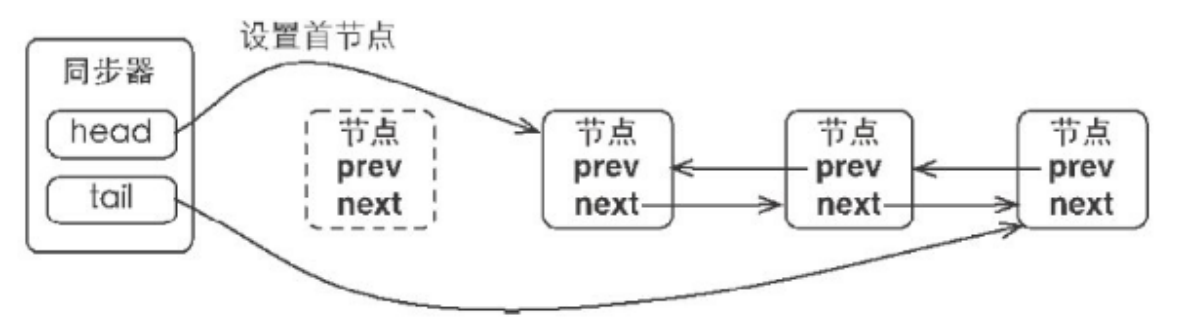
4. AQS原理

AQS，AbstractQueuedSynchronizer，抽象队列同步器，定义了一套多线程访问共享资源的同步器框架，许多并发工具的实现都依赖于它，如常用的ReentrantLock/Semaphore/CountDownLatch。

AQS使用一个volatile的int类型的成员变量state来表示同步状态，通过CAS修改同步状态的值。当线程调用 lock 方法时 ，如果 state=0，说明没有任何线程占有共享资源的锁，可以获得锁并将 state=1。如果 state=1，则说明有线程目前正在使用共享变量，其他线程必须加入同步队列进行等待。

```
private volatile int state;//共享变量，使用volatile修饰保证线程可见性
```

同步器依赖内部的同步队列（一个FIFO双向队列）来完成同步状态的管理，当前线程获取同步状态失败时，同步器会将当前线程以及等待状态（独占或共享 ）构造成为一个节点（Node）并将其加入同步队列并进行自旋，当同步状态释放时，会把首节中的后继节点对应的线程唤醒，使其再次尝试获取同步状态。



5. synchronized的用法有哪些?

1. 修饰普通方法：作用于当前对象实例，进入同步代码前要获得当前对象实例的锁
2. 修饰静态方法：作用于当前类，进入同步代码前要获得当前类对象的锁，synchronized关键字加到static 静态方法和 synchronized(class)代码块上都是是给 Class 类上锁
3. 修饰代码块：指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁

6. Synchronized的作用有哪些?

原子性：确保线程互斥的访问同步代码； 可见性：保证共享变量的修改能够及时可见，其实是通过Java内存模型中的 “对一个变量unlock 操作之前，必须要同步到主内存中；如果对一个变量进行lock操作，则将会清空工作内存中此变量的值，在执行引擎使用此变量前，需要重新从主内存中load操作或assign操作初始化变量值” 来保证的； 有序性：有效解决重排序问题，即 “一个unlock操作先行发生(happen-before)于后面对同一个锁的lock操作”。

7. synchronized 底层实现原理?

synchronized 同步代码块的实现是通过 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置。当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor(monitor对象存在于每个Java对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因) 的持有权。

其内部包含一个计数器，当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行 monitorexit 指令后，将锁计数器设为0 ，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止

synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的确实是 ACC_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法，JVM 通过该 ACC_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

8. ReentrantLock 是如何实现可重入性的?

ReentrantLock 内部自定义了同步器 Sync，在加锁的时候通过 CAS 算法，将线程对象放到一个双向链表中，每次获取锁的时候，检查当前维护的那个线程 ID 和当前请求的线程 ID 是否 一致，如果一致，同步状态加1，表示锁被当前线程获取了多次。

9. ReentrantLock和synchronized区别

1. 使用synchronized关键字实现同步，线程执行完同步代码块会自动释放锁，而ReentrantLock需要手动释放锁。
2. synchronized是非公平锁，ReentrantLock可以设置为公平锁。
3. ReentrantLock上等待获取锁的线程是可中断的，线程可以放弃等待锁。而synchonized会无限期等待下去。
4. ReentrantLock 可以设置超时获取锁。在指定的截止时间之前获取锁，如果截止时间到了还没有获取到锁，则返回。
5. ReentrantLock 的 tryLock() 方法可以尝试非阻塞的获取锁，调用该方法后立刻返回，如果能够获取则返回true，否则返回false。

10. wait()和sleep()的区别

相同点：

1. 使当前线程暂停运行，把机会交给其他线程
2. 任何线程在等待期间被中断都会抛出InterruptedException

不同点：

1. wait() 是Object超类中的方法；而sleep()是线程Thread类中的方法
2. 对锁的持有不同，wait()会释放锁，而sleep()并不释放锁
3. 唤醒方法不完全相同，wait() 依靠notify或者notifyAll 、中断、达到指定时间来唤醒；而sleep()到达指定时间被唤醒
4. 调用obj.wait()需要先获取对象的锁，而 Thread.sleep()不用

11. wait(),notify()和suspend(),resume()之间的区别

- wait() 使得线程进入阻塞等待状态，并且释放锁
- notify()唤醒一个处于等待状态的线程，它一般跟wait（）方法配套使用。
- suspend()使得线程进入阻塞状态，并且不会自动恢复，必须对应的resume() 被调用，才能使得线程重新进入可执行状态。suspend()方法很容易引起死锁问题。

- resume()方法跟suspend()方法配套使用。

suspend()不建议使用,suspend()方法在调用后，线程不会释放已经占有的资 源（比如锁），而是占有着资源进入睡眠状态，这样容易引发死锁问题。

12. Runnable和 Callable有什么区别?

- Callable接口方法是call()，Runnable的方法是run()；
- Callable接口call方法有返回值，支持泛型，Runnable接口run方法无返回值。
- Callable接口call()方法允许抛出异常；而Runnable接口run()方法不能继续上抛异常；

volatile和synchronized的区别是什么?

1. volatile只能使用在变量上；而synchronized可以在类，变量，方法和代码块上。
2. volatile至保证可见性；synchronized保证原子性与可见性。
3. volatile禁用指令重排序；synchronized不会。
4. volatile不会造成阻塞；synchronized会。

13. 线程执行顺序怎么控制?

假设有T1、T2、T3三个线程，你怎样保证T2在T1执行完后执行，T3在T2执行完后执行?

可以使用**join方法**解决这个问题。比如在线程A中，调用线程B的join方法表示的意思就是：**A等待B线程执行完毕后（释放CPU执行权），在继续执行。**

代码如下：

```
public class ThreadTest {

    public static void main(String[] args) {

        Thread spring = new Thread(new SeasonThreadTask("春天"));
        Thread summer = new Thread(new SeasonThreadTask("夏天"));
        Thread autumn = new Thread(new SeasonThreadTask("秋天"));

        try
        {
            //春天线程先启动
            spring.start();
            //主线程等待线程spring执行完，再往下执行
            spring.join();
            //夏天线程再启动
            summer.start();
            //主线程等待线程summer执行完，再往下执行
            summer.join();
            //秋天线程最后启动
            autumn.start();
            //主线程等待线程autumn执行完，再往下执行
            autumn.join();
        } catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}

class SeasonThreadTask implements Runnable{
```

```
    private String name;

    public SeasonThreadTask(String name){
        this.name = name;
    }

    @Override
    public void run() {
        for (int i = 1; i <4; i++) {
            System.out.println(this.name + "来了: " + i + "次");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

运行结果：

```
春天来了: 1次
春天来了: 2次
春天来了: 3次
夏天来了: 1次
夏天来了: 2次
夏天来了: 3次
秋天来了: 1次
秋天来了: 2次
秋天来了: 3次
```

14. 乐观锁一定就是好的吗?

乐观锁避免了悲观锁独占对象的现象，提高了并发性能，但它也有缺点:

- 乐观锁只能保证一个共享变量的原子操作。如果多一个或几个变量，乐观锁将变得力不从心，但互斥锁能轻易解决，不管对象数量多少及对象颗粒度大小。
- 长时间自旋可能导致开销大。假如 CAS 长时间不成功而一直自旋，会 给 CPU 带来很大的开销。
- ABA 问题。CAS 的核心思想是通过比对内存值与预期值是否一样而判 断内存值是否被改过，但这个判断逻辑不严谨，假如内存值原来是 A， 后来被一条线程改为 B，最后又被改成了 A，则 CAS 认为此内存值并 没有发生改变，但实际上是有被其他线程改过的，这种情况对依赖过程值的情景的运算结果影响很大。解决的思路是引入版本号，每次变量更新都把版本号加一。

15. 守护线程是什么?

守护线程是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。在 Java 中垃圾回收线程就是特殊的守护线程。

16. 线程间通信方式

16.1. volatile

volatile是轻量级的同步机制，volatile保证变量对所有线程的可见性，不保证原子性。

16.2. synchronized

保证线程对变量访问的可见性和排他性。

16.3. 等待通知机制

wait/notify为 Object 对象的方法，调用wait/notify需要先获得对象的锁。对象调用wait之后线程释放锁，将线程放到对象的等待队列，当通知线程调用此对象的notify()方法后，等待线程并不会立即从wait返回，需要等待通知线程释放锁（通知线程执行完同步代码块），等待队列里的线程获取锁，获取锁成功才能从wait()方法返回，即从wait方法返回前提是线程获得锁。

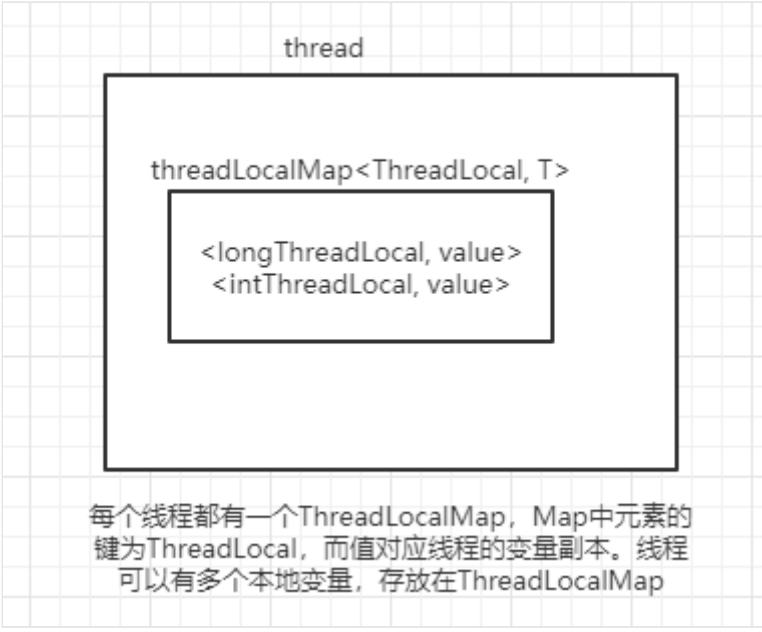
等待通知机制依托于同步机制，目的是确保等待线程从wait方法返回时能感知到通知线程对对象的变量值的修改。

17. ThreadLocal

线程本地变量。当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程。

17.1. ThreadLocal原理

每个线程都有一个ThreadLocalMap(ThreadLocal内部类)，Map中元素的键为ThreadLocal，而值对应线程的变量副本。



调用threadLocal.set()-->调用getMap(Thread)-->返回当前线程的ThreadLocalMap<ThreadLocal, value>-->map.set(this, value)，this是ThreadLocal

```
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
```

```
        createMap(t, value);
}

ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}

void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

调用get()-->调用getMap(Thread)-->返回当前线程的ThreadLocalMap<ThreadLocal, value>-->map.getEntry(this)，返回value

```
    public T get() {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null) {
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null) {
                @SuppressWarnings("unchecked")
                T result = (T)e.value;
                return result;
            }
        }
        return setInitialValue();
    }
```

threadLocals的类型ThreadLocalMap的键为ThreadLocal对象，因为每个线程中可有多个threadLocal变量，如longLocal和stringLocal。

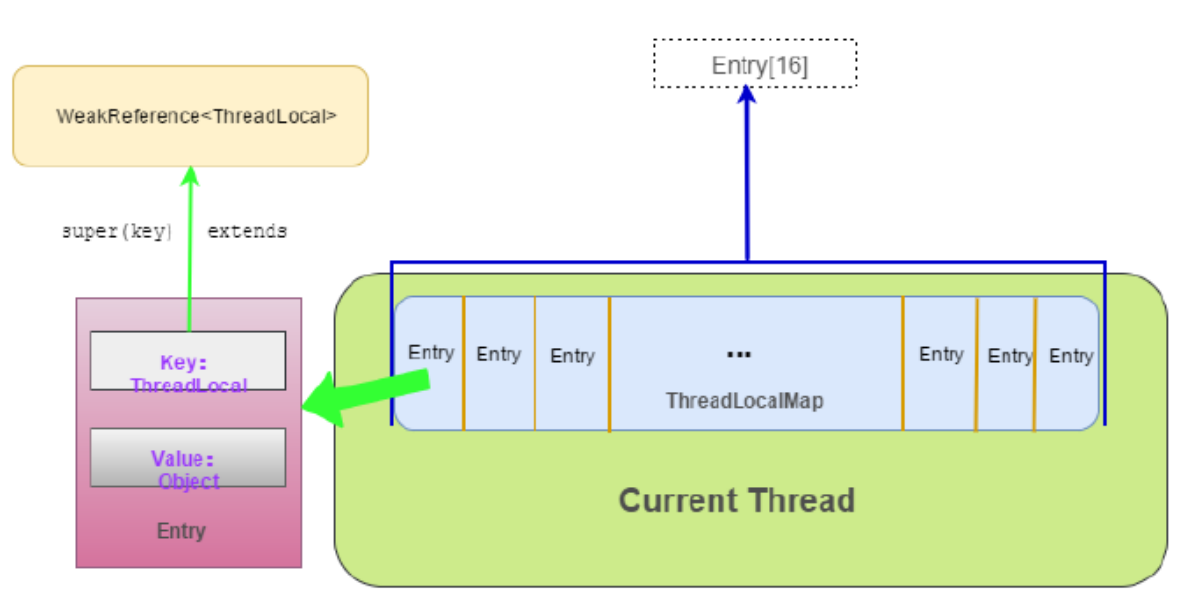
```
public class ThreadLocalDemo {
    ThreadLocal<Long> longLocal = new ThreadLocal<>();

    public void set() {
        longLocal.set(Thread.currentThread().getId());
    }
    public Long get() {
        return longLocal.get();
    }

    public static void main(String[] args) throws InterruptedException {
        ThreadLocalDemo threadLocalDemo = new ThreadLocalDemo();
        threadLocalDemo.set();
        System.out.println(threadLocalDemo.get());

        Thread thread = new Thread(() -> {
            threadLocalDemo.set();
            System.out.println(threadLocalDemo.get());
        }
        );

        thread.start();
        thread.join();

        System.out.println(threadLocalDemo.get());
```

```
    }
}
```

ThreadLocal 并不是用来解决共享资源的多线程访问的问题，因为每个线程中的资源只是副本，并不共享。因此ThreadLocal适合作为线程上下文变量，简化线程内传参。

17.2. ThreadLocal内存泄漏的原因?

每个Thread都有一个ThreadLocalMap的内部属性，map的key是ThreaLocal，定义为弱引用，value是强引用类型。GC的时候会自动回收key，而value的回收取决于Thread对象的生命周期。一般会通过线程池的方式复用Thread对象节省资源，这也就导致了Thread对象的生命周期比较长，这样便一直存在一条强引用链的关系：Thread --> ThreadLocalMap-->Entry-->Value，随着任务的执行，value就有可能越来越多且无法释放，最终导致内存泄漏。



解决方法：每次使用完ThreadLocal就调用它的remove()方法，手动将对应的键值对删除，从而避免内存泄漏。

```
currentTime.set(System.currentTimeMillis());
result = joinPoint.proceed();
Log log = new Log("INFO",System.currentTimeMillis() - currentTime.get());
currentTime.remove();
```

17.3. ThreadLocal使用场景有哪些?

ThreadLocal 适用场景：每个线程需要有自己单独的实例，且需要在多个方法中共享实例，即同时满足实例在线程间的隔离与方法间的共享。比如Java web应用中，每个线程有自己单独的 Session 实例，就可以使用ThreadLocal来实现。

18. 锁的分类

18.1. 公平锁与非公平锁

按照线程访问顺序获取对象锁。synchronized 是非公平锁， Lock 默认是非公平锁，可以设置为公平锁，公平锁会影响性能。

```
public ReentrantLock() {
    sync = new NonfairSync();
}

public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

18.2. 共享式与独占式锁

共享式与独占式的最主要区别在于：同一时刻独占式只能有一个线程获取同步状态，而共享式在同一时刻可以有多个线程获取同步状态。例如读操作可以有多个线程同时进行，而写操作同一时刻只能有一个线程进行写操作，其他操作都会被阻塞。

18.3. 悲观锁与乐观锁

悲观锁，每次访问资源都会加锁，执行完同步代码释放锁，synchronized 和 ReentrantLock 属于悲观锁。

乐观锁，不会锁定资源，所有的线程都能访问并修改同一个资源，如果没有冲突就修改成功并退出，否则就会继续循环尝试。乐观锁最常见的实现就是CAS。

乐观锁一般来说有以下2种方式：

1. 使用数据版本记录机制实现，这是乐观锁最常用的一种实现方式。给数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的version字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。
2. 使用时间戳。数据库表增加一个字段，字段类型使用时间戳（timestamp），和上面的version类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则OK，否则就是版本冲突。

适用场景：

- 悲观锁适合写操作多的场景。
- 乐观锁适合读操作多的场景，不加锁可以提升读操作的性能。

19. CAS

19.1. 什么是CAS?

CAS全称 Compare And Swap，比较与交换，是乐观锁的主要实现方式。CAS 在不使用锁的情况下实现多线程之间的变量同步。ReentrantLock 内部的 AQS 和原子类内部都使用了 CAS。

CAS算法涉及到三个操作数：

- 需要读写的内存值 V。
- 进行比较的值 A。
- 要写入的新值 B。

只有当 V 的值等于 A 时，才会使用原子方式用新值B来更新V的值，否则会继续重试直到成功更新值。

以 AtomicInteger 为例，AtomicInteger 的 getAndIncrement()方法底层就是CAS实现，关键代码是`compareAndSwapInt(obj, offset, expect, update)`，其含义就是，如果`obj`内的`value`和`expect`相等，就证明没有其他线程改变过这个变量，那么就更新它为`update`，如果不相等，那就会继续重试直到成功更新值。

19.2. CAS存在的问题?

CAS 三大问题：

1. **ABA问题。**CAS需要在操作值的时候检查内存值是否发生变化，没有发生变化才会更新内存值。但是如果内存值原来是A，后来变成了B，然后又变成了A，那么CAS进行检查时会发现值没有发生变化，但是实际上是有变化的。ABA问题的解决思路就是在变量前面添加版本号，每次变量更新的时候都把版本号加一，这样变化过程就从`A－B－A`变成了`1A－2B－3A`。

   JDK从1.5开始提供了AtomicStampedReference类来解决ABA问题，原子更新带有版本号的引用类型。

2. **循环时间长开销大。**CAS操作如果长时间不成功，会导致其一直自旋，给CPU带来非常大的开销。

3. **只能保证一个共享变量的原子操作。**对一个共享变量执行操作时，CAS能够保证原子操作，但是对多个共享变量操作时，CAS是无法保证操作的原子性的。

   Java从1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，可以把多个变量放在一个对象里来进行CAS操作。

20. 并发工具

在JDK的并发包里提供了几个非常有用的并发工具类。CountDownLatch、CyclicBarrier和Semaphore工具类提供了一种并发流程控制的手段。

20.1. CountDownLatch

CountDownLatch用于某个线程等待其他线程**执行完任务**再执行，与thread.join()功能类似。常见的应用场景是开启多个线程同时执行某个任务，等到所有任务执行完再执行特定操作，如汇总统计结果。

```
public class CountDownLatchDemo {
    static final int N = 4;
    static CountDownLatch latch = new CountDownLatch(N);

    public static void main(String[] args) throws InterruptedException {

       for(int i = 0; i < N; i++) {
            new Thread(new Thread1()).start();
       }

       latch.await(1000, TimeUnit.MILLISECONDS); //调用await()方法的线程会被挂起，它会等待直到count值为0才继续执行;等待timeout时间后count值还没变为0的话就会继续执行
       System.out.println("task finished");
    }

    static class Thread1 implements Runnable {

        @Override
        public void run() {
            try {
                System.out.println(Thread.currentThread().getName() + "starts working");
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                latch.countDown();
```

```
            }
        }
    }
}
```

运行结果：

```
Thread-0starts working
Thread-1starts working
Thread-2starts working
Thread-3starts working
task finished
```

20.2. CyclicBarrier

CyclicBarrier(同步屏障)，用于一组线程互相等待到某个状态，然后这组线程再**同时**执行。

```
public CyclicBarrier(int parties, Runnable barrierAction) {
}
public CyclicBarrier(int parties) {
}
```

参数parties指让多少个线程或者任务等待至某个状态；参数barrierAction为当这些线程都达到某个状态时会执行的内容。

```
public class CyclicBarrierTest {
    // 请求的数量
    private static final int threadCount = 10;
    // 需要同步的线程数量
    private static final CyclicBarrier cyclicBarrier = new CyclicBarrier(5);

    public static void main(String[] args) throws InterruptedException {
        // 创建线程池
        ExecutorService threadPool = Executors.newFixedThreadPool(10);

        for (int i = 0; i < threadCount; i++) {
            final int threadNum = i;
            Thread.sleep(1000);
            threadPool.execute(() -> {
                try {
                    test(threadNum);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } catch (BrokenBarrierException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            });
        }
        threadPool.shutdown();
    }

    public static void test(int threadnum) throws InterruptedException,
BrokenBarrierException {
```

```
            System.out.println("threadnum:" + threadnum + "is ready");
            try {
                /**等待60秒，保证子线程完全执行结束*/
                cyclicBarrier.await(60, TimeUnit.SECONDS);
            } catch (Exception e) {
                System.out.println("-----CyclicBarrierException------");
            }
            System.out.println("threadnum:" + threadnum + "is finish");
        }

    }
```

运行结果如下，可以看出CyclicBarrier是可以重用的：

```
threadnum:0is ready
threadnum:1is ready
threadnum:2is ready
threadnum:3is ready
threadnum:4is ready
threadnum:4is finish
threadnum:3is finish
threadnum:2is finish
threadnum:1is finish
threadnum:0is finish
threadnum:5is ready
threadnum:6is ready
...
```

当四个线程都到达barrier状态后，会从四个线程中选择一个线程去执行Runnable。

20.3. CyclicBarrier和CountDownLatch区别

CyclicBarrier 和 CountDownLatch 都能够实现线程之间的等待。

CountDownLatch用于某个线程等待其他线程**执行完任务**再执行。CyclicBarrier用于一组线程互相等待到某个状态，然后这组线程再**同时**执行。 CountDownLatch的计数器只能使用一次，而CyclicBarrier的计数器可以使用reset()方法重置，可用于处理更为复杂的业务场景。

20.4. Semaphore

Semaphore类似于锁，它用于控制同时访问特定资源的线程数量，控制并发线程数。

```
public class SemaphoreDemo {
    public static void main(String[] args) {
        final int N = 7;
        Semaphore s = new Semaphore(3);
        for(int i = 0; i < N; i++) {
            new Worker(s, i).start();
        }
    }

    static class Worker extends Thread {
        private Semaphore s;
        private int num;
        public Worker(Semaphore s, int num) {
            this.s = s;
```

```
            this.num = num;
        }

        @Override
        public void run() {
            try {
                s.acquire();
                System.out.println("worker" + num +  " using the machine");
                Thread.sleep(1000);
                System.out.println("worker" + num +  " finished the task");
                s.release();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

运行结果如下，可以看出并非按照线程访问顺序获取资源的锁，即

```
worker0 using the machineworker1 using the machineworker2 using the
machineworker2 finished the taskworker0 finished the taskworker3 using the
machineworker4 using the machineworker1 finished the taskworker6 using the
machineworker4 finished the taskworker3 finished the taskworker6 finished the
taskworker5 using the machineworker5 finished the task
```

21. 原子类

21.1. 基本类型原子类

使用原子的方式更新基本类型

- AtomicInteger：整型原子类
- AtomicLong：长整型原子类
- AtomicBoolean ：布尔型原子类

AtomicInteger 类常用的方法：

```
public final int get() //获取当前的值
public final int getAndSet(int newValue)//获取当前的值，并设置新的值
public final int getAndIncrement()//获取当前的值，并自增
public final int getAndDecrement() //获取当前的值，并自减
public final int getAndAdd(int delta) //获取当前的值，并加上预期的值
boolean compareAndSet(int expect, int update) //如果输入的数值等于预期值，则以原子方式
将该值设置为输入值（update）
public final void lazySet(int newValue)//最终设置为newValue,使用 lazySet 设置之后可能
导致其他线程在之后的一小段时间内还是可以读到旧的值。
```

AtomicInteger 类主要利用 CAS (compare and swap) 保证原子操作，从而避免加锁的高开销。

21.2. 数组类型原子类

使用原子的方式更新数组里的某个元素

- AtomicIntegerArray：整形数组原子类
- AtomicLongArray：长整形数组原子类
- AtomicReferenceArray ：引用类型数组原子类

AtomicIntegerArray 类常用方法：

```
public final int get(int i) //获取 index=i 位置元素的值
public final int getAndSet(int i, int newValue)//返回 index=i 位置的当前的值，并将其设置为新值：newValue
public final int getAndIncrement(int i)//获取 index=i 位置元素的值，并让该位置的元素自增
public final int getAndDecrement(int i) //获取 index=i 位置元素的值，并让该位置的元素自减
public final int getAndAdd(int i, int delta) //获取 index=i 位置元素的值，并加上预期的值
boolean compareAndSet(int i, int expect, int update) //如果输入的数值等于预期值，则以原子方式将 index=i 位置的元素值设置为输入值（update）
public final void lazySet(int i, int newValue)//最终 将index=i 位置的元素设置为newValue,使用 lazySet 设置之后可能导致其他线程在之后的一小段时间内还是可以读到旧的值。
```

21.3. 引用类型原子类

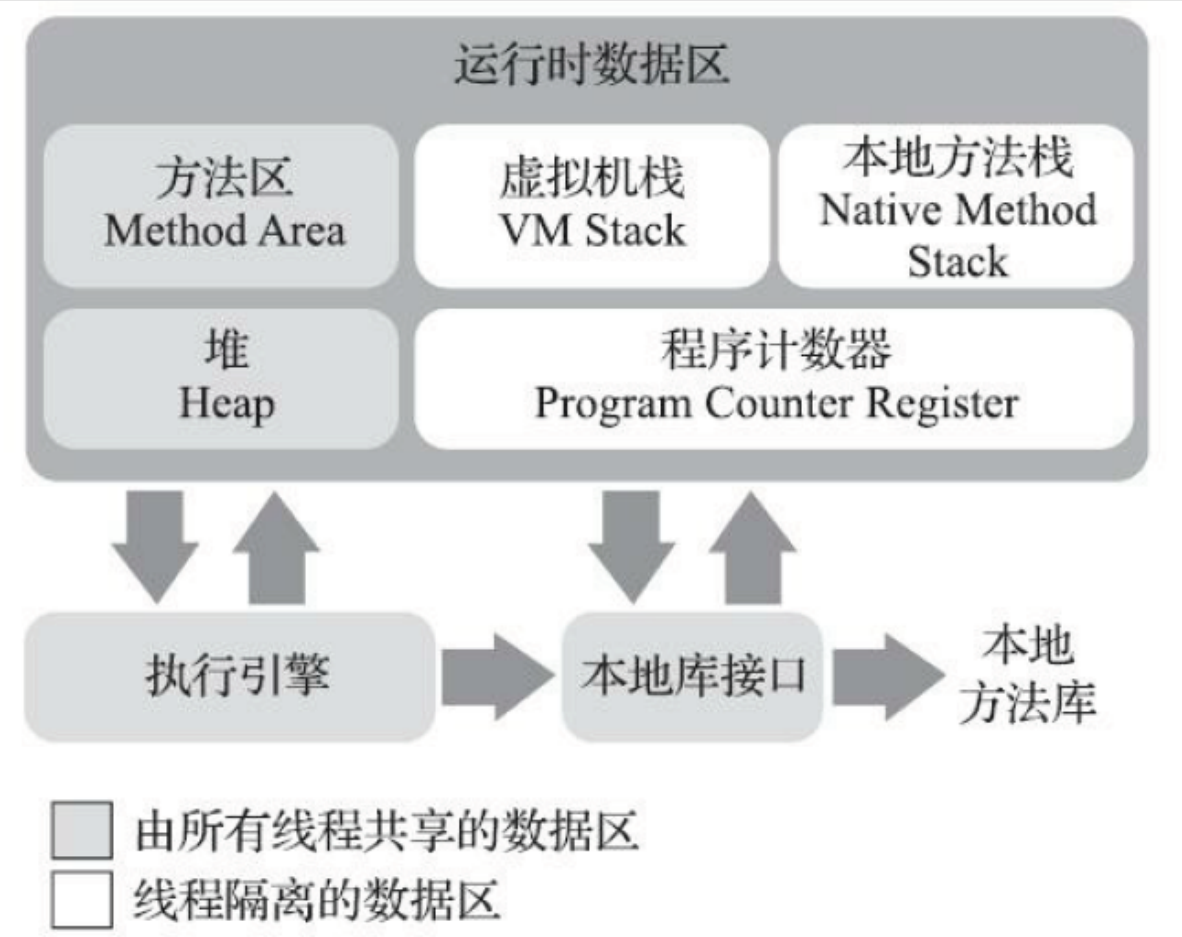
- AtomicReference：引用类型原子类
- AtomicStampedReference：带有版本号的引用类型原子类。该类将整数值与引用关联起来，可用于解决原子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。
- AtomicMarkableReference ： 原子更新带有标记的引用类型。该类将 boolean 标记与引用关联起来

JVM

1. 讲一下JVM内存结构?

Java 内存模型（JMM）是基于共享内存的多线程通信机制。

JVM内存结构 = 类加载器 + 执行引擎 + 运行时数据区域 。



> 图片来源：深入理解Java虚拟机-周志明

1.1. 程序计数器

程序计数器主要有两个作用：

1. 当前线程所执行的字节码的行号指示器，通过改变它实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
2. 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

程序计数器是唯一一个不会出现 `OutOfMemoryError` 的内存区域，它的生命周期随着线程的创建而创建，随着线程的结束而死亡。

1.2. 虚拟机栈

Java 虚拟机栈是由一个个栈帧组成，而每个栈帧中都拥有：局部变量表、操作数栈、动态链接、方法出口信息。每一次函数调用都会有一个对应的栈帧被压入虚拟机栈，每一个函数调用结束后，都会有一个栈帧被弹出。

局部变量表是用于存放方法参数和方法内的局部变量。

每个栈帧都包含一个指向运行时常量池中该栈所属方法的符号引用，在方法调用过程中，会进行动态链接，将这个符号引用转化为直接引用。

- 部分符号引用在类加载阶段的时候就转化为直接引用，这种转化就是静态链接
- 部分符号引用在运行期间转化为直接引用，这种转化就是动态链接

Java 虚拟机栈也是线程私有的，每个线程都有各自的 Java 虚拟机栈，而且随着线程的创建而创建，随着线程的死亡而死亡。Java 虚拟机栈会出现两种错误：`StackOverFlowError` 和 `OutOfMemoryError`。

可以通过 -Xss 参数来指定每个线程的 Java 虚拟机栈内存大小，在 JDK 1.4 中默认为 256K，而在 JDK 1.5+ 默认为 1M：

```
java -Xss2M
```

1.3. 本地方法栈

虚拟机栈为虚拟机执行 Java 方法服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。Native 方法一般是用其它语言（C、C++ 或汇编语言等）编写的，并且被编译为基于本机硬件和操作系统的程序，对待这些方法需要特别处理。

本地方法被执行的时候，在本地方法栈也会创建一个栈帧，用于存放该本地方法的局部变量表、操作数栈、动态链接、出口信息。

1.4. 堆

此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例以及数组都在这里分配内存。Java 堆是垃圾收集器管理的主要区域，因此也被称作GC堆。

Java 堆可以细分为：新生代（Eden 空间、From Survivor、To Survivor 空间）和老年代。进一步划分的目的是更好地回收内存，或者更快地分配内存。

通过 -Xms设定程序启动时占用内存大小，通过 -Xmx 设定程序运行期间最大可占用的内存大小。如果程序运行需要占用更多的内存，超出了这个设置值，就会抛出OutOfMemory异常。

```
java -Xms1M -Xmx2M
```

通过 -Xss 设定每个线程的堆栈大小。设置这个参数，需要评估一个线程大约需要占用多少内存，可能会有多少线程同时运行等。

> 在这里也给大家分享一个github仓库，上面放了**200多本经典的计算机书籍**，包括C语言、C++、Java、Python、前端、数据库、操作系统、计算机网络、数据结构和算法、机器学习、编程人生等，可以star一下，下次找书直接在上面搜索，仓库持续更新中~
>
> github地址：https://github.com/Tyson0314/java-books
>
> 如果github访问不了，可以访问gitee仓库。
>
> gitee地址：https://gitee.com/tysondai/java-books

1.5. 方法区

方法区与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。方法区逻辑上属于堆的一部分。

对方法区进行垃圾回收的主要目标是对常量池的回收和对类的卸载。

永久代

方法区是 JVM 的规范，而永久代（PermGen）是方法区的一种实现方式，并且只有 HotSpot 有永久代。而对于其他类型的虚拟机，如 JRockit（Oracle）、J9（IBM） 并没有永久代。由于方法区主要存储类的相关信息，所以对于动态生成类的情况比较容易出现永久代的内存溢出。最典型的场景就是，在 jsp 页面比较多的情况，容易出现永久代内存溢出。

元空间

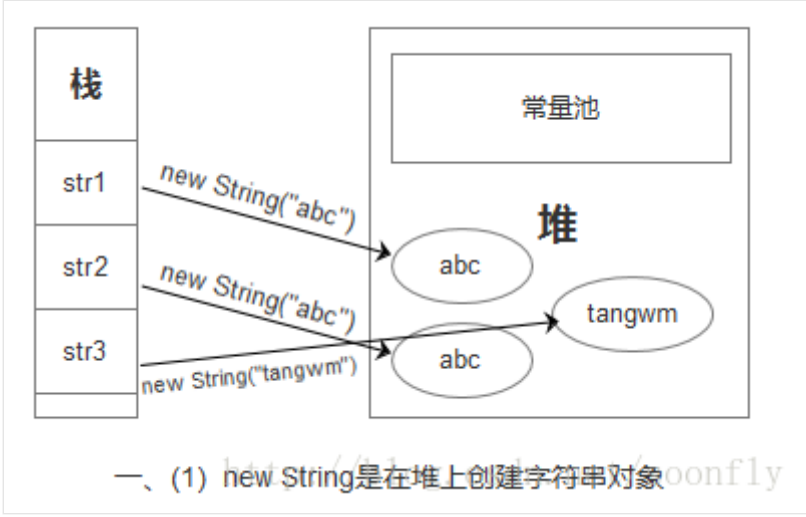
JDK 1.8 的时候，HotSpot 的永久代被彻底移除了，使用元空间替代。元空间的本质和永久代类似，都是对JVM规范中方法区的实现。两者最大的区别在于：元空间并不在虚拟机中，而是使用直接内存。

为什么要将永久代替换为元空间呢?

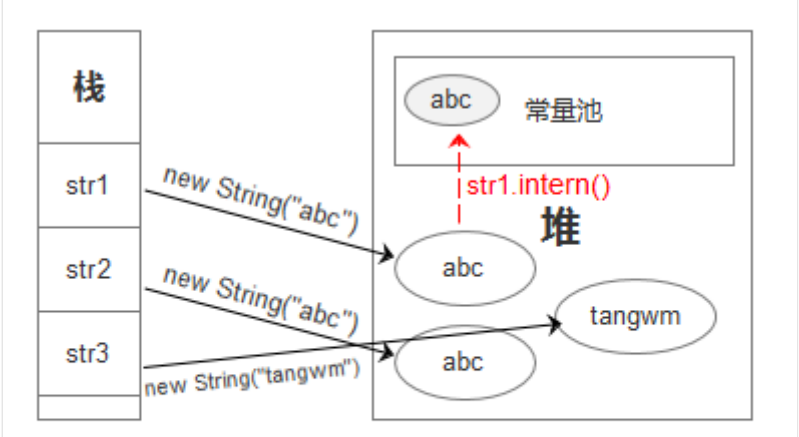
永久代内存受限于 JVM 可用内存，而元空间使用的是直接内存，受本机可用内存的限制，虽然元空间仍旧可能溢出，但是相比永久代内存溢出的概率更小。

1.6. 运行时常量池

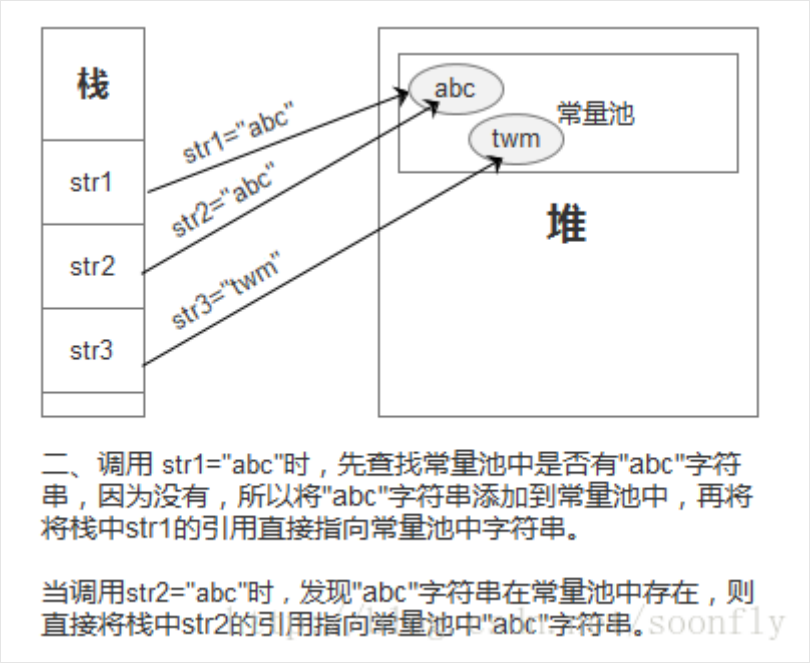
运行时常量池是方法区的一部分，在类加载之后，会将编译器生成的各种字面量和符号引号放到运行时常量池。在运行期间动态生成的常量，如 String 类的 intern()方法，也会被放入运行时常量池。



一、(1) new String是在堆上创建字符串对象



一、(2) 调用 str1.intern() ，编译器会将"abc"字符串添加到常量池中，并返回指向该常量的引用。如果此后再调用str2.intern()，因为常量池中已有"abc"存在，因此就直接返回常量池中"abc"的引用。



> 图片来源：https://blog.csdn.net/soonfly

1.7. 直接内存

直接内存并不是虚拟机运行时数据区的一部分，也不是虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用。而且也可能导致 OutOfMemoryError 错误出现。

NIO的Buffer提供了DirectBuffer，可以直接访问系统物理内存，避免堆内内存到堆外内存的数据拷贝操作，提高效率。DirectBuffer直接分配在物理内存中，并不占用堆空间，其可申请的最大内存受操作系统限制，不受最大堆内存的限制。

直接内存的读写操作比堆内存快，可以提升程序I/O操作的性能。通常在I/O通信过程中，会存在堆内内存到堆外内存的数据拷贝操作，对于需要频繁进行内存间数据拷贝且生命周期较短的暂存数据，都建议存储到直接内存。

2. Java对象的定位方式

Java 程序通过栈上的 reference 数据来操作堆上的具体对象。对象的访问方式由虚拟机实现而定，目前主流的访问方式有使用句柄和直接指针两种：

- 如果使用句柄的话，那么 Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。使用句柄来访问的最大好处是 reference 中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而 reference 本身不需要修改。
- 直接指针。reference 中存储的直接就是对象的地址。对象包含到对象类型数据的指针，通过这个指针可以访问对象类型数据。使用直接指针访问方式最大的好处就是访问对象速度快，它节省了一次指针定位的时间开销，虚拟机hotspot主要是使用直接指针来访问对象。

3. 说一下堆栈的区别?

物理地址

堆的物理地址分配对对象是不连续的。在GC的时候需要考虑到不连续的分配，性能较慢。

栈使用的是数据结构中的栈，先进后出的原则，物理地址分配是连续的。性能快。

存放的内容

堆存放的是对象的实例和数组。

栈存放：局部变量，操作数栈，返回结果。

程序的可见度

堆对于整个应用程序都是共享、可见的。

栈只对于线程是可见的，线程私有。

4. 什么情况下会发生栈内存溢出?

当线程请求的栈深度超过了虚拟机允许的最大深度时，会抛出StackOverFlowError异常。通过调整参数`-xss`可以调整JVM栈的大小。

5. 类文件结构

Class 文件结构如下：

```
ClassFile {
    u4             magic; //Class 文件的标志
    u2             minor_version;//Class 的小版本号
    u2             major_version;//Class 的大版本号
    u2             constant_pool_count;//常量池的数量
    cp_info        constant_pool[constant_pool_count-1];//常量池
    u2             access_flags;//Class 的访问标记
    u2             this_class;//当前类
    u2             super_class;//父类
    u2             interfaces_count;//接口
    u2             interfaces[interfaces_count];//一个类可以实现多个接口
    u2             fields_count;//Class 文件的字段属性
    field_info     fields[fields_count];//一个类会可以有个字段
    u2             methods_count;//Class 文件的方法数量
    method_info    methods[methods_count];//一个类可以有个多个方法
    u2             attributes_count;//此类的属性表中的属性数
    attribute_info attributes[attributes_count];//属性表集合
}
```

魔数：class 文件标志。

文件版本：高版本的 Java 虚拟机可以执行低版本编译器生成的 Class 文件，但是低版本的 Java 虚拟机不能执行高版本编译器生成的 Class 文件。

常量池：存放字面量和符号引用。字面量类似于Java的常量，如字符串，声明为final的常量值等。符号引用包含三类：类和接口的全限定名，方法的名称和描述符，字段的名称和描述符。

访问标志：识别一些类或者接口层次的访问信息，包括：这个 Class 是类还是接口，是否为 public 或者 abstract 类型，如果是类的话是否声明为 final 等等。

当前类的索引this_class：类索引用于确定这个类的全限定名。

属性表集合：在 Class 文件，字段表，方法表中都可以携带自己的属性表集合，以用于描述某些场景专有的信息。与 Class 文件中其它的数据项目要求的顺序、长度和内容不同，属性表集合的限制稍微宽松一些，不再要求各个属性表具有严格的顺序，并且只要不与已有的属性名重复，任何人实现的编译器都可以向属性表中写 入自己定义的属性信息，Java 虚拟机运行时会忽略掉它不认识的属性。

6. 什么是类加载？类加载的过程?

类的加载指的是将类的class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个对象，这个对象封装了类在方法区内的数据结构，并且提供了访问方法区内的类信息的接口。

加载

类加载过程的一个阶段：通过一个类的完全限定查找此类字节码文件，并利用字节码文件创建一个Class对象。

1. 通过全类名获取定义此类的二进制字节流
2. 将字节流所代表的静态存储结构转换为方法区的运行时数据结构
3. 在内存中生成一个代表该类的 Class 对象，作为方法区这些数据的访问入口

验证

确保Class文件的字节流中包含的信息符合虚拟机规范，保证这些信息被当作代码运行后不会危害虚拟机自身的安全。主要包括四种验证：文件格式验证，元数据验证，字节码验证，符号引用验证。

准备

为类变量分配内存并设置类变量初始值的阶段。此阶段进行内存分配的仅包括类变量，不包括实例变量和final修饰的static变量（因为final在编译的时候就会分配了），实例变量会在对象实例化时随着对象一块分配在 Java 堆中。

解析

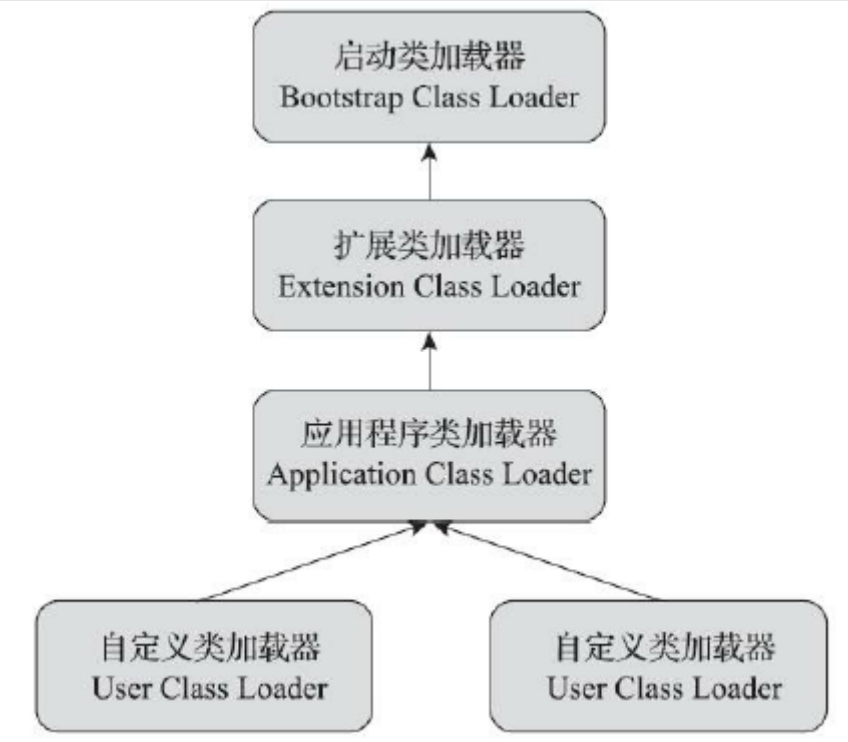
虚拟机将常量池内的符号引用替换为直接引用的过程。符号引用用于描述目标，直接引用直接指向目标的地址。

初始化

初始化阶段就是执行类构造器<clinit>()方法的过程。<clinit>()并不是程序员在Java代码中直接编写的方法，它是Javac编译器的自动生成的，由编译器自动收集类中的所有类变量的赋值动作和静态语句块中的语句合并产生的。

7. 什么是双亲委派模型?

一个类加载器收到一个类的加载请求时，它首先不会自己尝试去加载它，而是把这个请求委派给父类加载器去完成，这样层层委派，因此所有的加载请求最终都会传送到顶层的启动类加载器中，只有当父类加载器反馈自己无法完成这个加载请求时，子加载器才会尝试自己去加载。



双亲委派模型的具体实现代码在抽象类 java.lang.ClassLoader 中，此类的 loadClass() 方法运行过程如下：先检查类是否已经加载过，如果没有则让父类加载器去加载。当父类加载器加载失败时抛出 ClassNotFoundException，此时尝试自己去加载。源码如下：

```
public abstract class ClassLoader {
    // The parent class loader for delegation
    private final ClassLoader parent;

    public Class<?> loadClass(String name) throws ClassNotFoundException {
        return loadClass(name, false);
    }

    protected Class<?> loadClass(String name, boolean resolve) throws
ClassNotFoundException {
        synchronized (getClassLoadingLock(name)) {
            // First, check if the class has already been loaded
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                try {
                    if (parent != null) {
                        c = parent.loadClass(name, false);
                    } else {
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    // ClassNotFoundException thrown if class not found
                    // from the non-null parent class loader
                }

                if (c == null) {
                    // If still not found, then invoke findClass in order
                    // to find the class.
                    c = findClass(name);
                }
            }
            if (resolve) {
```

```
                resolveClass(c);
            }
            return c;
        }
    }

    protected Class<?> findClass(String name) throws ClassNotFoundException {
        throw new ClassNotFoundException(name);
    }
}
```

8. 为什么需要双亲委派模型?

双亲委派模型的好处：可以防止内存中出现多份同样的字节码。如果没有双亲委派模型而是由各个类加载器自行加载的话，如果用户编写了一个java.lang.Object的同名类并放在ClassPath中，多个类加载器都去加载这个类到内存中，系统中将会出现多个不同的Object类，那么类之间的比较结果及类的唯一性将无法保证。

9. 什么是类加载器，类加载器有哪些?

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器:

- 启动类加载器：用来加载 Java 核心类库，无法被 Java 程序直接引用。
- 扩展类加载器：它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
- 系统类加载器：它根据应用的类路径来加载 Java 类。可通过 ClassLoader.getSystemClassLoader()来获取它。
- 用户自定义类加载器：通过继承 java.lang.ClassLoader类的方式实现。

10. 类的实例化顺序?

1. 父类中的static代码块，当前类的static
2. 顺序执行父类的普通代码块
3. 父类的构造函数
4. 当前类普通代码块
5. 当前类的构造函数

11. 如何判断一个对象是否存活?

堆中几乎放着所有的对象实例，对堆垃圾回收前的第一步就是要判断那些对象已经死亡（即不能再被任何途径使用的对象）。判断对象是否存活有两种方法：引用计数法和可达性分析。

引用计数法

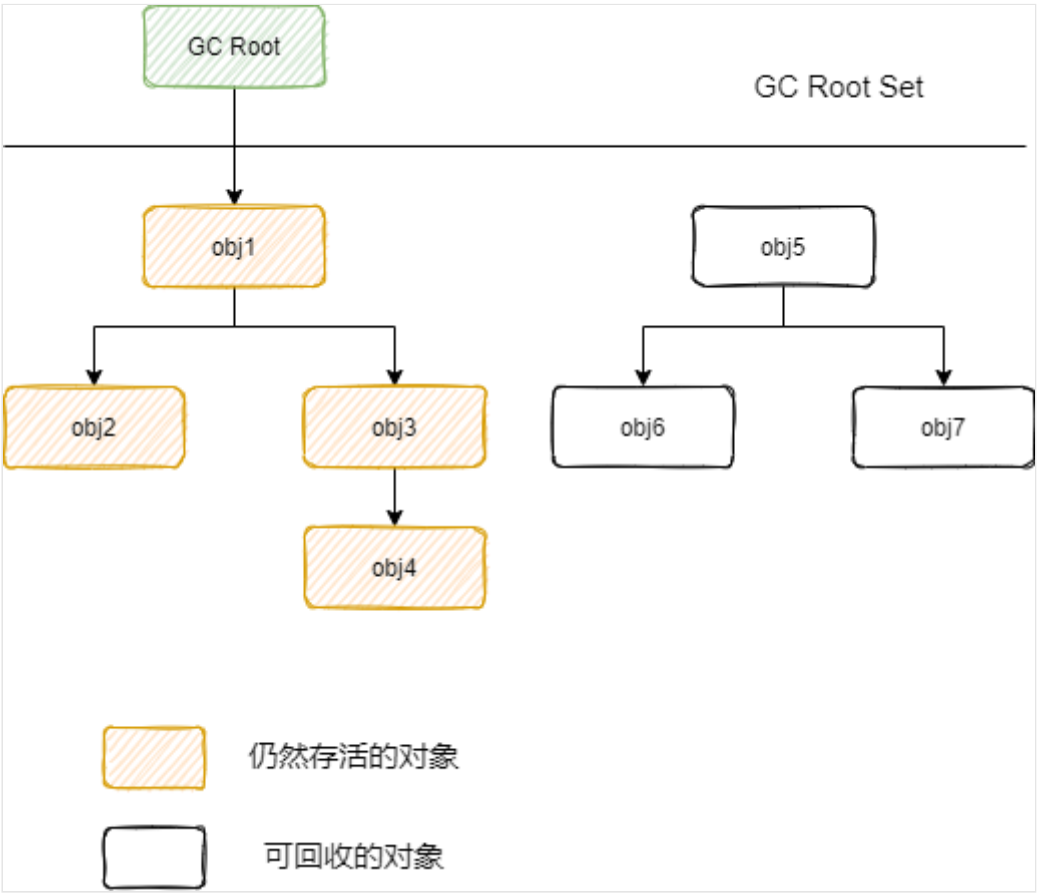
给对象中添加一个引用计数器，每当有一个地方引用它，计数器就加 1；当引用失效，计数器就减 1；任何时候计数器为 0 的对象就是不可能再被使用的。

这种方法很难解决对象之间相互循环引用的问题。比如下面的代码，objA 和 objB 互相引用，这种情况下，引用计数器的值都是1，不会被垃圾回收。

```
public class ReferenceCountingGc {
    Object instance = null;
    public static void main(String[] args) {
        ReferenceCountingGc objA = new ReferenceCountingGc();
        ReferenceCountingGc objB = new ReferenceCountingGc();
        objA.instance = objB;
        objB.instance = objA;
        objA = null;
        objB = null;
    }
}
```

可达性分析

通过GC Root对象为起点，从这些节点向下搜索，搜索所走过的路径叫引用链，当一个对象到GC Root没有任何的引用链相连时，说明这个对象是不可用的。



12. 可作为GC Roots的对象有哪些?

1. 虚拟机栈(栈帧中的本地变量表)中引用的对象
2. 本地方法栈中JNI（Native方法）引用的对象
3. 方法区中类静态属性引用的对象
4. 方法区中常量引用的对象
5. 所有被同步锁（synchronized关键字）持有的对象。

13. 什么情况下类会被卸载?

需要同时满足下面 3 个条件才算是 “无用的类” ：

- 该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
- 加载该类的 ClassLoader 已经被回收。

- 该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

虚拟机可以对满足上述 3 个条件的类进行回收，但不一定被回收。

14. 强引用、软引用、弱引用、虚引用是什么，有什么区别?

强引用：垃圾回收器绝不会回收它。当内存空间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

软引用：如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

弱引用：在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。

虚引用：虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。**虚引用主要用来跟踪对象被垃圾回收的活动。**

在程序设计中一般很少使用弱引用与虚引用，使用软引用的情况较多，这是因为软引用可以加速 JVM 对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出等问题的产生。

15. Minor GC 和 Full GC的区别?

- Minor GC：回收新生代，因为新生代对象存活时间很短，因此 Minor GC 会频繁执行，执行的速度一般也会比较快。
- Full GC：回收老年代和新生代，老年代对象其存活时间长，因此 Full GC 很少执行，执行速度会比 Minor GC 慢很多。

16. 内存的分配策略?

对象优先在 Eden 分配

大多数情况下，对象在新生代 Eden 上分配，当 Eden 空间不够时，发起 Minor GC。

大对象直接进入老年代

大对象是指需要连续内存空间的对象，最典型的大对象是那种很长的字符串以及数组。经常出现大对象会提前触发垃圾收集以获取足够的连续空间分配给大对象。

可以设置JVM参数 -XX:PretenureSizeThreshold，大于此值的对象直接在老年代分配，避免在 Eden 和 Survivor 之间的大量内存复制。

长期存活的对象进入老年代

通过参数 `-XX:MaxTenuringThreshold` 可以设置对象进入老年代的年龄阈值。对象在 Survivor 中每经过一次 MinorGC，年龄就增加 1 岁，当它的年龄增加到一定程度，就会被晋升到老年代中。

动态对象年龄判定

虚拟机并不是永远要求对象的年龄必须达到 MaxTenuringThreshold 才能晋升老年代，如果在 Survivor 中相同年龄所有对象大小的总和大于 Survivor 空间的一半，则年龄大于或等于该年龄的对象可以直接进入老年代，无需等到 MaxTenuringThreshold 中要求的年龄。

空间分配担保

在发生 Minor GC 之前，虚拟机先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果条件成立的话，那么 Minor GC 可以确认是安全的。如果不成立的话虚拟机会查看 HandlePromotionFailure 的值是否允许担保失败。如果允许，那么就会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次 Minor GC；如果小于，或者 HandlePromotionFailure 的值不允许冒险，那么就要进行一次 Full GC。

17. Full GC 的触发条件?

对于 Minor GC，其触发条件比较简单，当 Eden 空间满时，就将触发一次 Minor GC。而 Full GC 触发条件相对复杂，有以下情况会发生 full GC：

调用 System.gc()

只是建议虚拟机执行 Full GC，但是虚拟机不一定真正去执行。不建议使用这种方式，而是让虚拟机管理内存。

老年代空间不足

老年代空间不足的常见场景为前文所讲的大对象直接进入老年代、长期存活的对象进入老年代等。为了避免以上原因引起的 Full GC，应当尽量不要创建过大的对象以及数组。除此之外，可以通过 -Xmn 参数调大新生代的大小，让对象尽量在新生代被回收掉，不进入老年代。还可以通过 -XX:MaxTenuringThreshold 调大对象进入老年代的年龄，让对象在新生代多存活一段时间。

空间分配担保失败

使用复制算法的 Minor GC 需要老年代的内存空间作担保，如果担保失败会执行一次 Full GC。

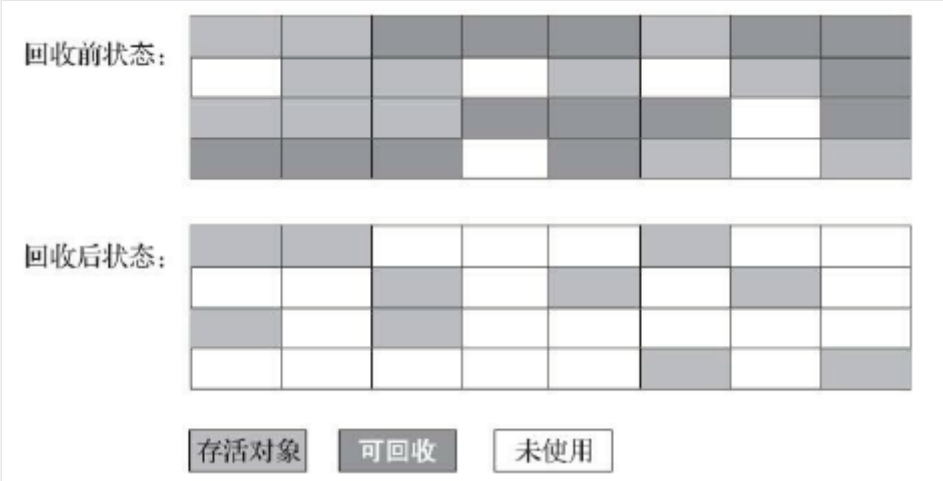
JDK 1.7 及以前的永久代空间不足

在 JDK 1.7 及以前，HotSpot 虚拟机中的方法区是用永久代实现的，永久代中存放的为一些 Class 的信息、常量、静态变量等数据。当系统中要加载的类、反射的类和调用的方法较多时，永久代可能会被占满，在未配置为采用 CMS GC 的情况下也会执行 Full GC。如果经过 Full GC 仍然回收不了，那么虚拟机会抛出 java.lang.OutOfMemoryError。

18. 垃圾回收算法有哪些?

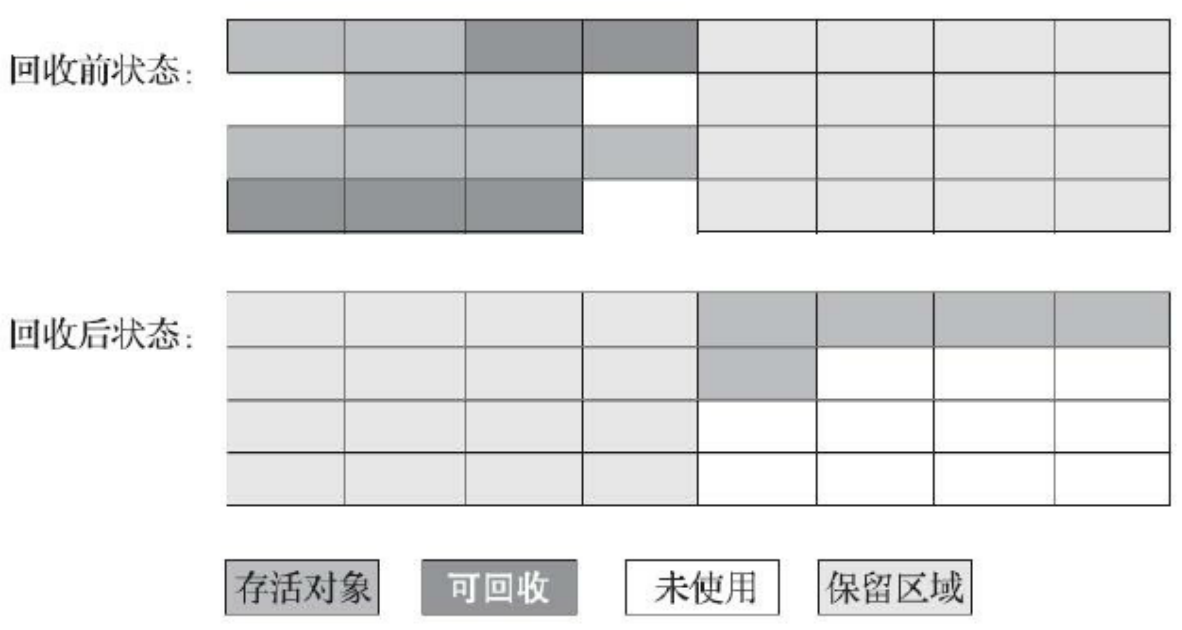
标记清除算法

标记清除算法就是分为“标记”和“清除”两个阶段。标记出所有需要回收的对象，标记结束后统一回收所有被标记的对象。这种垃圾回收算法效率较低，并且会产生大量不连续的空间碎片。



复制清除算法

半区复制，用于新生代垃圾回收。将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。



特点：实现简单，运行高效，但可用内存缩小为了原来的一半，浪费空间。

标记整理算法

根据老年代的特点提出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉边界以外的内存。

分类收集算法

根据各个年代的特点采用最适当的收集算法。

一般将堆分为新生代和老年代。

- 新生代使用：复制算法
- 老年代使用：标记清除算法或者标记整理算法

在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或者“标记-整理”算法来进行回收。

由于对象之间会存在跨代引用，如果要进行一次新生代垃圾收集，除了需要遍历新生代对象，还要额外遍历整个老年代的所有对象，这会给内存回收带来很大的性能负担。

跨代引用相对于同代引用来说仅占极少数。存在互相引用关系的两个对象，是应该倾向于同时生存或者同时消亡的。举个例子，如果某个新生代对象存在跨代引用，由于老年代对象难以消亡，该引用会使得新生代对象在收集时同样得以存活，进而在年龄增长之后晋升到老年代中，这时跨代引用也随即被消除了。

所以没必要为了少量的跨代引用去扫描整个老年代，只需在新生代建立一个全局的数据结构Remembered Set，这个结构把老年代划分成若干小块，标识出老年代的哪一块内存会存在跨代引用。此后当发生Minor GC时，只有包含了跨代引用的小块内存里的对象才会被加入到GC Roots进行扫描。

19. 有哪几种垃圾回收器，各自的优缺点是什么？

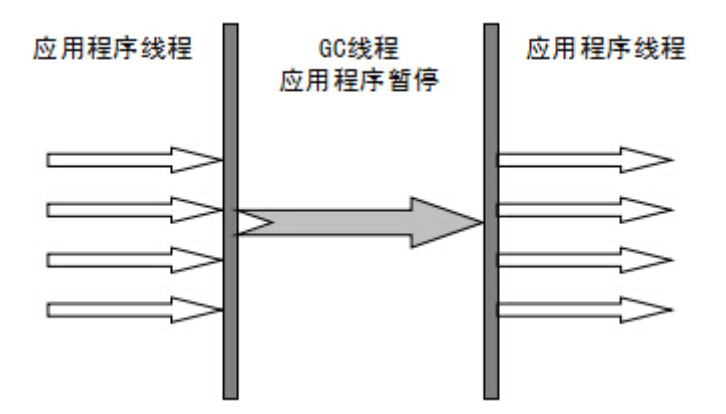
垃圾回收器主要分为以下几种：Serial、ParNew、Parallel Scavenge、Serial Old、Parallel Old、CMS、G1。

这7种垃圾收集器的特点：

收集器	串行、并行or并发	新生代/老年代	算法	目标	适用场景
Serial	串行	新生代	复制算法	响应速度优先	单CPU环境下的Client模式
ParNew	并行	新生代	复制算法	响应速度优先	多CPU环境时在Server模式下与CMS配合
Parallel Scavenge	并行	新生代	复制算法	吞吐量优先	在后台运算而不需要太多交互的任务
Serial Old	串行	老年代	标记-整理	响应速度优先	单CPU环境下的Client模式、CMS的后备预案
Parallel Old	并行	老年代	标记-整理	吞吐量优先	在后台运算而不需要太多交互的任务
CMS	并发	老年代	标记-清除	响应速度优先	集中在互联网站或B/S系统服务端上的Java应用
G1	并发	both	标记-整理+复制算法	响应速度优先	面向服务端应用，将来替换CMS

Serial 收集器

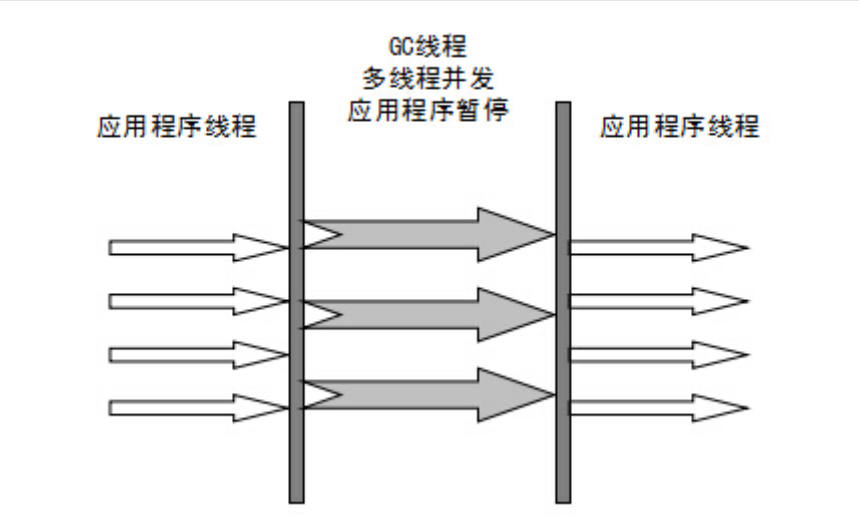
单线程收集器，使用一条垃圾收集线程去完成垃圾收集工作，在进行垃圾收集工作的时候必须暂停其他所有的工作线程（ "Stop The World" ），直到它收集结束。



特点：简单高效；内存消耗最小；没有线程交互的开销，单线程收集效率高；需暂停所有的工作线程，用户体验不好。

ParNew 收集器

Serial 收集器的多线程版本，除了使用多线程进行垃圾收集外，其余行为（控制参数、收集算法、回收策略等等）和 Serial 收集器完全一样。



除了 Serial 收集器外，只有它能与 CMS 收集器配合工作。

Parallel Scavenge 收集器

新生代收集器，基于复制清除算法实现的收集器。吞吐量优先收集器，也是能够并行收集的多线程收集器，允许多个垃圾回收线程同时运行，降低垃圾收集时间，提高吞吐量。所谓吞吐量就是 CPU 中用于运行用户代码的时间与 CPU 总消耗时间的比值（吞吐量 = 运行用户代码时间 /（运行用户代码时间 + 垃圾收集时间））。Parallel Scavenge 收集器关注点是吞吐量，高效率的利用 CPU 资源。CMS 垃圾收集器关注点更多的是用户线程的停顿时间。

Parallel Scavenge收集器提供了两个参数用于**精确控制吞吐量**，分别是控制最大垃圾收集停顿时间的-XX：MaxGCPauseMillis参数以及直接设置吞吐量大小的-XX：GCTimeRatio参数。

-XX：MaxGCPauseMillis参数允许的值是一个大于0的毫秒数，收集器将尽力保证内存回收花费的时间不超过用户设定值。

-XX：GCTimeRatio参数的值则应当是一个大于0小于100的整数，也就是垃圾收集时间占总时间的比率，相当于吞吐量的倒数。

相比ParNew收集器，Parallel Scavenge 的优点：

1. 精确控制吞吐量；
2. 垃圾收集的自适应的调节策略。通过参数-XX：+UseAdaptiveSizePolicy 打开自适应调节策略，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整参数以提供最合适的停顿时间或者最大的吞吐量。调整的参数包括新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX：SurvivorRatio）、晋升老年代对象大小（-XX：PretenureSizeThreshold）等。

Serial Old 收集器

Serial 收集器的老年代版本，它同样是一个单线程收集器，使用标记整理算法。它主要有两大用途：一种用途是在 JDK1.5 以及以前的版本中与 Parallel Scavenge 收集器搭配使用，另一种用途是作为 CMS 收集器的后备方案。

Parallel Old 收集器

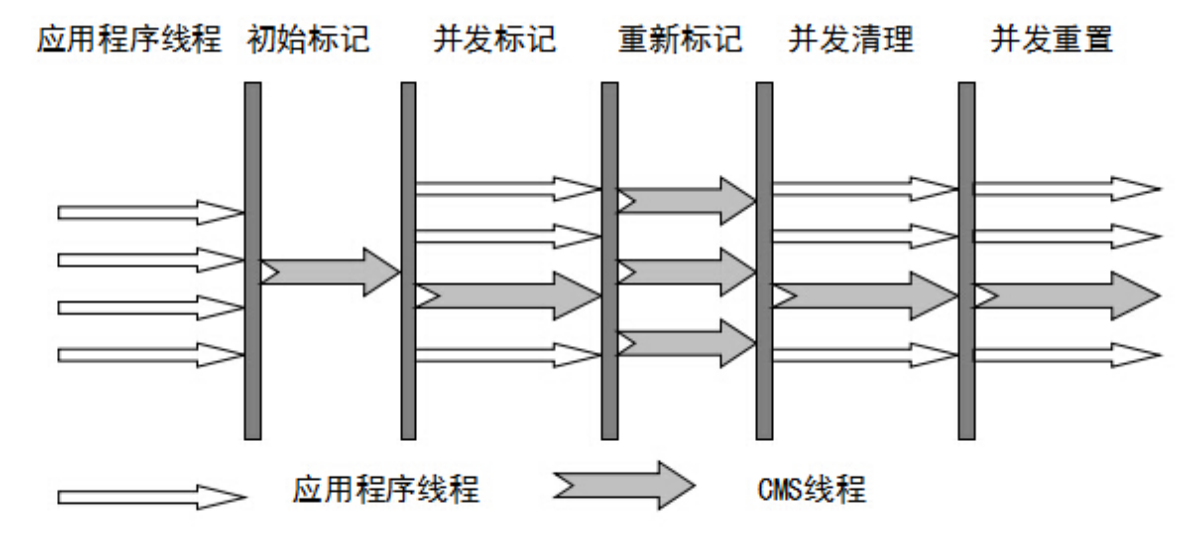
Parallel Scavenge 收集器的老年代版本。多线程垃圾收集，使用标记-整理算法。在注重吞吐量以及 CPU 资源的场合，都可以优先考虑 Parallel Scavenge 收集器和 Parallel Old 收集器。

CMS 收集器

Concurrent Mark Sweep 并发标记清除，目的是获取最短应用停顿时间。第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程基本上同时工作。在并发标记和并发清除阶段，虽然用户线程没有被暂停，但是由于垃圾收集器线程占用了一部分系统资源，应用程序的吞吐量会降低。

CMS 垃圾回收基于标记清除算法实现，整个过程分为四个步骤：

- 初始标记： stw暂停所有的其他线程，记录直接与 gc root 直接相连的对象，速度很快 。
- 并发标记：从GC Roots开始对堆中对象进行可达性分析，找出存活对象，耗时较长，但是不需要停顿用户线程。
- 重新标记： 在并发标记期间对象的引用关系可能会变化，需要重新进行标记。此阶段也会stw，停顿时间一般会比初始标记阶段的时间稍长，远远比并发标记阶段时间短。
- 并发清除：清除死亡对象，由于不需要移动存活对象，所以这个阶段也是可以与用户线程同时并发的。



由于在整个过程中耗时最长的并发标记和并发清除阶段中，垃圾收集器线程都可以与用户线程一起工作，所以从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

优点：并发收集，低停顿。

缺点：

- 标记清除算法导致收集结束有大量空间碎片，往往出现老年代空间剩余，但无法找到足够大连续空间来分配当前对象，不得不提前触发一次 Full GC。
- 会产生浮动垃圾，由于CMS并发清理阶段用户线程还在运行着，会不断有新的垃圾产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好等到下一次GC去处理；
- 对处理器资源非常敏感。在并发阶段，收集器占用了一部分线程资源，导致应用程序变慢，降低总吞吐量。

CMS垃圾回收特点：

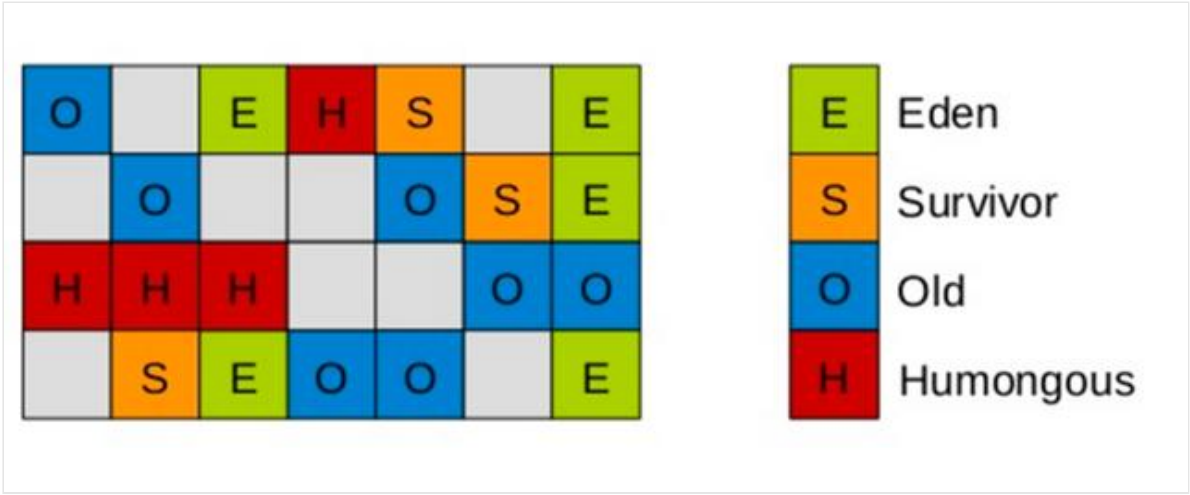
1. cms只会回收老年代和永久代（1.8开始为元数据区，需要设置CMSClassUnloadingEnabled），不会收集年轻代；
2. cms垃圾回收器开始执行回收操作，有一个触发阈值，默认是老年代或永久带达到92%，不能等到old内存用尽时回收，否则会导致并发回收失败。因为需要预留空间给用户线程运行。

G1收集器

G1垃圾收集器的目标是用在多核、大内存的机器上，在不同应用场景中追求高吞吐量和低停顿之间的最佳平衡。

在G1收集器出现之前的所有其他收集器，包括CMS在内，垃圾收集的目标范围要么是整个新生代（Minor GC），要么就是整个老年代（Major GC），再要么就是整个Java堆（Full GC）。而G1可以面向堆内存任何部分来组成回收集（Collection Set，一般简称CSet）进行回收，衡量标准不再是它属于哪个分代，而是哪块内存中存放的垃圾数量最多，回收收益最大，这就是G1收集器的Mixed GC模式。

G1将整个堆分成相同大小的分区（Region），有四种不同类型的分区：Eden、Survivor、Old和Humongous（大对象）。分区的大小取值范围为1M到32M，都是2的幂次方。Region大小可以通过`-XX:G1HeapRegionSize`参数指定。Humongous区域用于存储大对象。G1认为只要大小超过了一个Region容量一半的对象即可判定为大对象。



G1 收集器对各个Region回收所获得的空间大小和回收所需时间的经验值进行排序，得到一个优先级列表，每次根据用户设置的最大的回收停顿时间（使用参数-XX：MaxGCPauseMillis指定，默认值是200毫秒），优先处理回收价值最大的 Region。

Java堆分成多个独立Region，Region里面会存在跨Region引用对象，在垃圾回收寻找GC Roots需要扫描整个堆。G1采用了Rset（Remembered Set）来避免扫描整个堆。每个Region会有一个RSet，记录了哪些Region引用本Region中对象，即谁引用了我的对象，这样的话，在做可达性分析的时候就可以避免全堆扫描。

特点：可以由用户指定期望的垃圾收集停顿时间。

G1 收集器的回收过程分为以下几个步骤：

- 初始标记：。stw暂停所有的其他线程，记录直接与 gc root 直接相连的对象，速度很快 。
- 并发标记。从GC Root开始对堆中对象进行可达性分析，递归扫描整个堆里的对象图，找出要回收的对象，这阶段耗时较长，但可与用户程序并发执行。
- 最终标记。对用户线程做另一个短暂的暂停，用于处理并发阶段对象引用出现变动的区域。
- 筛选回收。对各个Region的回收价值和成本进行排序，根据用户所期望的停顿时间来制定回收计划，然后把决定回收的那一部分Region的存活对象复制到空的Region中，再清理掉整个旧的Region的全部空间。这里的操作涉及存活对象的移动，是必须暂停用户线程，由多条收集器线程并行完成的。

20. 常用的 JVM 调优的参数都有哪些?

- -Xms2g：初始化推大小为 2g；
- -Xmx2g：堆最大内存为 2g；
- -XX:NewRatio=4：设置年轻的和老年代的内存比例为 1:4；
- -XX:SurvivorRatio=8：设置新生代 Eden 和 Survivor 比例为 8:2；
- –XX:+UseParNewGC：指定使用 ParNew + Serial Old 垃圾回收器组合；
- -XX:+UseParallelOldGC：指定使用 ParNew + ParNew Old 垃圾回收器组合；
- -XX:+UseConcMarkSweepGC：指定使用 CMS + Serial Old 垃圾回收器组合；
- -XX:+PrintGC：开启打印 gc 信息；
- -XX:+PrintGCDetails：打印 gc 详细信息。

21. JVM调优工具有哪些?

jps：列出本机所有java进程的pid。

选项

- -q 仅输出VM标识符，不包括class name,jar name,arguments in main method
- -m 输出main method的参数
- -l 输出完全的包名，应用主类名，jar的完全路径名
- -v 输出jvm参数
- -V 输出通过flag文件传递到JVM中的参数(.hotspotrc文件或-XX:Flags=所指定的文件
- -Joption 传递参数到vm,例如:-J-Xms48m

```
jps -lvm
//output
//4124 com.zzx.Application -
javaagent:E:\IDEA2019\lib\idea_rt.jar=10291:E:\IDEA2019\bin -Dfile.encoding=UTF-
8
```

jstack：查看某个Java进程内的线程堆栈信息。-l，long listings，打印额外的锁信息，发生死锁时可以使用`jstack -l pid`观察锁持有情况。

```
jstack -l 4124 | more
```

output：

```
"http-nio-8001-exec-10" #40 daemon prio=5 os_prio=0 tid=0x000000002542f000
nid=0x4028 waiting on condition [0x000000002cc9e000]
   java.lang.Thread.State: WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        - parking to wait for  <0x000000077420d7e8> (a
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
        at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
        at
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(Abst
ractQueuedSynchronizer.java:2039)
        at
java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQueue.java:442)
        at org.apache.tomcat.util.threads.TaskQueue.take(TaskQueue.java:103)
        at org.apache.tomcat.util.threads.TaskQueue.take(TaskQueue.java:31)
        at
java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1074)
        at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1134)
        at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        at
org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:6
1)
        at java.lang.Thread.run(Thread.java:748)

   Locked ownable synchronizers:
        - None
```

jstat：虚拟机各种运行状态信息（类装载、内存、垃圾收集、jit编译等运行数据）。gcuitl 查看新生代、老年代及持久代GC的情况。

```
jstat -gcutil 4124
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
  0.00   0.00  67.21  19.20  96.36  94.96     10    0.084     3    0.191
 0.275
```

jmap：查看堆内存快照。查看进程中新生代、老年代、永久代的使用情况。

查询进程4124的堆内存快照：

```
>jmap -heap 4124
Attaching to process ID 4124, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.221-b11

using thread-local object allocation.
Parallel GC with 6 thread(s)

Heap Configuration:
   MinHeapFreeRatio         = 0
   MaxHeapFreeRatio         = 100
   MaxHeapSize              = 4238344192 (4042.0MB)
   NewSize                  = 88604672 (84.5MB)
   MaxNewSize               = 1412431872 (1347.0MB)
   OldSize                  = 177733632 (169.5MB)
   NewRatio                 = 2
   SurvivorRatio            = 8
   MetaspaceSize            = 21807104 (20.796875MB)
   CompressedClassSpaceSize = 1073741824 (1024.0MB)
   MaxMetaspaceSize         = 17592186044415 MB
   G1HeapRegionSize         = 0 (0.0MB)

Heap Usage:
PS Young Generation
Eden Space:
   capacity = 327155712 (312.0MB)
   used     = 223702392 (213.33922576904297MB)
   free     = 103453320 (98.66077423095703MB)
   68.37795697725736% used
From Space:
   capacity = 21495808 (20.5MB)
   used     = 0 (0.0MB)
   free     = 21495808 (20.5MB)
   0.0% used
To Space:
   capacity = 23068672 (22.0MB)
   used     = 0 (0.0MB)
   free     = 23068672 (22.0MB)
   0.0% used
PS Old Generation
   capacity = 217579520 (207.5MB)
   used     = 41781472 (39.845916748046875MB)
   free     = 175798048 (167.65408325195312MB)
   19.20285144484187% used

27776 interned Strings occupying 3262336 bytes.
```

查询进程pid = 41843 存活的对象占用内存前100排序：`jmap -histo:live 41843 | head -n 100`

22. 对象头了解吗?

Java对象保存在内存中时，由以下三部分组成：对象头、实例数据和对齐填充字节。

java的对象头由以下三部分组成：mark word、指向类信息的指针和数组长度（数组对象才有）。

mark word包含：对象的hashcode、分代年龄和锁标志位。

对象的实例数据就是在java代码中对象的属性和值。

对齐填充字节：因为JVM要求java的对象占的内存大小应该是8bit的倍数，所以后面有几个字节用于把对象的大小补齐至8bit的倍数。

内存对齐的主要作用是：

1. 平台原因：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。
2. 性能原因：经过内存对齐后，CPU的内存访问速度大大提升。

23. main方法执行过程

以下是示例代码：

```
public class App {
    public static void main(String[] args) {
        Student s = new Student("大彬");
        s.getName();
    }
}

class Student {
    public String name;

    public Student(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}
```

执行main方法的步骤如下:

1. 编译好 App.java 后得到 App.class 后，执行 App.class，系统会启动一个 JVM 进程，从 classpath 路径中找到一个名为 App.class 的二进制文件，将 App 的类信息加载到运行时数据区的方法区内，这个过程叫做 App 类的加载。
2. JVM 找到 App 的主程序入口，执行main方法。
3. 这个main中的第一条语句为 `Student student = new Student("大彬")`，就是让 JVM 创建一个 Student对象，但是这个时候方法区中是没有 Student 类的信息的，所以 JVM 马上加载 Student 类，把 Student 类的信息放到方法区中。
4. 加载完 Student 类后，JVM 在堆中为一个新的 Student 实例分配内存，然后调用构造函数初始化 Student 实例，这个 Student 实例持有 **指向方法区中的 Student 类的类型信息** 的引用。

5. 执行student.getName()时，JVM 根据 student 的引用找到 student 对象，然后根据 student 对象持有的引用定位到方法区中 student 类的类型信息的方法表，获得 getName() 的字节码地址。
6. 执行getName()方法。

24. 对象创建过程

1. **类加载检查**：虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。
2. **分配内存**：在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需的内存大小在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。
3. **初始化零值。**分配到的内存空间都初始化为零值（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。
4. **设置对象头。**Hotspot 虚拟机的对象头包括两部分信息，第一部分用于存储对象自身的运行时数据（哈希码、GC 分代年龄、锁状态标志等等），另一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是那个类的实例。
5. **执行init方法。**按照Java代码进行初始化。

25. 如何排查 OOM 的问题?

排查 OOM 的方法：

- 增加VM参数 `-XX:+HeapDumpOnOutOfMemoryError` 和 `-XX:HeapDumpPath=/tmp/heapdump.hprof`，当 OOM 发生时自动 dump 堆内存信息到指定目录；
- jstat 查看监控 JVM 的内存和 GC 情况，评估问题大概出在什么区域；
- 使用 MAT 工具载入 dump 文件，分析大对象的占用情况 。

26. GC是什么？为什么要GC?

GC 是垃圾收集的意思（Gabage Collection）。内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的。

27. 参考资料

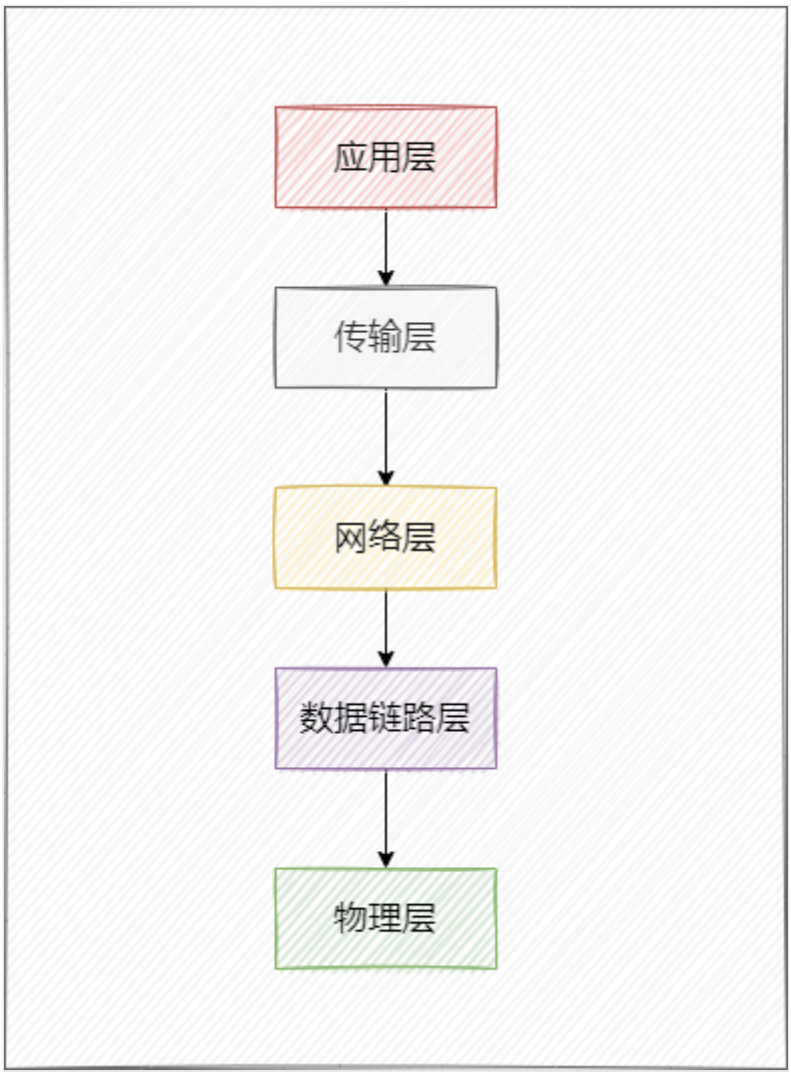
- 周志明. 深入理解 Java 虚拟机 [M]. 机械工业出版社
- https://blog.csdn.net/thinkwon/article/details/104390752

计算机网络

> 首先给大家分享一个github仓库，上面放了**200多本经典的计算机书籍**，包括C语言、C++、Java、Python、前端、数据库、操作系统、计算机网络、数据结构和算法、机器学习、编程人生等，可以star一下，下次找书直接在上面搜索，仓库持续更新中~
>
> github地址：https://github.com/Tyson0314/java-books
>
> 如果github访问不了，可以访问gitee仓库。
>
> gitee地址：https://gitee.com/tysondai/java-books

1. 网络分层结构

计算机网络体系大致分为三种，OSI七层模型、TCP/IP四层模型和五层模型。一般面试的时候考察比较多的是五层模型。



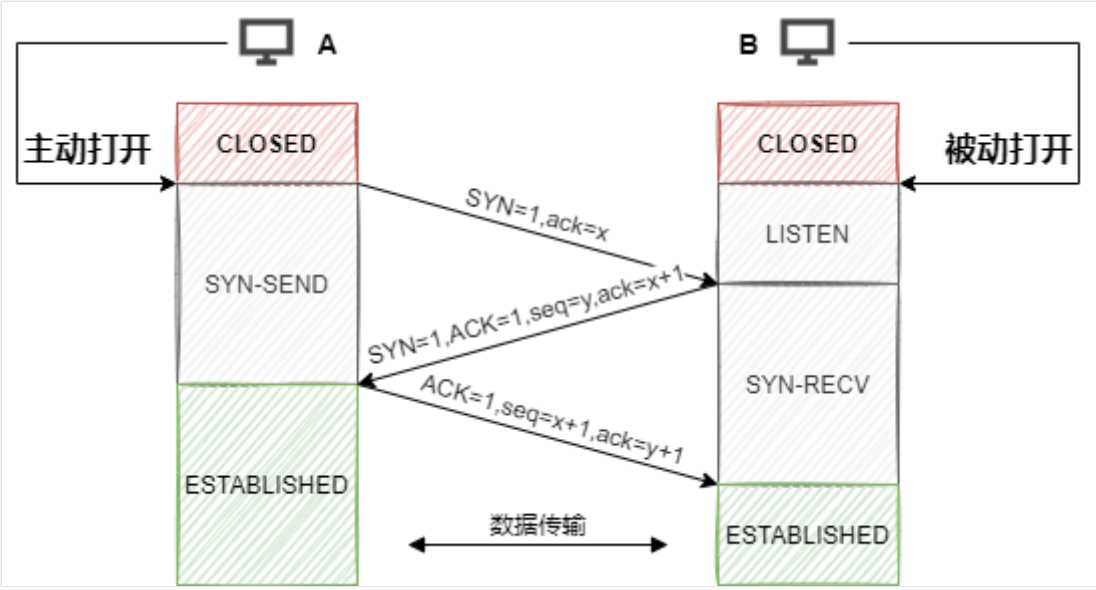
TCP/IP五层模型：应用层、传输层、网络层、数据链路层、物理层。

- **应用层**：为应用程序提供交互服务。在互联网中的应用层协议很多，如域名系统DNS、HTTP协议、SMTP协议等。
- **传输层**：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**：选择合适的路由和交换结点，确保数据及时传送。主要包括IP协议。
- **数据链路层**：在两个相邻节点之间传送数据时，**数据链路层将网络层交下来的 IP 数据报组装成帧**，在两个相邻节点间的链路上传送帧。

- **物理层**：实现相邻节点间比特流的透明传输，尽可能屏蔽传输介质和物理设备的差异。

2. 三次握手

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`CLOSED`。



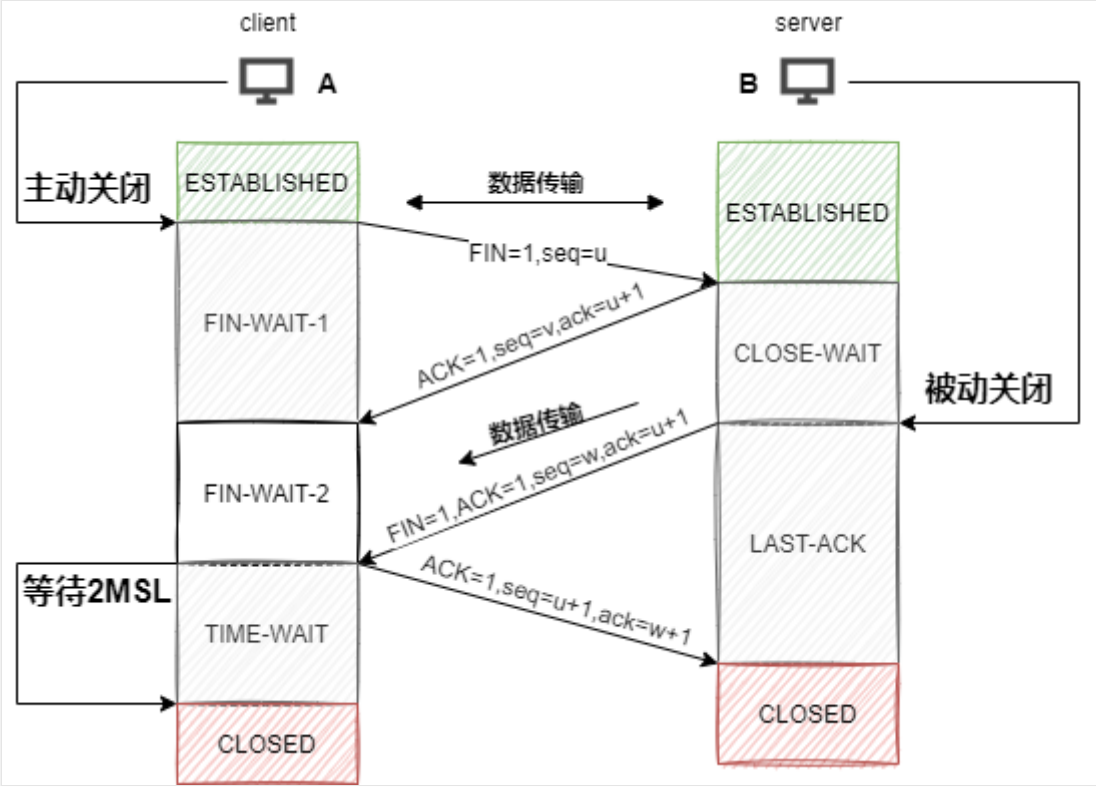
1. 第一次握手：客户端向服务端发起建立连接请求，客户端会随机生成一个起始序列号x，客户端向服务端发送的字段中包含标志位`SYN=1`，序列号`seq=x`。第一次握手前客户端的状态为`CLOSE`，第一次握手后客户端的状态为`SYN-SENT`。此时服务端的状态为`LISTEN`。
2. 第二次握手：服务端在收到客户端发来的报文后，会随机生成一个服务端的起始序列号y，然后给客户端回复一段报文，其中包括标志位`SYN=1`，`ACK=1`，序列号`seq=y`，确认号`ack=x+1`。第二次握手前服务端的状态为`LISTEN`，第二次握手后服务端的状态为`SYN-RCVD`，此时客户端的状态为`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）
3. 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，其中包含标志位`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`。**此时连接建立完成。**

3. 两次握手可以吗?

第三次握手主要为了**防止已失效的连接请求报文段**突然又传输到了服务端，导致产生问题。

- 比如客户端A发出连接请求，可能因为网络阻塞原因，A没有收到确认报文，于是A再重传一次连接请求。
- 连接成功，等待数据传输完毕后，就释放了连接。
- 然后A发出的第一个连接请求等到连接释放以后的某个时间才到达服务端B，此时B误认为A又发出一次新的连接请求，于是就向A发出确认报文段。
- 如果不采用三次握手，只要B发出确认，就建立新的连接了，**此时A不会响应B的确认且不发送数据，则B一直等待A发送数据，浪费资源。**

4. 四次挥手



1. A的应用进程先向其TCP发出连接释放报文段（`FIN=1，seq=u`），并停止再发送数据，主动关闭TCP连接，进入`FIN-WAIT-1`（终止等待1）状态，等待B的确认。
2. B收到连接释放报文段后即发出确认报文段（`ACK=1，ack=u+1，seq=v`），B进入`CLOSE-WAIT`（关闭等待）状态，此时的TCP处于半关闭状态，A到B的连接释放。
3. A收到B的确认后，进入`FIN-WAIT-2`（终止等待2）状态，等待B发出的连接释放报文段。
4. B发送完数据，就会发出连接释放报文段（`FIN=1，ACK=1，seq=w，ack=u+1`），B进入`LAST-ACK`（最后确认）状态，等待A的确认。
5. A收到B的连接释放报文段后，对此发出确认报文段（`ACK=1，seq=u+1，ack=w+1`），A进入`TIME-WAIT`（时间等待）状态。此时TCP未释放掉，需要经过时间等待计时器设置的时间`2MSL`（最大报文段生存时间）后，A才进入`CLOSED`状态。B收到A发出的确认报文段后关闭连接，若没收到A发出的确认报文段，B就会重传连接释放报文段。

5. 第四次挥手为什么要等待2MSL?

- **保证A发送的最后一个ACK报文段能够到达B。**这个`ACK`报文段有可能丢失，B收不到这个确认报文，就会超时重传连接释放报文段，然后A可以在`2MSL`时间内收到这个重传的连接释放报文段，接着A重传一次确认，重新启动2MSL计时器，最后A和B都进入到`CLOSED`状态，若A在`TIME-WAIT`状态不等待一段时间，而是发送完ACK报文段后立即释放连接，则无法收到B重传的连接释放报文段，所以不会再发送一次确认报文段，B就无法正常进入到`CLOSED`状态。
- **防止已失效的连接请求报文段出现在本连接中。**A在发送完最后一个`ACK`报文段后，再经过2MSL，就可以使这个连接所产生的所有报文段都从网络中消失，使下一个新的连接中不会出现旧的连接请求报文段。

6. 为什么是四次挥手?

因为当Server端收到Client端的`SYN`连接请求报文后，可以直接发送`SYN+ACK`报文。**但是在关闭连接时，当Server端收到Client端发出的连接释放报文时，很可能并不会立即关闭SOCKET**，所以Server端先回复一个`ACK`报文，告诉Client端我收到你的连接释放报文了。只有等到Server端所有的报文都发送完了，这时Server端才能发送连接释放报文，之后两边才会真正的断开连接。故需要四次挥手。

7. TCP有哪些特点?

- TCP是**面向连接**的运输层协议。
- **点对点**，每一条TCP连接只能有两个端点。
- TCP提供**可靠交付**的服务。
- TCP提供**全双工通信**。
- **面向字节流**。

8. TCP和UDP的区别?

1. TCP**面向连接**；UDP是无连接的，即发送数据之前不需要建立连接。
2. TCP提供**可靠的服务**；UDP不保证可靠交付。
3. TCP**面向字节流**，把数据看成一连串无结构的字节流；UDP是面向报文的。
4. TCP有**拥塞控制**；UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如实时视频会议等）。
5. 每一条TCP连接只能是**点到点**的；UDP支持一对一、一对多、多对一和多对多的通信方式。
6. TCP首部开销20字节；UDP的首部开销小，只有8个字节。

9. HTTP协议的特点?

1. HTTP允许传输**任意类型**的数据。传输的类型由Content-Type加以标记。
2. **无状态**。对于客户端每次发送的请求，服务器都认为是一个新的请求，上一次会话和下一次会话之间没有联系。
3. 支持**客户端/服务器模式**。

10. HTTP报文格式

HTTP请求由**请求行、请求头部、空行和请求体**四个部分组成。

- **请求行**：包括请求方法，访问的资源URL，使用的HTTP版本。`GET`和`POST`是最常见的HTTP方法，除此以外还包括`DELETE、HEAD、OPTIONS、PUT、TRACE`。
- **请求头**：格式为"属性名:属性值"，服务端根据请求头获取客户端的信息，主要有`cookie、host、connection、accept-language、accept-encoding、user-agent`。
- **请求体**：用户的请求数据如用户名，密码等。

请求报文示例：

```
POST /xxx HTTP/1.1 请求行
Accept:image/gif.image/jpeg, 请求头部
Accept-Language:zh-cn
Connection:Keep-Alive
Host:localhost
User-Agent:Mozila/4.0(compatible;MSIE5.01;Window NT5.0)
Accept-Encoding:gzip,deflate

username=dabin 请求体
```

HTTP响应也由四个部分组成，分别是：**状态行、响应头、空行和响应体**。

- **状态行**：协议版本，状态码及状态描述。
- **响应头**：响应头字段主要有`connection、content-type、content-encoding、content-length、set-cookie、Last-Modified，、Cache-Control、Expires`。
- **响应体**：服务器返回给客户端的内容。

响应报文示例：

```
HTTP/1.1 200 OK
Server:Apache Tomcat/5.0.12
Date:Mon,6Oct2003 13:23:42 GMT
Content-Length:112

<html>
    <body>响应体</body>
</html>
```

11. HTTP状态码有哪些?

1xx	服务器收到请求，需要请求者继续执行操作
2xx	请求正常处理完毕
3xx	重定向，需要进一步操作已完成请求
4xx	客户端错误，服务器无法处理请求
5xx	服务器处理请求出错

12. POST和GET的区别?

- GET请求参数通过URL传递，POST的参数放在请求体中。
- GET产生一个TCP数据包；POST产生两个TCP数据包。对于GET方式的请求，浏览器会把请求头和请求体一并发送出去；而对于POST，浏览器先发送请求头，服务器响应100 continue，浏览器再发送请求体。
- GET请求会被浏览器主动缓存，而POST不会，除非手动设置。
- GET请求只能进行url编码，而POST支持多种编码方式。
- GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。

13. HTTP长连接和短连接?

HTTP1.0默认使用的是短连接。浏览器和服务器每进行一次HTTP操作，就建立一次连接，任务结束就中断连接。

HTTP/1.1起，默认使用长连接。要使用长连接，客户端和服务器的HTTP首部的Connection都要设置为keep-alive，才能支持长连接。

HTTP长连接，指的是**复用TCP连接。**多个HTTP请求可以复用同一个TCP连接，这就节省了TCP连接建立和断开的消耗。

1. # 14. HTTP1.1和 HTTP2.0的区别?

   HTTP2.0相比HTTP1.1支持的特性：

   - **新的二进制格式**：HTTP1.1 基于文本格式传输数据；HTTP2.0采用二进制格式传输数据，解析更高效。
   - **多路复用**：在一个连接里，允许同时发送多个请求或响应，**并且这些请求或响应能够并行的传输而不被阻塞**，避免 HTTP1.1 出现的”队头堵塞”问题。

- **头部压缩**，HTTP1.1的header带有大量信息，而且每次都要重复发送；HTTP2.0 把header从数据中分离，并封装成头帧和数据帧，**使用特定算法压缩头帧**，有效减少头信息大小。并且HTTP2.0**在客户端和服务器端记录了之前发送的键值对，对于相同的数据，不会重复发送。**比如请求a发送了所有的头信息字段，请求b则**只需要发送差异数据**，这样可以减少冗余数据，降低开销。
- **服务端推送**：HTTP2.0允许服务器向客户端推送资源，无需客户端发送请求到服务器获取。

15. HTTPS与HTTP的区别?

1. HTTP是超文本传输协议，信息是**明文传输**；HTTPS则是具有**安全性**的ssl加密传输协议。
2. HTTP和HTTPS用的端口不一样，HTTP端口是80，HTTPS是443。
3. HTTPS协议**需要到CA机构申请证书**，一般需要一定的费用。
4. HTTP运行在TCP协议之上；HTTPS运行在SSL协议之上，SSL运行在TCP协议之上。

16. 什么是数字证书?

服务端可以向证书颁发机构CA申请证书，以避免中间人攻击（防止证书被篡改）。证书包含三部分内容：**证书内容、证书签名算法和签名**，签名是为了验证身份。

服务端把证书传输给浏览器，浏览器从证书里取公钥。证书可以证明该公钥对应本网站。

数字签名的制作过程：

1. CA使用证书签名算法对证书内容进行**hash运算。**
2. 对hash后的值**用CA的私钥加密**，得到数字签名。

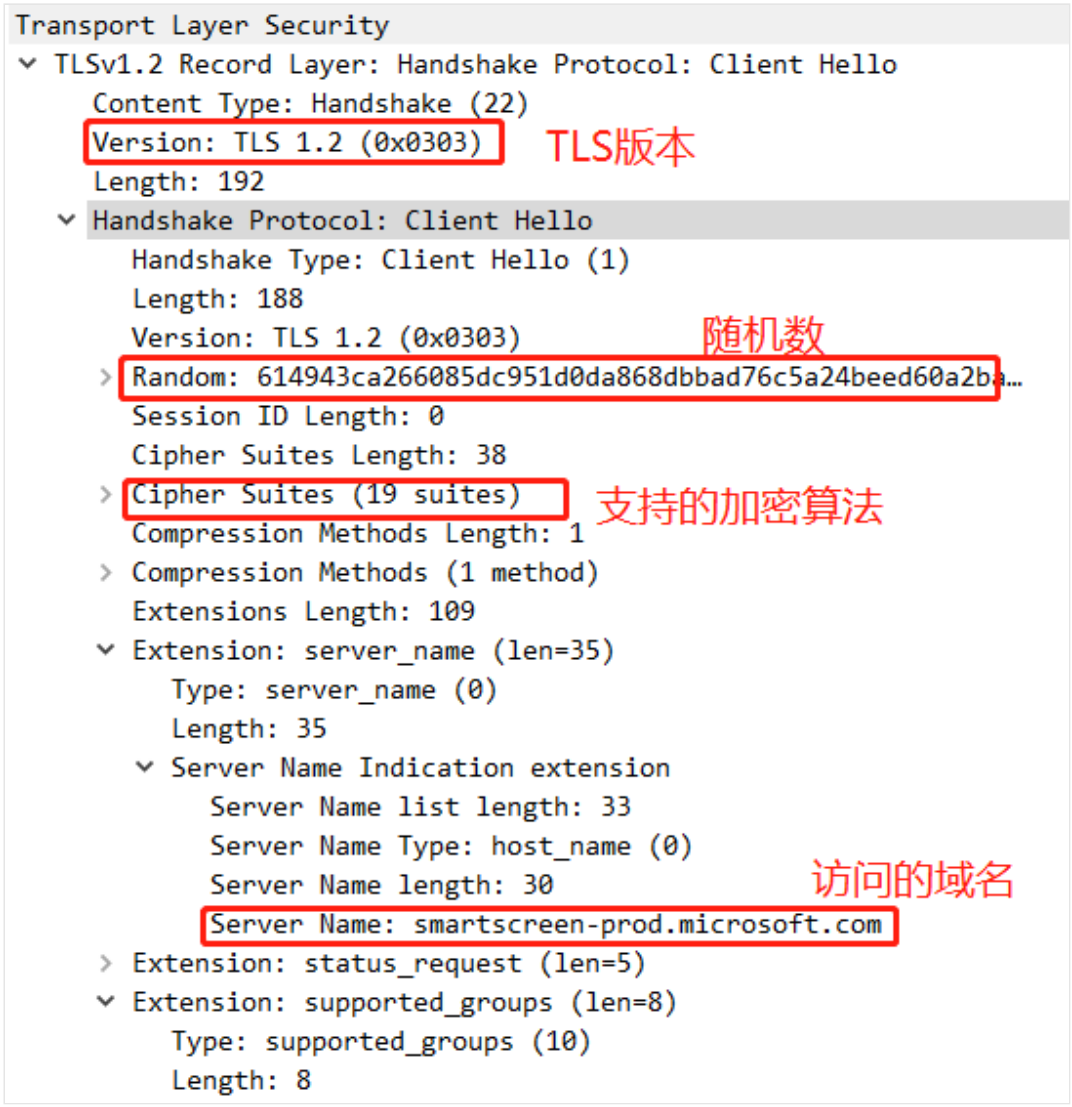
浏览器验证过程：

1. 获取证书，得到证书内容、证书签名算法和数字签名。
2. 用CA机构的公钥**对数字签名解密**（由于是浏览器信任的机构，所以浏览器会保存它的公钥）。
3. 用证书里的签名算法**对证书内容进行hash运算。**
4. 比较解密后的数字签名和对证书内容做hash运算后得到的哈希值，相等则表明证书可信。

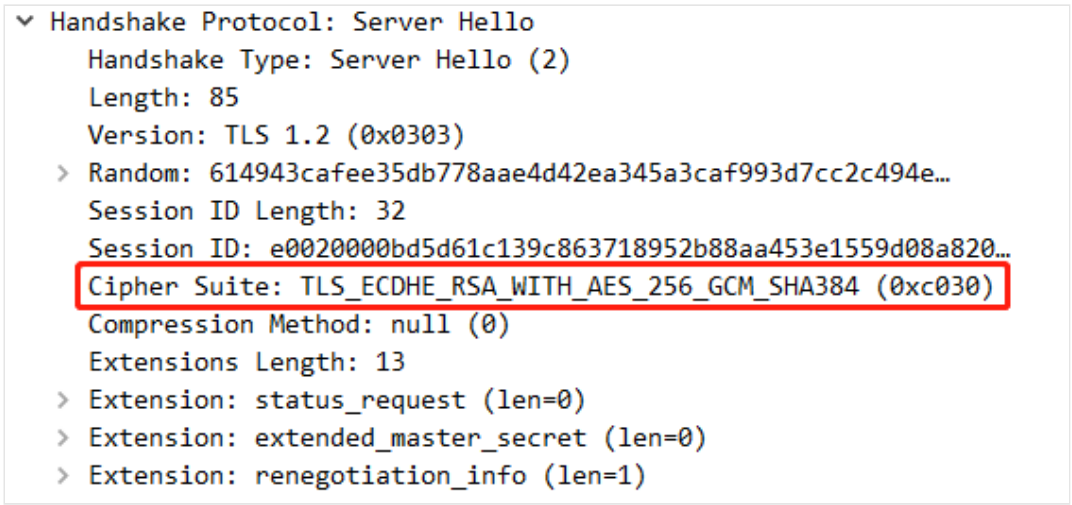
17. HTTPS原理

首先是TCP三次握手，然后客户端发起一个HTTPS连接建立请求，客户端先发一个`Client Hello`的包，然后服务端响应`Server Hello`，接着再给客户端发送它的证书，然后双方经过密钥交换，最后使用交换的密钥加解密数据。

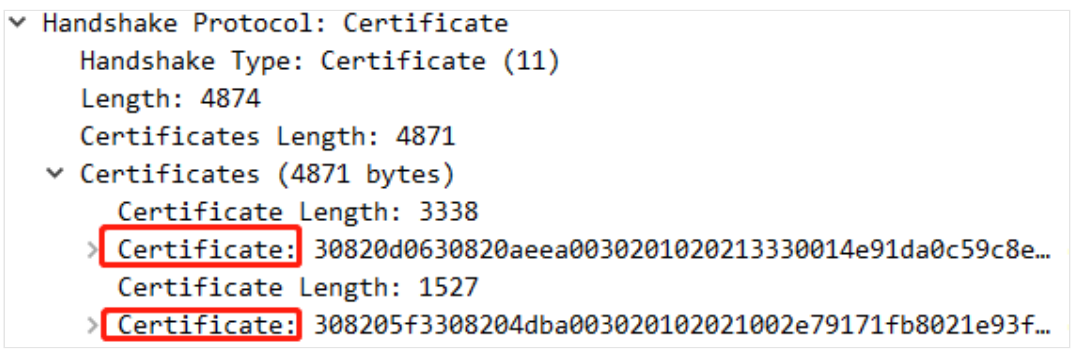
1. **协商加密算法 。**在`Client Hello`里面客户端会告知服务端自己当前的一些信息，包括客户端要使用的TLS版本，支持的加密算法，要访问的域名，给服务端生成的一个随机数（Nonce）等。需要提前告知服务器想要访问的域名以便服务器发送相应的域名的证书过来。



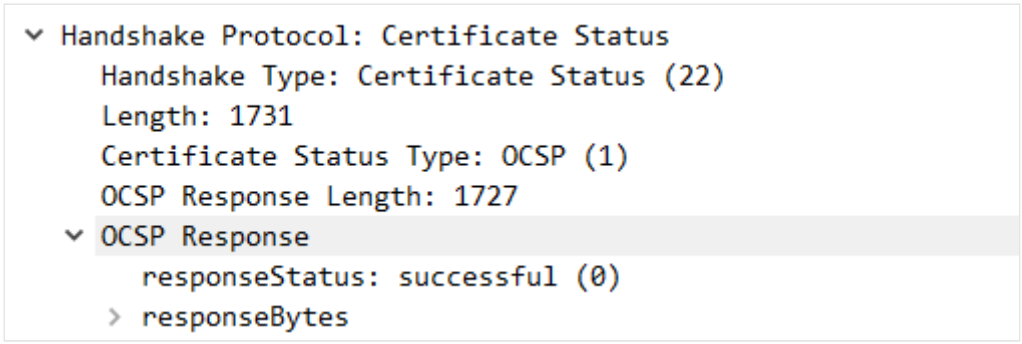
2. 服务端响应`Server Hello`，告诉客户端服务端**选中的加密算法。**



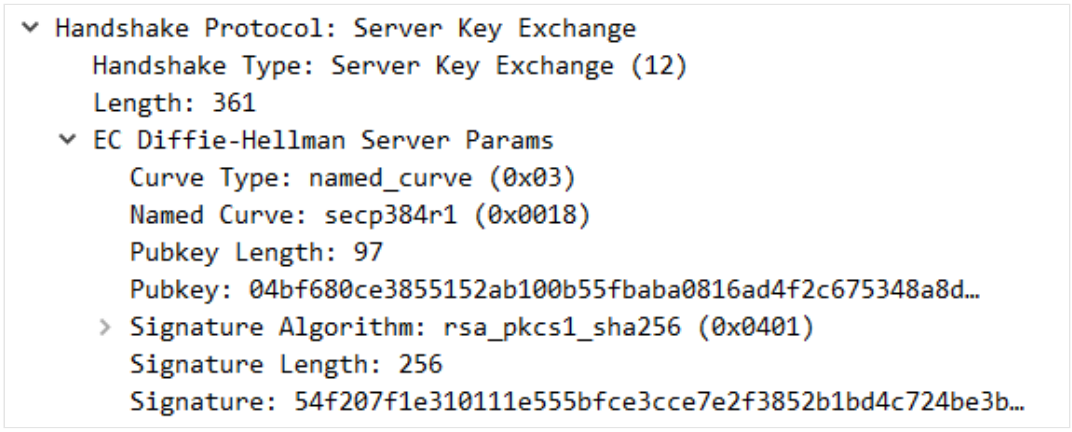
3. 接着服务端给客户端发来了2个证书。第二个证书是第一个证书的签发机构（CA）的证书。

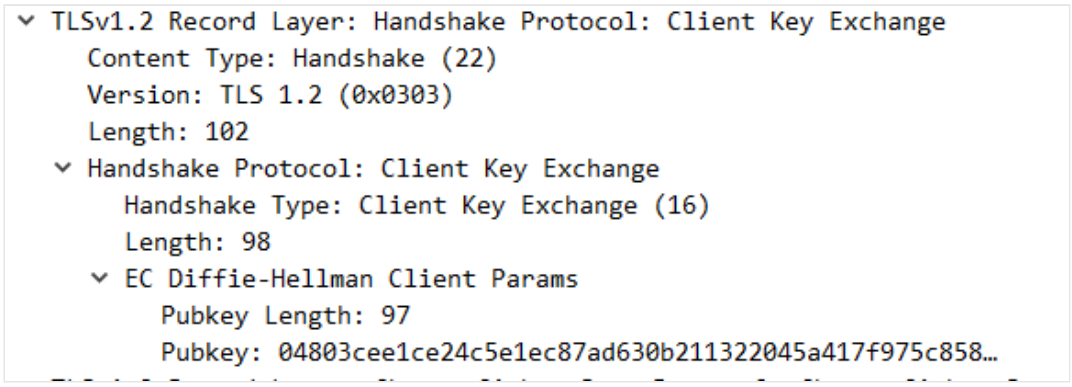


4. 客户端使用证书的认证机构CA公开发布的RSA公钥**对该证书进行验证**，下图表明证书认证成功。

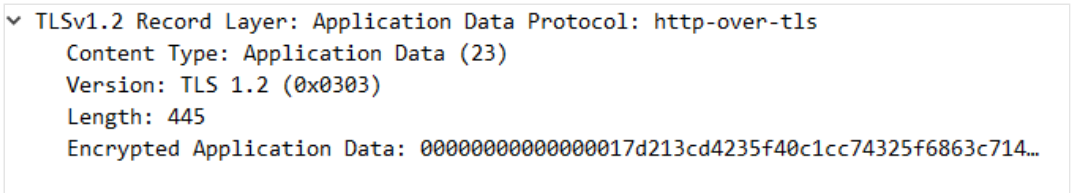


5. 验证通过之后，浏览器和服务器通过**密钥交换算法**产生共享的**对称密钥。**





6. 开始传输数据，使用同一个对称密钥来加解密。

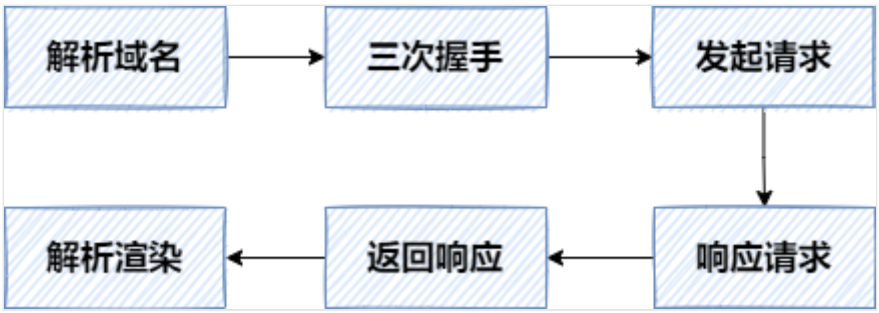


18. DNS 的解析过程?

1. 浏览器搜索**自己的DNS缓存**
2. 若没有，则搜索**操作系统中的DNS缓存和hosts文件**
3. 若没有，则操作系统将域名发送至**本地域名服务器**，本地域名服务器查询自己的DNS缓存，查找成功则返回结果，否则依次向**根域名服务器、顶级域名服务器、权限域名服务器**发起查询请求，最终返回IP地址给本地域名服务器
4. 本地域名服务器将得到的IP地址返回给**操作系统**，同时自己也**将IP地址缓存起来**
5. 操作系统将 IP 地址返回给浏览器，同时自己也将IP地址缓存起来
6. 浏览器得到域名对应的IP地址

19. 浏览器中输入URL返回页面过程?

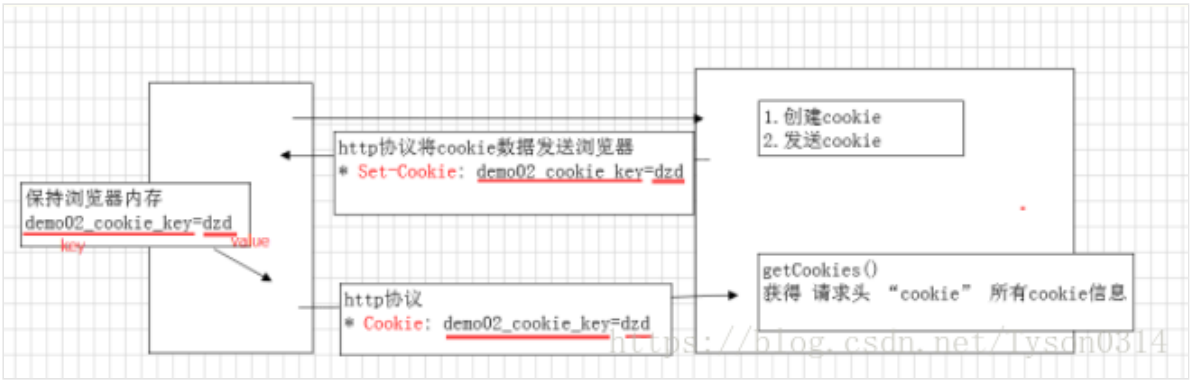
1. **解析域名**，找到主机 IP。
2. 浏览器利用 IP 直接与网站主机通信，**三次握手**，建立 TCP 连接。浏览器会以一个随机端口向服务端的 web 程序 80 端口发起 TCP 的连接。
3. 建立 TCP 连接后，浏览器向主机发起一个HTTP请求。
4. 服务器**响应请求**，返回响应数据。
5. 浏览器**解析响应内容，进行渲染**，呈现给用户。



20. 什么是cookie和session?

由于HTTP协议是无状态的协议，需要用某种机制来识具体的用户身份，用来跟踪用户的整个会话。常用的会话跟踪技术是cookie与session。

cookie就是由服务器发给客户端的特殊信息，而这些信息以文本文件的方式存放在客户端，然后客户端每次向服务器发送请求的时候都会带上这些特殊的信息。说得更具体一些：当用户使用浏览器访问一个支持cookie的网站的时候，用户会提供包括用户名在内的个人信息并且提交至服务器；接着，服务器在向客户端回传相应的超文本的同时也会发回这些个人信息，当然这些信息并不是存放在HTTP响应体中的，而是存放于HTTP响应头；当客户端浏览器接收到来自服务器的响应之后，浏览器会将这些信息存放在一个统一的位置。 自此，客户端再向服务器发送请求的时候，都会把相应的cookie存放在HTTP请求头再次发回至服务器。服务器在接收到来自客户端浏览器的请求之后，就能够通过分析存放于请求头的cookie得到客户端特有的信息，从而动态生成与该客户端相对应的内容。网站的登录界面中"请记住我"这样的选项，就是通过cookie实现的。

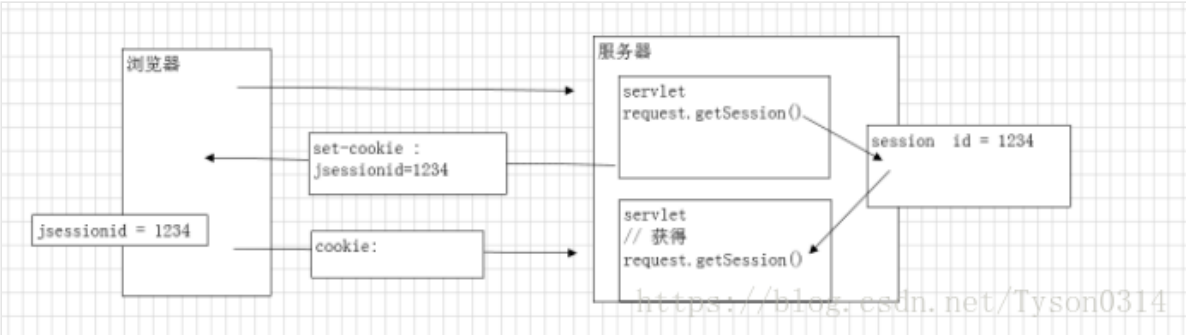


cookie工作流程：

1. servlet创建cookie，保存少量数据，发送给浏览器。
2. 浏览器获得服务器发送的cookie数据，将自动的保存到浏览器端。
3. 下次访问时，浏览器将自动携带cookie数据发送给服务器。

session原理：首先浏览器请求服务器访问web站点时，服务器首先会检查这个客户端请求是否已经包含了一个session标识、称为SESSIONID，如果已经包含了一个sessionid则说明以前已经为此客户端创建过session，服务器就按照sessionid把这个session检索出来使用，如果客户端请求不包含session id，则服务器为此客户端创建一个session，并且生成一个与此session相关联的独一无二的sessionid存放到cookie中，这个sessionid将在本次响应中返回到客户端保存，这样在交互的过程中，浏览器端每次请求时，都会带着这个sessionid，服务器根据这个sessionid就可以找得到对应的session。以此来达到共享

数据的目的。 这里需要注意的是，session不会随着浏览器的关闭而死亡，而是等待超时时间。



21. Cookie和Session的区别?

- **作用范围不同**，Cookie 保存在客户端，Session 保存在服务器端。
- **有效期不同**，Cookie 可设置为长时间保持，比如我们经常使用的默认登录功能，Session 一般失效时间较短，客户端关闭或者 Session 超时都会失效。
- **隐私策略不同**，Cookie 存储在客户端，容易被窃取；Session 存储在服务端，安全性相对 Cookie 要好一些。
- **存储大小不同**， 单个 Cookie 保存的数据不能超过 4K；对于 Session 来说存储没有上限，但出于对服务器的性能考虑，Session 内不要存放过多的数据，并且需要设置 Session 删除机制。

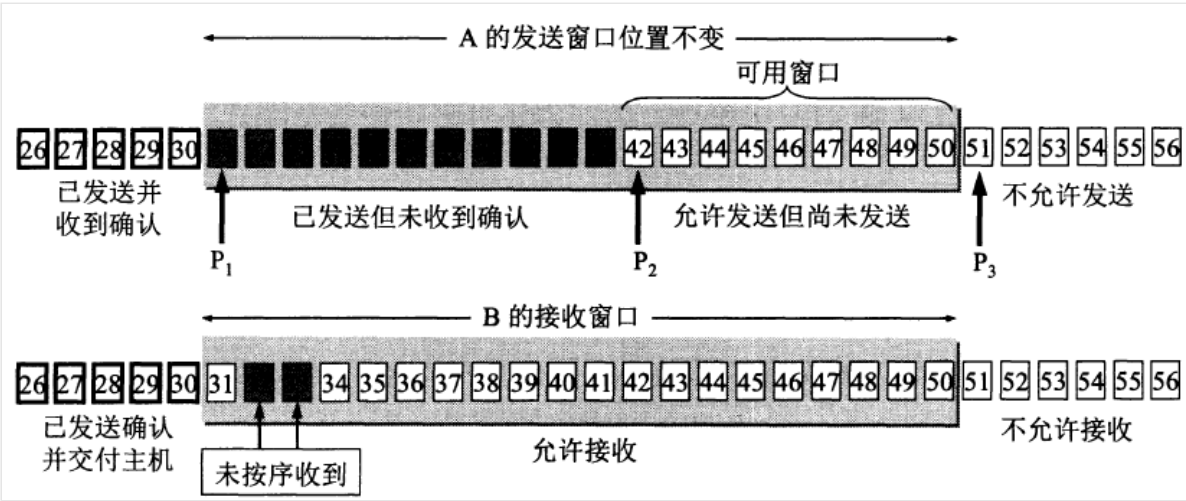
22. 什么是对称加密和非对称加密?

对称加密：通信双方使用**相同的密钥**进行加密。特点是加密速度快，但是缺点是密钥泄露会导致密文数据被破解。常见的对称加密有 `AES` 和 `DES` 算法。

非对称加密：它需要生成两个密钥，**公钥和私钥**。公钥是公开的，任何人都可以获得，而私钥是私人保管的。公钥负责加密，私钥负责解密；或者私钥负责加密，公钥负责解密。这种加密算法**安全性更高**，但是**计算量相比对称加密大很多**，加密和解密都很慢。常见的非对称算法有 `RSA` 和 `DSA` 。

23. 滑动窗口机制

TCP 利用滑动窗口实现流量控制。流量控制是为了控制发送方发送速率，保证接收方来得及接收。 TCP 会话的双方都各自维护一个发送窗口和一个接收窗口。接收窗口大小取决于应用、系统、硬件的限制。发送窗口则取决于对端通告的接收窗口。接收方发送的确认报文中的window字段可以用来控制发送方窗口大小，从而影响发送方的发送速率。将接收方的确认报文window字段设置为 0，则发送方不能发送数据。



TCP头包含window字段，16bit位，它代表的是窗口的字节容量，最大为65535。这个字段是接收端告诉发送端自己还有多少缓冲区可以接收数据。于是发送端就可以根据这个接收端的处理能力来发送数据，而不会导致接收端处理不过来。接收窗口的大小是约等于发送窗口的大小。

24. 详细讲一下拥塞控制?

防止过多的数据注入到网络中。 几种拥塞控制方法：慢开始(slow-start)、拥塞避免(congestion avoidance)、快重传(fast retransmit)和快恢复(fast recovery)。

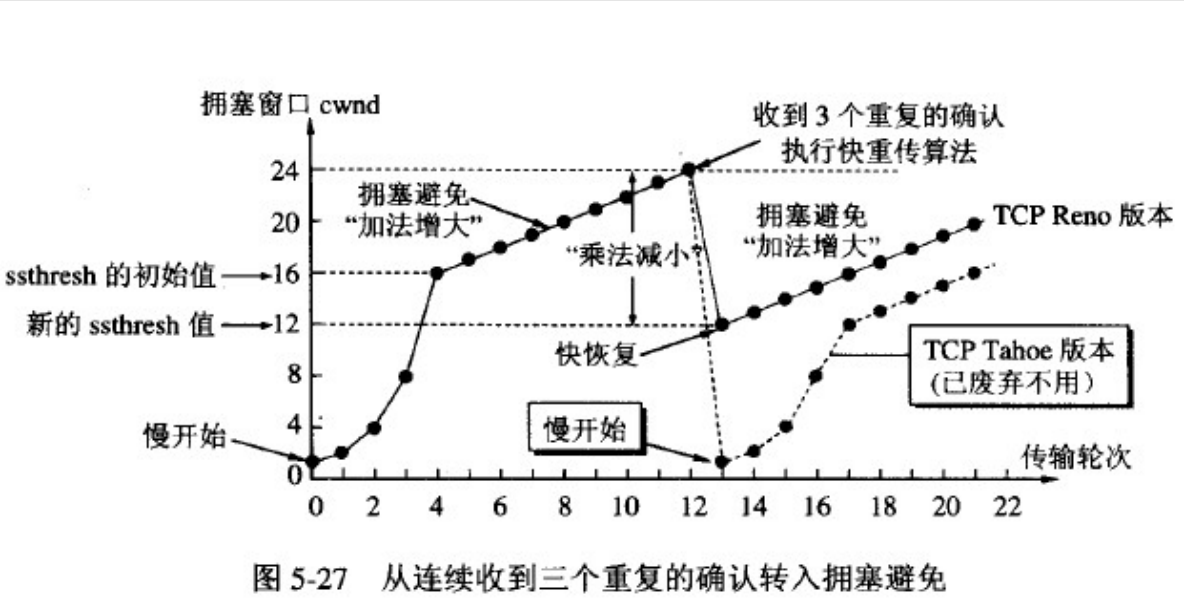


图 5-27　从连续收到三个重复的确认转入拥塞避免

24.1. 慢开始

把拥塞窗口 cwnd 设置为一个最大报文段MSS的数值。而在每收到一个对新的报文段的确认后，把拥塞窗口增加至多一个MSS的数值。每经过一个传输轮次，拥塞窗口 cwnd 就加倍。 为了防止拥塞窗口cwnd增长过大引起网络拥塞，还需要设置一个慢开始门限ssthresh状态变量。

当 cwnd < ssthresh 时，使用慢开始算法。

当 cwnd > ssthresh 时，停止使用慢开始算法而改用拥塞避免算法。

当 cwnd = ssthresh 时，既可使用慢开始算法，也可使用拥塞控制避免算法。

24.2. 拥塞避免

让拥塞窗口cwnd缓慢地增大，每经过一个往返时间RTT就把发送方的拥塞窗口cwnd加1，而不是加倍。这样拥塞窗口cwnd按线性规律缓慢增长。

无论在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞（其根据就是没有收到确认），就要把慢开始门限ssthresh设置为出现拥塞时的发送 方窗口值的一半（但不能小于2）。然后把拥塞窗口cwnd重新设置为1，执行慢开始算法。这样做的目的就是要迅速减少主机发送到网络中的分组数，使得发生 拥塞的路由器有足够时间把队列中积压的分组处理完毕。

24.3. 快重传

有时个别报文段会在网络中丢失，但实际上网络并未发生拥塞。如果发送方迟迟收不到确认，就会产生超时，就会误认为网络发生了拥塞。这就导致发送方错误地启动慢开始，把拥塞窗口cwnd又设置为1，因而降低了传输效率。

快重传算法可以避免这个问题。快重传算法首先要求接收方每收到一个失序的报文段后就立即发出重复确认，使发送方及早知道有报文段没有到达对方。

发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待重传计时器到期。由于发送方尽早重传未被确认的报文段，因此采用快重传后可以使整个网络吞吐量提高约20%。

24.4. 快恢复

当发送方连续收到三个重复确认，就会把慢开始门限ssthresh减半，接着把cwnd值设置为慢开始门限ssthresh减半后的数值，然后开始执行拥塞避免算法，使拥塞窗口缓慢地线性增大。

在采用快恢复算法时，慢开始算法只是在TCP连接建立时和网络出现超时时才使用。 采用这样的拥塞控制方法使得TCP的性能有明显的改进。

25. ARP协议

ARP解决了同一个局域网上的主机和路由器IP和MAC地址的解析。

- 每台主机都会在自己的ARP缓冲区中建立一个ARP列表，以表示IP地址和MAC地址的对应关系。
- 当源主机需要将一个数据包要发送到目的主机时，会首先检查自己 ARP列表中是否存在该 IP地址对应的MAC地址，如果有，就直接将数据包发送到这个MAC地址；如果没有，就向本地网段发起一个ARP请求的广播包，查询此目的主机对应的MAC地址。此ARP请求数据包里包括源主机的IP地址、硬件地址、以及目的主机的IP地址。
- 网络中所有的主机收到这个ARP请求后，会检查数据包中的目的IP是否和自己的IP地址一致。如果不相同就忽略此数据包；如果相同，该主机首先将发送端的MAC地址和IP地址添加到自己的ARP列表中，如果ARP表中已经存在该IP的信息，则将其覆盖，然后给源主机发送一个 ARP响应数据包，告诉对方自己是它需要查找的MAC地址。
- 源主机收到这个ARP响应数据包后，将得到的目的主机的IP地址和MAC地址添加到自己的ARP列表中，并利用此信息开始数据的传输。
- 如果源主机一直没有收到ARP响应数据包，表示ARP查询失败。

MySQL

1. 事务的四大特性?

事务特性ACID：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。

- 原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚。
- 一致性是指一个事务执行之前和执行之后都必须处于一致性状态。比如a与b账户共有1000块，两人之间转账之后无论成功还是失败，它们的账户总和还是1000。
- 隔离性。跟隔离级别相关，如read committed，一个事务只能读到已经提交的修改。
- 持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

2. 数据库的三大范式

第一范式1NF

确保数据库表字段的原子性。

比如字段 `userInfo`：`山东省 1318162008'` ，依照第一范式必须拆分成 `userInfo`：`山东省` `userTel`：`1318162008` 两个字段。

第二范式2NF

首先要满足第一范式，另外包含两部分内容，一是表必须有一个主键；二是非主键列必须完全依赖于主键，而不能只依赖于主键的一部分。

举个例子。假定选课关系表为StudentCourse(学号, 姓名, 年龄, 课程名称, 成绩, 学分)，主键为(学号, 课程名称)。其中学分完全依赖于课程名称，姓名年龄完全依赖学号，不符合第二范式，会导致数据冗余（学生选n门课，姓名年龄有n条记录）、插入异常（插入一门新课，因为没有学号，无法保存新课记录）等问题。

可以拆分成三个表：学生：Student(学号, 姓名, 年龄)；课程：Course(课程名称, 学分)；选课关系：StudentCourseRelation(学号, 课程名称, 成绩)。

第三范式3NF

首先要满足第二范式，另外非主键列必须直接依赖于主键，不能存在传递依赖。即不能存在：非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况。

假定学生关系表为Student(学号, 姓名, 年龄, 学院id, 学院地点, 学院电话)，主键为"学号"，其中学院id依赖于学号，而学院地点和学院电话依赖于学院id，存在传递依赖，不符合第三范式。

可以把学生关系表分为如下两个表：学生：(学号, 姓名, 年龄, 学院id)；学院：(学院,id 地点, 电话)。

2NF和3NF的区别?

- 2NF依据是非主键列是否完全依赖于主键，还是依赖于主键的一部分。
- 3NF依据是非主键列是直接依赖于主键，还是直接依赖于非主键。

3. 事务隔离级别有哪些?

先了解下几个概念：脏读、不可重复读、幻读。

- 脏读是指在一个事务处理过程里读取了另一个未提交的事务中的数据。
- 不可重复读是指在对于数据库中的某行记录，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，另一个事务修改了数据并提交了。

- 幻读是当某个事务在读取某个范围内的记录时，另外一个事务又在该范围内插入了新的记录，当之前的事务再次读取该范围的记录时，会产生幻行，就像产生幻觉一样，这就是发生了幻读。

不可重复读和脏读的区别是，脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

幻读和不可重复读都是读取了另一条已经提交的事务，不同的是不可重复读的重点是修改，幻读的重点在于新增或者删除。

事务隔离就是为了解决上面提到的脏读、不可重复读、幻读这几个问题。

MySQL数据库为我们提供的四种隔离级别：

- Serializable (串行化)：通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。
- Repeatable read (可重复读)：MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行，解决了不可重复读的问题。
- Read committed (读已提交)：一个事务只能看见已经提交事务所做的改变。可避免脏读的发生。
- Read uncommitted (读未提交)：所有事务都可以看到其他未提交事务的执行结果。

查看隔离级别：

```
select @@transaction_isolation;
```

设置隔离级别：

```
set session transaction isolation level read uncommitted;
```

4. 索引

4.1. 什么是索引?

索引是存储引擎用于提高数据库表的访问速度的一种数据结构。

4.2. 索引的优缺点?

优点：

- 加快数据查找的速度
- 为用来排序或者是分组的字段添加索引，可以加快分组和排序的速度
- 加速表与表之间的连接

缺点：

- 建立索引需要占用物理空间
- 会降低表的增删改的效率，因为每次对表记录进行增删改，需要进行动态维护索引，导致增删改时间变长

4.3. 索引的作用?

数据是存储在磁盘上的，查询数据时，如果没有索引，会加载所有的数据到内存，依次进行检索，读取磁盘次数较多。有了索引，就不需要加载所有数据，因为B+树的高度一般在2-4层，最多只需要读取2-4次磁盘，查询速度大大提升。

4.4. 什么情况下需要建索引?

1. 经常用于查询的字段
2. 经常用于连接的字段（如外键）建立索引，可以加快连接的速度
3. 经常需要排序的字段建立索引，因为索引已经排好序，可以加快排序查询速度

4.5. 什么情况下不建索引?

1. where条件中用不到的字段不适合建立索引
2. 表记录较少
3. 需要经常增删改
4. 参与列计算的列不适合建索引
5. 区分度不高的字段不适合建立索引，性别等

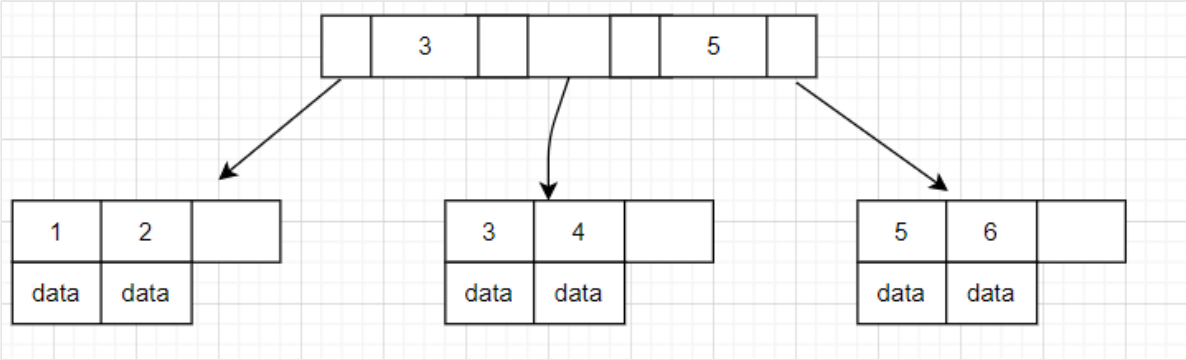
4.6. 索引的数据结构

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

B+ 树是基于B 树和叶子节点顺序访问指针进行实现，它具有B树的平衡性，并且通过顺序访问指针来提高区间查询的性能。

在 B+ 树中，节点中的 key 从左到右递增排列，如果某个指针的左右相邻 key 分别是 key_i 和 key_{i+1}，则该指针指向节点的所有 key 大于等于 key_i 且小于等于 key_{i+1}。



进行查找操作时，首先在根节点进行二分查找，找到key所在的指针，然后递归地在指针所指向的节点进行查找。直到查找到叶子节点，然后在叶子节点上进行二分查找，找出 key 所对应的数据项。

MySQL 数据库使用最多的索引类型是BTREE索引，底层基于B+树数据结构来实现。

```
mysql> show index from blog\G;
*************************** 1. row ***************************
        Table: blog
   Non_unique: 0
     Key_name: PRIMARY
 Seq_in_index: 1
  Column_name: blog_id
    Collation: A
  Cardinality: 4
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
```

```
Index_comment:
      Visible: YES
   Expression: NULL
```

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。

4.7. Hash索引和B+树索引的区别?

- 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。

4.8. 为什么B+树比B树更适合实现数据库索引?

- 由于B+树的数据都存储在叶子结点中，叶子结点均为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，而在数据库中基于范围的查询是非常频繁的，所以通常B+树用于数据库索引。
- B+树的节点只存储索引key值，具体信息的地址存在于叶子节点的地址中。这就使以页为单位的索引中可以存放更多的节点。减少更多的I/O支出。
- B+树的查询效率更加稳定，任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

4.9. 索引有什么分类?

1. 主键索引：名为primary的唯一非空索引，不允许有空值。
2. 唯一索引：索引列中的值必须是唯一的，但是允许为空值。唯一索引和主键索引的区别是：UNIQUE 约束的列可以为null且可以存在多个null值。UNIQUE KEY的用途：唯一标识数据库表中的每条记录，主要是用来防止数据重复插入。创建唯一索引的SQL语句如下：

   ```
   ALTER TABLE table_name
   ADD CONSTRAINT constraint_name UNIQUE KEY(column_1,column_2,...);
   ```

3. 组合索引：在表中的多个字段组合上创建的索引，只有在查询条件中使用了这些字段的左边字段时，索引才会被使用，使用组合索引时遵循最左前缀原则。
4. 全文索引：只有在MyISAM引擎上才能使用，只能在CHAR,VARCHAR,TEXT类型字段上使用全文索引。

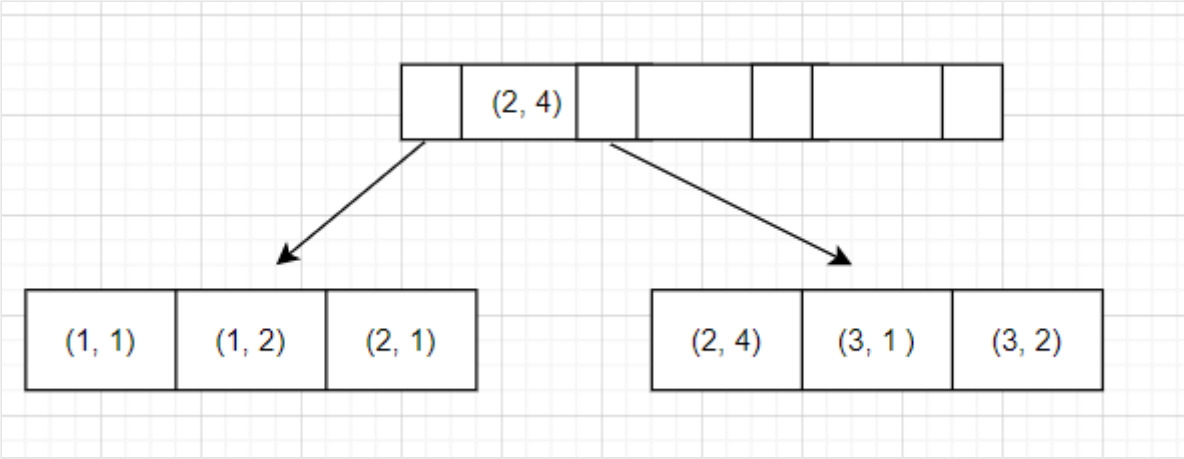
4.10. 什么是最左匹配原则?

如果 SQL 语句中用到了组合索引中的最左边的索引，那么这条 SQL 语句就可以利用这个组合索引去进行匹配。当遇到范围查询(>、<、between、like)就会停止匹配，后面的字段不会用到索引。

对(a,b,c)建立索引，查询条件使用 a/ab/abc 会走索引，使用 bc 不会走索引。

对(a,b,c,d)建立索引，查询条件为`a = 1 and b = 2 and c > 3 and d = 4`，那么，a,b,c三个字段能用到索引，而d就匹配不到。因为遇到了范围查询!

如下图，对(a, b) 建立索引，a 在索引树中是全局有序的，而 b 是全局无序，局部有序（当a相等时，会对b进行比较排序）。直接执行 `b = 2` 这种查询条件没有办法利用索引。

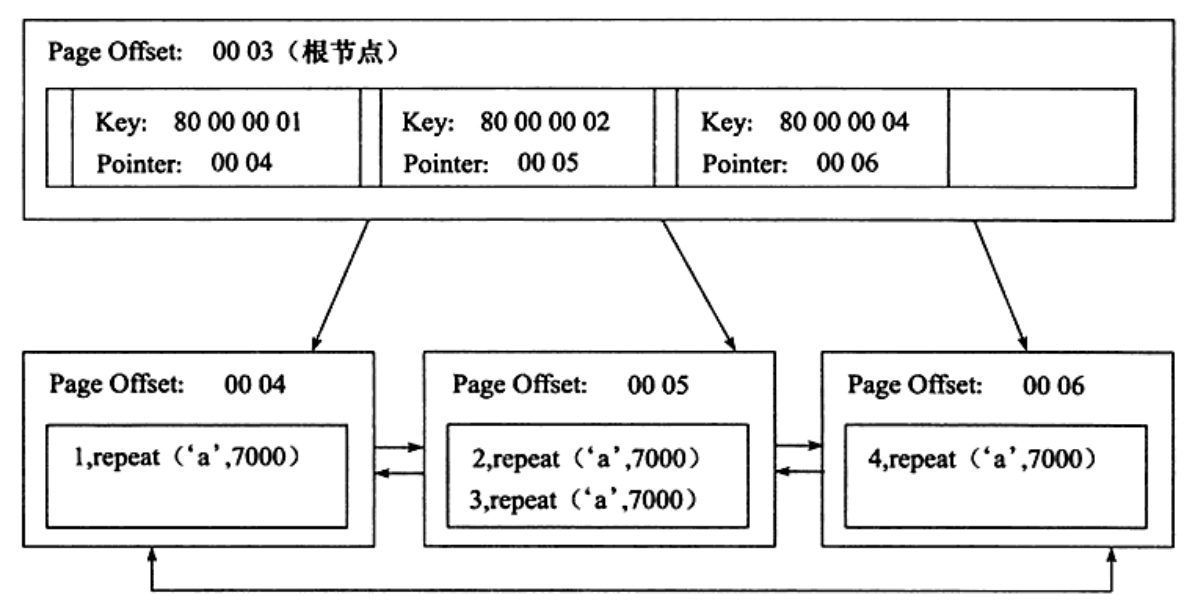


从局部来看，当a的值确定的时候，b是有序的。例如a = 1时，b值为1，2是有序的状态。当a=2时候，b的值为1,4也是有序状态。 因此，你执行 `a = 1 and b = 2` 是a,b字段能用到索引的。而你执行 `a > 1 and b = 2` 时，a字段能用到索引，b字段用不到索引。因为a的值此时是一个范围，不是固定的，在这个范围内b值不是有序的，因此b字段用不上索引。

4.11. 什么是聚集索引?

InnoDB使用表的主键构造主键索引树，同时叶子节点中存放的即为整张表的记录数据。聚集索引叶子节点的存储是逻辑上连续的，使用双向链表连接，叶子节点按照主键的顺序排序，因此对于主键的排序查找和范围查找速度比较快。

聚集索引的叶子节点就是整张表的行记录。InnoDB 主键使用的是聚簇索引。聚集索引要比非聚集索引查询效率高很多。



对于InnoDB来说，聚集索引一般是表中的主键索引，如果表中没有显示指定主键，则会选择表中的第一个不允许为NULL的唯一索引。如果没有主键也没有合适的唯一索引，那么innodb内部会生成一个隐藏的主键作为聚集索引，这个隐藏的主键长度为6个字节，它的值会随着数据的插入自增。

4.12. 什么是覆盖索引?

select的数据列只用从索引中就能够取得，不需要**回表**进行二次查询，换句话说查询列要被所使用的索引覆盖。对于innodb表的二级索引，如果索引能覆盖到查询的列，那么就可以避免对主键索引的二次查询。

不是所有类型的索引都可以成为覆盖索引。覆盖索引要存储索引列的值，而哈希索引、全文索引不存储索引列的值，所以MySQL只能使用b+树索引做覆盖索引。

对于使用了覆盖索引的查询，在查询前面使用explain，输出的extra列会显示为`using index`。

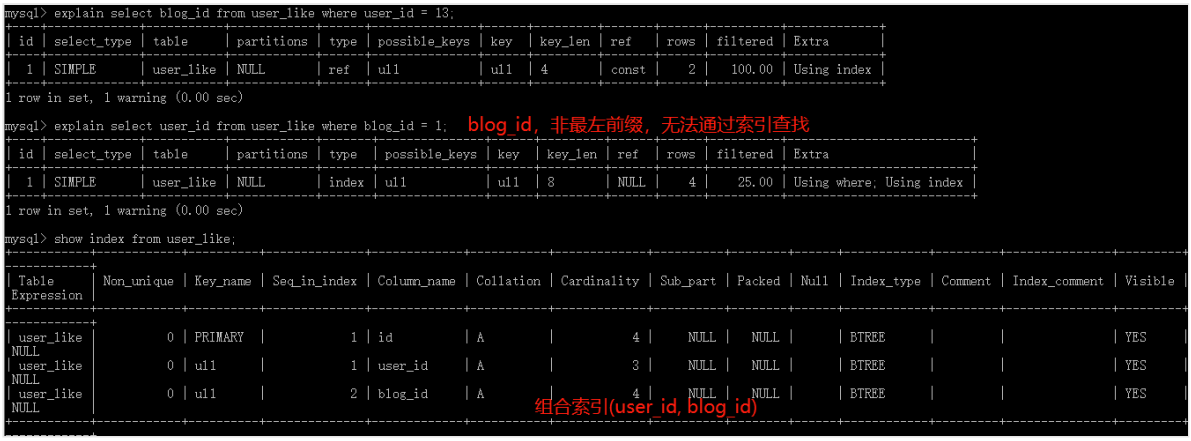
比如`user_like`用户点赞表，组合索引为(user_id, blog_id)，user_id和blog_id都不为null。

```
explain select blog_id from user_like where user_id = 13;
```

Extra中为`Using index`，查询的列被索引覆盖，并且where筛选条件符合最左前缀原则，通过**索引查找**就能直接找到符合条件的数据，不需要回表查询数据。

```
explain select user_id from user_like where blog_id = 1;
```

Extra中为`Using where; Using index`， 查询的列被索引覆盖，where筛选条件不符合最左前缀原则，无法通过索引查找找到符合条件的数据，但可以通过**索引扫描**找到符合条件的数据，也不需要回表查询数据。



4.13. 索引的设计原则?

- 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。
- 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。
- 索引不是越多越好，每个索引都需要额外的物理空间，维护也需要花费时间。
- 利用最左前缀原则。

4.14. 索引什么时候会失效?

导致索引失效的情况：

- 对于组合索引，不是使用组合索引最左边的字段，则不会使用索引
- 以%开头的like查询如`%abc`，无法使用索引；非%开头的like查询如`abc%`，相当于范围查询，会使用索引
- 查询条件中列类型是字符串，没有使用引号，可能会因为类型不同发生隐式转换，使索引失效
- 判断索引列是否不等于某个值时

- 对索引列进行运算
- 查询条件使用or连接，也会导致索引失效

4.15. 什么是前缀索引?

有时需要在很长的字符列上创建索引，这会造成索引特别大且慢。使用前缀索引可以避免这个问题。

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。

创建前缀索引的关键在于选择足够长的前缀以保证较高的索引选择性。索引选择性越高查询效率就越高，因为选择性高的索引可以让MySQL在查找时过滤掉更多的数据行。

建立前缀索引的方式：

```
// email列创建前缀索引
ALTER TABLE table_name ADD KEY(column_name(prefix_length));
```

5. 常见的存储引擎有哪些?

MySQL中常用的四种存储引擎分别是： MyISAM存储引擎、innoDB存储引擎、MEMORY存储引擎、ARCHIVE存储引擎。MySQL 5.5版本后默认的存储引擎为InnoDB。

InnoDB存储引擎

InnoDB是MySQL默认的事务型存储引擎，使用最广泛，基于聚簇索引建立的。InnoDB内部做了很多优化，如能够自动在内存中创建自适应hash索引，以加速读操作。

优点：支持事务和崩溃修复能力。InnoDB引入了行级锁和外键约束。

缺点：占用的数据空间相对较大。

适用场景：需要事务支持，并且有较高的并发读写频率。

MyISAM存储引擎

数据以紧密格式存储。对于只读数据，或者表比较小、可以容忍修复操作，可以使用MyISAM引擎。MyISAM会将表存储在两个文件中，数据文件.MYD和索引文件.MYI。

优点：访问速度快。

缺点：MyISAM不支持事务和行级锁，不支持崩溃后的安全恢复，也不支持外键。

适用场景：对事务完整性没有要求；只读的数据，或者表比较小，可以忍受修复repair操作。

MyISAM特性：

1. MyISAM对整张表加锁，而不是针对行。读取数据时会对需要读到的所有表加共享锁，写入时则对表加排它锁。但在读取表记录的同时，可以往表中插入新的记录（并发插入）。
2. 对于MyISAM表，MySQL可以手动或者自动执行检查和修复操作。执行表的修复可能会导致数据丢失，而且修复操作非常慢。可以通过`CHECK TABLE tablename`检查表的错误，如果有错误执行`REPAIR TABLE tablename`进行修复。

MEMORY存储引擎

MEMORY引擎将数据全部放在内存中，访问速度较快，但是一旦系统奔溃的话，数据都会丢失。

MEMORY引擎默认使用哈希索引，将键的哈希值和指向数据行的指针保存在哈希索引中。

优点：访问速度较快。

缺点：

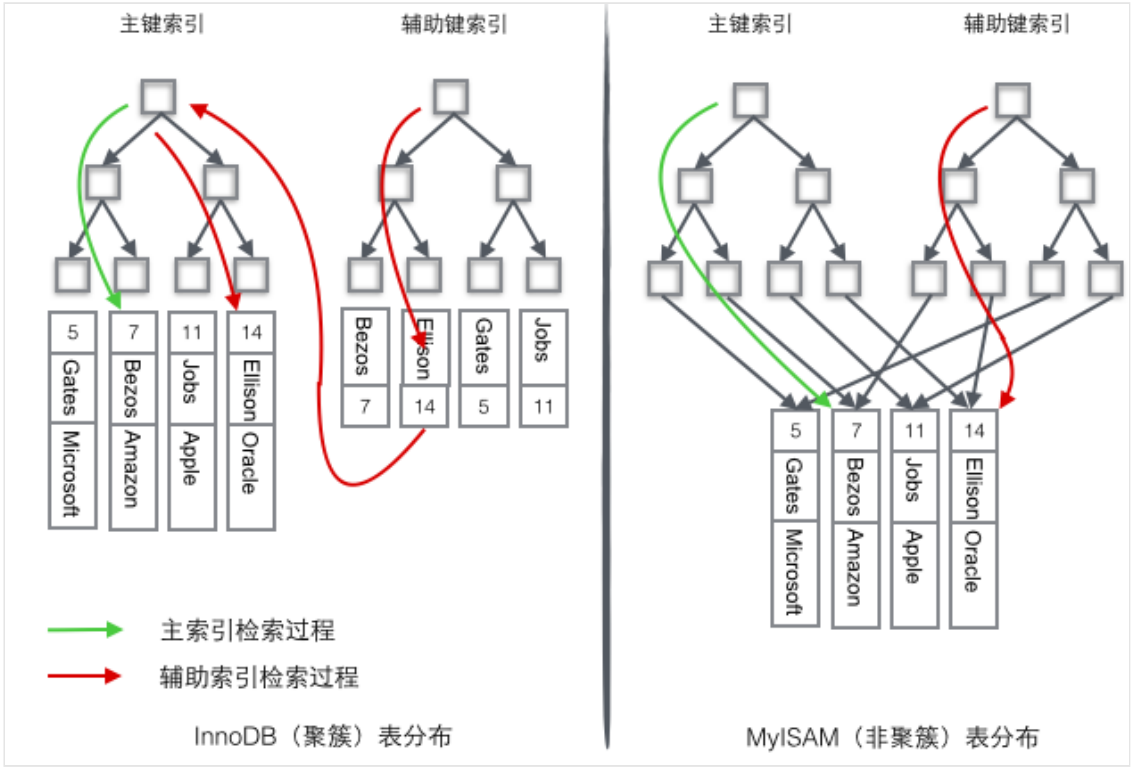
1. 哈希索引数据不是按照索引值顺序存储，无法用于排序。
2. 不支持部分索引匹配查找，因为哈希索引是使用索引列的全部内容来计算哈希值的。
3. 只支持等值比较，不支持范围查询。
4. 当出现哈希冲突时，存储引擎需要遍历链表中所有的行指针，逐行进行比较，直到找到符合条件的行。

ARCHIVE存储引擎

该存储引擎非常适合存储大量独立的、作为历史记录的数据。ARCHIVE提供了压缩功能，拥有高效的插入速度，但是这种引擎不支持索引，所以查询性能较差。

6. MyISAM和InnoDB的区别?

1. **是否支持行级锁**：`MyISAM` 只有表级锁，而 `InnoDB` 支持行级锁和表级锁，默认为行级锁。
2. **是否支持事务和崩溃后的安全恢复**： `MyISAM` 注重性能，每次查询具有原子性，其执行速度比 `InnoDB` 类型更快，但是不提供事务支持。而 `InnoDB` 提供事务支持，具有事务、回滚和崩溃修复能力。
3. **是否支持外键**： `MyISAM` 不支持，而 `InnoDB` 支持。
4. **是否支持MVCC**： `MyISAM` 不支持， `InnoDB` 支持。应对高并发事务，MVCC比单纯的加锁更高效。
5. `MyISAM` 不支持聚集索引， `InnoDB` 支持聚集索引。
    - `MyISAM` 引擎主键索引和其他索引区别不大，叶子节点都包含索引值和行指针。
    - `innoDB` 引擎二级索引叶子存储的是索引值和主键值（不是行指针），这样可以减少行移动和数据页分裂时二级索引的维护工作。



> 图片来源于网络

7. MVCC 实现原理?

MVCC(`Multiversion concurrency control`) 就是同一份数据保留多版本的一种方式，进而实现并发控制。在查询的时候，通过read view和版本链找到对应版本的数据。

作用：提升并发性能。对于高并发场景，MVCC比行级锁更有效、开销更小。

MVCC 实现原理如下：

MVCC 的实现依赖于版本链，版本链是通过表的三个隐藏字段实现。

- `DB_TRX_ID`：当前事务id，通过事务id的大小判断事务的时间顺序。
- `DB_ROLL_PRT`：回滚指针，指向当前行记录的上一个版本，通过这个指针将数据的多个版本连接在一起构成undo log版本链。
- `DB_ROLL_ID`：主键，如果数据表没有主键，InnoDB会自动生成主键。

每条表记录大概是这样的：

name	age	DB_ROW_ID	DB_TRX_ID	DB_ROLL_PTR
大彬	18	1	2	0x233333

使用事务更新行记录的时候，就会生成版本链，执行过程如下：

1. 用排他锁锁住该行；
2. 将该行原本的值拷贝到 undo log，作为旧版本用于回滚；
3. 修改当前行的值，生成一个新版本，更新事务id，使回滚指针指向旧版本的记录，这样就形成一条版本链。

下面举个例子方便大家理解。

1. 初始数据如下，其中DB_ROW_ID和DB_ROLL_PTR为空。

name	age	DB_ROW_ID	DB_TRX_ID	DB_ROLL_PTR
大彬	18	1	null	null

2. 事务A对该行数据做了修改，将age修改为12，效果如下：

name	age	DB_ROW_ID	DB_TRX_ID	DB_ROLL_PTR
大彬	12	1	1	0x100000

undo log

大彬	18	1	null	null

3. 之后事务B也对该行记录做了修改，将age修改为8，效果如下：

name	age	DB_ROW_ID	DB_TRX_ID	DB_ROLL_PTR
大彬	8	1	2	0x23333

undo log

大彬	12	1	1	0x100000

大彬	18	1	null	null

4. 此时undo log有两行记录，并且通过回滚指针连在一起。

接下来了解下read view的概念。

`read view`可以理解成对数据在每个时刻的状态拍成"照片"记录下来。这样获取某时刻的数据时就还是原来的"照片"上的数据，是不会变的。

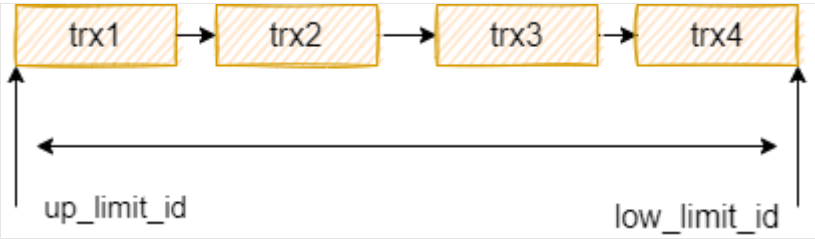
在`read view`内部维护一个活跃事务链表，表示生成`read view`的时候还在活跃的事务。这个链表包含在创建`read view`之前还未提交的事务，不包含创建`read view`之后提交的事务。

不同隔离级别创建read view的时机不同。

- read committed：每次执行select都会创建新的read_view，保证能读取到其他事务已经提交的修改。
- repeatable read：在一个事务范围内，第一次select时更新这个read_view，以后不会再更新，后续所有的select都是复用之前的read_view。这样可以保证事务范围内每次读取的内容都一样，即可重复读。

read view的记录筛选方式

前提：`DATA_TRX_ID`表示每个数据行的最新的事务ID；`up_limit_id`表示当前快照中的最先开始的事务；`low_limit_id`表示当前快照中的最慢开始的事务，即最后一个事务。



- 如果`DATA_TRX_ID` < `up_limit_id`：说明在创建`read view`时，修改该数据行的事务已提交，该版本的记录可被当前事务读取到。
- 如果`DATA_TRX_ID` >= `low_limit_id`：说明当前版本的记录的事务是在创建`read view`之后生成的，该版本的数据行不可以被当前事务访问。此时需要通过版本链找到上一个版本，然后重新判断该版本的记录对当前事务的可见性。
- 如果`up_limit_id` <= `DATA_TRX_ID` < `low_limit_i`：
    1. 需要在活跃事务链表中查找是否存在ID为`DATA_TRX_ID`的值的事务。
    2. 如果存在，因为在活跃事务链表中的事务是未提交的，所以该记录是不可见的。此时需要通过版本链找到上一个版本，然后重新判断该版本的可见性。
    3. 如果不存在，说明事务trx_id 已经提交了，这行记录是可见的。

总结：InnoDB 的`MVCC`是通过`read view`和版本链实现的，版本链保存有历史版本记录，通过`read view`判断当前版本的数据是否可见，如果不可见，再从版本链中找到上一个版本，继续进行判断，直到找到一个可见的版本。

8. 快照读和当前读

表记录有两种读取方式。

- 快照读：读取的是快照版本。普通的SELECT就是快照读。通过MVCC来进行并发控制的，不用加锁。
- 当前读：读取的是最新版本。`UPDATE、DELETE、INSERT、SELECT … LOCK IN SHARE MODE、SELECT … FOR UPDATE`是当前读。

快照读情况下，InnoDB通过mvcc机制避免了幻读现象。而mvcc机制无法避免当前读情况下出现的幻读现象。因为当前读每次读取的都是最新数据，这时如果两次查询中间有其它事务插入数据，就会产生幻读。

下面举个例子说明下：

1. 首先，user表只有两条记录，具体如下：

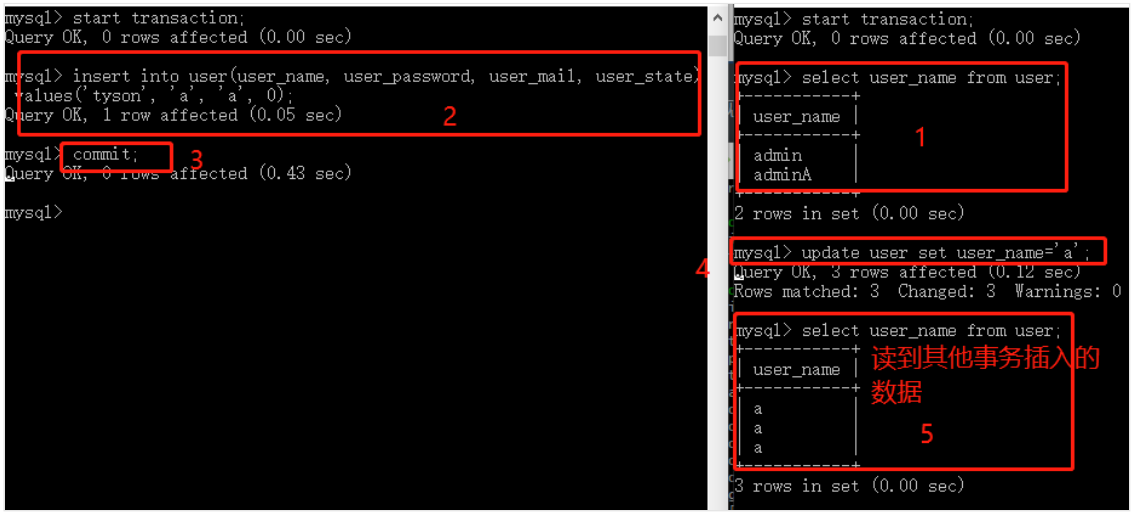
2. 事务a和事务b同时开启事务`start transaction`；
3. 事务a插入数据然后提交；

```
insert into user(user_name, user_password, user_mail, user_state)
values('tyson', 'a', 'a', 0);
```

4. 事务b执行全表的update；

```
update user set user_name = 'a';
```

5. 事务b然后执行查询，查到了事务a中插入的数据。（下图左边是事务b，右边是事务a）



以上就是当前读出现的幻读现象。

那么MySQL如何实现避免幻读?

- 在快照读情况下，MySQL通过mvcc来避免幻读。
- 在当前读情况下，MySQL通过next-key来避免幻读（加行锁和间隙锁来实现的）。

next-key包括两部分：行锁和间隙锁。行锁是加在索引上的锁，间隙锁是加在索引之间的。

`Serializable`隔离级别也可以避免幻读，会锁住整张表，并发性极低，一般不会使用。

9. 共享锁和排他锁

SELECT 的读取锁定主要分为两种方式：共享锁和排他锁。

```
select * from table where id<6 lock in share mode;--共享锁
select * from table where id<6 for update;--排他锁
```

这两种方式主要的不同在于`LOCK IN SHARE MODE`多个事务同时更新同一个表单时很容易造成死锁。

申请排他锁的前提是，没有线程对该结果集的任何行数据使用排它锁或者共享锁，否则申请会受到阻塞。在进行事务操作时，MySQL会对查询结果集的每行数据添加排它锁，其他线程对这些数据的更改或删除操作会被阻塞（只能读操作），直到该语句的事务被commit语句或rollback语句结束为止。

`SELECT... FOR UPDATE`使用注意事项：

1. for update 仅适用于Innodb，且必须在事务范围内才能生效。
2. 根据主键进行查询，查询条件为 like或者不等于，主键字段产生**表锁**。
3. 根据非索引字段进行查询，name字段产生**表锁**。

10. 大表怎么优化?

某个表有近千万数据，查询比较慢，如何优化?

当MySQL单表记录数过大时，数据库的性能会明显下降，一些常见的优化措施如下:

- 限定数据的范围。比如：用户在查询历史信息的时候，可以控制在一个月的时间范围内;
- 读写分离： 经典的数据库拆分方案，主库负责写，从库负责读;
- 通过分库分表的方式进行优化，主要有垂直拆分和水平拆分。

11. MySQL 执行计划了解吗?

通过 explain 命令获取 select 语句的执行计划，了解 select 语句以下信息:

- 表的加载顺序
- sql 的查询类型
- 可能用到哪些索引，实际上用到哪些索引
- 读取的行数
- ...

详细内容可以参考我的另一篇文章

12. bin log/redo log/undo log

MySQL日志主要包括查询日志、慢查询日志、事务日志、错误日志、二进制日志等。其中比较重要的是bin log（二进制日志）和 redo log（重做日志）和 undo log（回滚日志）。

bin log

二进制日志（bin log）是MySQL数据库级别的文件，记录对MySQL数据库执行修改的所有操作，不会记录select和show语句，主要用于恢复数据库和同步数据库。

redo log

重做日志（redo log）是Innodb引擎级别，用来记录Innodb存储引擎的事务日志，不管事务是否提交都会记录下来，用于数据恢复。当数据库发生故障，InnoDB存储引擎会使用redo log恢复到发生故障前的时刻，以此来保证数据的完整性。将参数 `innodb_flush_log_at_tx_commit` 设置为1，那么在执行commit时会将redo log同步写到磁盘。

undo log

除了记录redo log外，当进行数据修改时还会记录undo log，undo log用于数据的撤回操作，它保留了记录修改前的内容。通过undo log可以实现事务回滚，并且可以根据undo log回溯到某个特定的版本的数据，**实现MVCC**。

13. bin log和redo log有什么区别?

1. bin log会记录所有日志记录，包括innoDB、MyISAM等存储引擎的日志；redo log只记录innoDB自身的事务日志。
2. bin log只在事务提交前写入到磁盘，一个事务只写一次；而在事务进行过程，会有redo log不断写入磁盘。
3. binlog 是逻辑日志，记录的是SQL语句的原始逻辑；redo log 是物理日志，记录的是在某个数据页上做了什么修改。

14. 讲一下MySQL架构?

MySQL主要分为 Server 层和存储引擎层：

- **Server 层**：主要包括连接器、查询缓存、分析器、优化器、执行器等，所有跨存储引擎的功能都在这一层实现，比如存储过程、触发器、视图，函数等，还有一个通用的日志模块 binglog 日志模块。
- **存储引擎**： 主要负责数据的存储和读取。server 层通过api与存储引擎进行通信。

Server 层基本组件

- **连接器：** 当客户端连接 MySQL 时，server层会对其进行身份认证和权限校验。
- **查询缓存:** 执行查询语句的时候，会先查询缓存，先校验这个 sql 是否执行过，如果有缓存这个 sql，就会直接返回给客户端，如果没有命中，就会执行后续的操作。
- **分析器:** 没有命中缓存的话，SQL 语句就会经过分析器，主要分为两步，词法分析和语法分析，先看 SQL 语句要做什么，再检查 SQL 语句语法是否正确。
- **优化器：** 优化器对查询进行优化，包括重写查询、决定表的读写顺序以及选择合适的索引等，生成执行计划。
- **执行器：** 首先执行前会校验该用户有没有权限，如果没有权限，就会返回错误信息，如果有权限，就会根据执行计划去调用引擎的接口，返回结果。

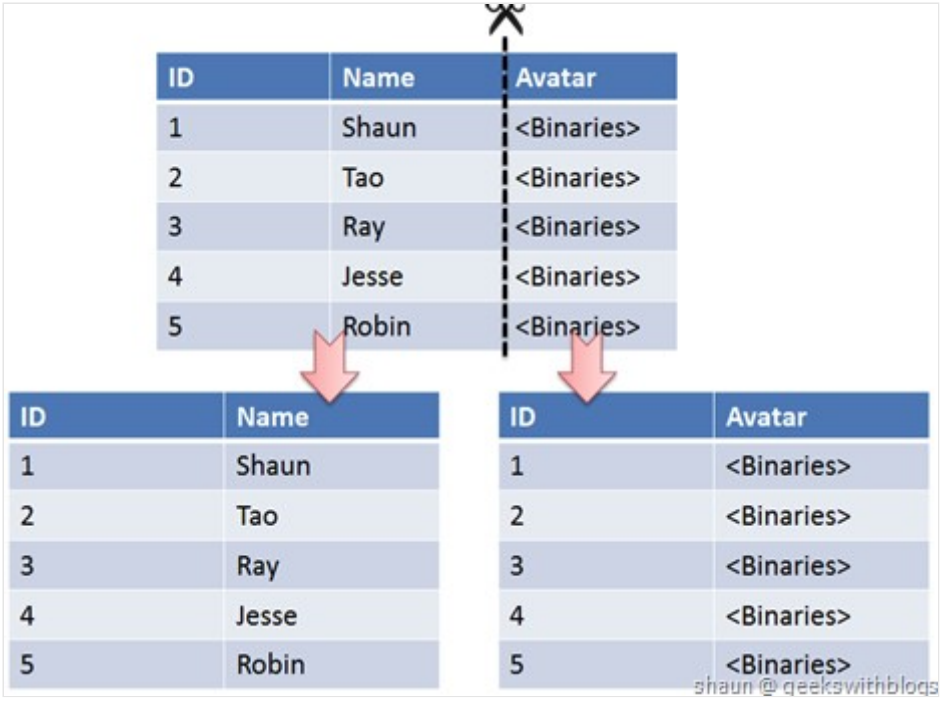
15. 分库分表

当单表的数据量达到1000W或100G以后，优化索引、添加从库等可能对数据库性能提升效果不明显，此时就要考虑对其进行切分了。切分的目的就在于减少数据库的负担，缩短查询的时间。

数据切分可以分为两种方式：垂直划分和水平划分。

垂直划分

垂直划分数据库是根据业务进行划分，例如购物场景，可以将库中涉及商品、订单、用户的表分别划分出成一个库，通过降低单库的大小来提高性能，但这种方式并没有解决高数据量带来的性能损耗。同样的，分表的情况就是将一个大表根据业务功能拆分成一个个子表，例如商品基本信息和商品描述，商品基本信息一般会展示在商品列表，商品描述在商品详情页，可以将商品基本信息和商品描述拆分成两张表。



> 图片来源于网络

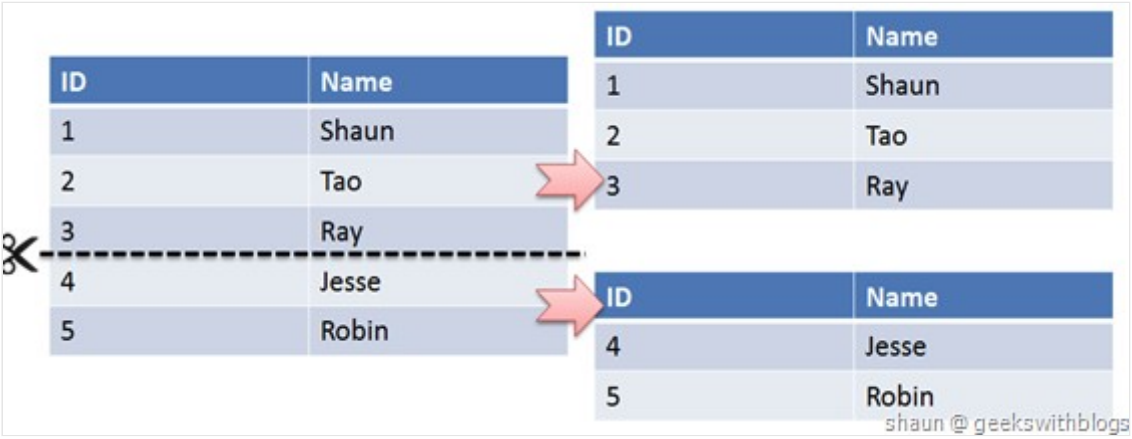
优点：行记录变小，数据页可以存放更多记录，在查询时减少I/O次数。

缺点：

- 主键出现冗余，需要管理冗余列；
- 会引起表连接JOIN操作，可以通过在业务服务器上进行join来减少数据库压力；
- 依然存在单表数据量过大的问题。

水平划分

水平划分是根据一定规则，例如时间或id序列值等进行数据的拆分。比如根据年份来拆分不同的数据库。每个数据库结构一致，但是数据得以拆分，从而提升性能。



> 图片来源于网络

优点：单库（表）的数据量得以减少，提高性能；切分出的表结构相同，程序改动较少。

缺点：

- 分片事务一致性难以解决
- 跨节点join性能差，逻辑复杂
- 数据分片在扩容时需要迁移

16. 什么是分区表?

分区表是一个独立的逻辑表，但是底层由多个物理子表组成。

当查询条件的数据分布在某一个分区的时候，查询引擎只会去某一个分区查询，而不是遍历整个表。在管理层面，如果需要删除某一个分区的数据，只需要删除对应的分区即可。

17. 分区表类型

1. 按照范围分区。

```
CREATE TABLE test_range_partition(
       id INT auto_increment,
       createdate DATETIME,
       primary key (id,createdate)
   )
   PARTITION BY RANGE (TO_DAYS(createdate) ) (
      PARTITION p201801 VALUES LESS THAN ( TO_DAYS('20180201') ),
      PARTITION p201802 VALUES LESS THAN ( TO_DAYS('20180301') ),
      PARTITION p201803 VALUES LESS THAN ( TO_DAYS('20180401') ),
      PARTITION p201804 VALUES LESS THAN ( TO_DAYS('20180501') ),
      PARTITION p201805 VALUES LESS THAN ( TO_DAYS('20180601') ),
      PARTITION p201806 VALUES LESS THAN ( TO_DAYS('20180701') ),
```

```
    PARTITION p201807 VALUES LESS THAN ( TO_DAYS('20180801') ),
    PARTITION p201808 VALUES LESS THAN ( TO_DAYS('20180901') ),
    PARTITION p201809 VALUES LESS THAN ( TO_DAYS('20181001') ),
    PARTITION p201810 VALUES LESS THAN ( TO_DAYS('20181101') ),
    PARTITION p201811 VALUES LESS THAN ( TO_DAYS('20181201') ),
    PARTITION p201812 VALUES LESS THAN ( TO_DAYS('20190101') )
);
```

在`/var/lib/mysql/data/`可以找到对应的数据文件，每个分区表都有一个使用#分隔命名的表文件：

```
   -rw-r----- 1 MySQL MySQL    65 Mar 14 21:47 db.opt
   -rw-r----- 1 MySQL MySQL  8598 Mar 14 21:50 test_range_partition.frm
   -rw-r----- 1 MySQL MySQL 98304 Mar 14 21:50
test_range_partition#P#p201801.ibd
   -rw-r----- 1 MySQL MySQL 98304 Mar 14 21:50
test_range_partition#P#p201802.ibd
   -rw-r----- 1 MySQL MySQL 98304 Mar 14 21:50
test_range_partition#P#p201803.ibd
...
```

2. list分区。对于List分区，分区字段必须是已知的，如果插入的字段不在分区时枚举值中，将无法插入。

```
create table test_list_partiotion
   (
       id int auto_increment,
       data_type tinyint,
       primary key(id,data_type)
   )partition by list(data_type)
   (
       partition p0 values in (0,1,2,3,4,5,6),
       partition p1 values in (7,8,9,10,11,12),
       partition p2 values in (13,14,15,16,17)
   );
```

3. hash分区，可以将数据均匀地分布到预先定义的分区中。

```
create table test_hash_partiotion
   (
       id int auto_increment,
       create_date datetime,
       primary key(id,create_date)
   )partition by hash(year(create_date)) partitions 10;
```

18. 分区的问题?

1. 打开和锁住所有底层表的成本可能很高。当查询访问分区表时，MySQL需要打开并锁住所有的底层表，这个操作在分区过滤之前发生，所以无法通过分区过滤来降低此开销，会影响到查询速度。可以通过批量操作来降低此类开销，比如批量插入、LOAD DATA INFILE和一次删除多行数据。
2. 维护分区的成本可能很高。例如重组分区，会先创建一个临时分区，然后将数据复制到其中，最后再删除原分区。
3. 所有分区必须使用相同的存储引擎。

19. 查询语句执行流程?

查询语句的执行流程如下：权限校验、查询缓存、分析器、优化器、权限校验、执行器、引擎。

举个例子，查询语句如下：

```
select * from user where id > 1 and name = '大彬';
```

1. 首先检查权限，没有权限则返回错误；
2. MySQL以前会查询缓存，缓存命中则直接返回，没有则执行下一步；
3. 词法分析和语法分析。提取表名、查询条件，检查语法是否有错误；
4. 两种执行方案，先查 `id > 1` 还是 `name = '大彬'`，优化器根据自己的优化算法选择执行效率最好的方案；
5. 校验权限，有权限就调用数据库引擎接口，返回引擎的执行结果。

20. 更新语句执行过程?

更新语句执行流程如下：分析器、权限校验、执行器、引擎、redo log(prepare 状态)、binlog、redo log(commit状态)

举个例子，更新语句如下：

```
update user set name = '大彬' where id = 1;
```

1. 先查询到 id 为1的记录，有缓存会使用缓存。
2. 拿到查询结果，将 name 更新为 大彬，然后调用引擎接口，写入更新数据，innodb 引擎将数据保存在内存中，同时记录 redo log，此时 redo log 进入 prepare 状态。
3. 执行器收到通知后记录 binlog，然后调用引擎接口，提交 redo log 为提交状态。
4. 更新完成。

为什么记录完 redo log，不直接提交，先进入prepare状态?

假设先写 redo log 直接提交，然后写 binlog，写完 redo log 后，机器挂了，binlog 日志没有被写入，那么机器重启后，这台机器会通过 redo log 恢复数据，但是这个时候 binlog 并没有记录该数据，后续进行机器备份的时候，就会丢失这一条数据，同时主从同步也会丢失这一条数据。

21. exist和in的区别?

exists 用于对外表记录做筛选。

exists 会遍历外表，将外查询表的每一行，代入内查询进行判断。当 exists 里的条件语句能够返回记录行时，条件就为真，返回外表当前记录。反之如果exists里的条件语句不能返回记录行，条件为假，则外表当前记录被丢弃。

```
select a.* from A awhere exists(select 1 from B b where a.id=b.id)
```

in 是先把后边的语句查出来放到临时表中，然后遍历临时表，将临时表的每一行，代入外查询去查找。

```
select * from Awhere id in(select id from B)
```

子查询的表大的时候，使用exists可以有效减少总的循环次数来提升速度；当外查询的表大的时候，使用IN可以有效减少对外查询表循环遍历来提升速度。

22. MySQL中int(10)和char(10)的区别?

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

23. truncate、delete与drop区别?

相同点：

1. truncate和不带where子句的delete、以及drop都会删除表内的数据。
2. drop、truncate都是DDL语句（数据定义语言），执行后会自动提交。

不同点：

1. truncate 和 delete 只删除数据不删除表的结构；drop 语句将删除表的结构被依赖的约束、触发器、索引；
2. 一般来说，执行速度: drop > truncate > delete。

24. having和where区别?

- 二者作用的对象不同，where子句作用于表和视图，having作用于组。
- WHERE在数据分组前进行过滤，HAVING在数据分组后进行过滤。

25. 什么是MySQL主从同步?

主从同步使得数据可以从一个数据库服务器复制到其他服务器上，在复制数据时，一个服务器充当主服务器（master），其余的服务器充当从服务器（slave）。

因为复制是异步进行的，所以从服务器不需要一直连接着主服务器，从服务器甚至可以通过拨号断断续续地连接主服务器。通过配置文件，可以指定复制所有的数据库，某个数据库，甚至是某个数据库上的某个表。

26. 为什么要做主从同步?

1. 读写分离，使数据库能支撑更大的并发。
2. 在主服务器上生成实时数据，而在从服务器上分析这些数据，从而提高主服务器的性能。
3. 数据备份，保证数据的安全。

27. 乐观锁和悲观锁是什么?

数据库中的并发控制是确保在多个事务同时存取数据库中同一数据时不破坏事务的隔离性和统一性以及数据库的统一性。乐观锁和悲观锁是并发控制主要采用的技术手段。

- 悲观锁：假定会发生并发冲突，在查询完数据的时候就把事务锁起来，直到提交事务。实现方式：使用数据库中的锁机制。
- 乐观锁：假设不会发生并发冲突，只在提交操作时检查是否数据是否被修改过。给表增加version字段，在修改提交之前检查version与原来取到的version值是否相等，若相等，表示数据没有被修改，可以更新，否则，数据为脏数据，不能更新。实现方式：乐观锁一般使用版本号机制或CAS算法实现。

28. 用过processlist吗?

`show processlist` 或 `show full processlist` 可以查看当前 MySQL 是否有压力，正在运行的sql，有没有慢 SQL 正在执行。返回参数如下：

- **id** - 线程ID，可以用：`kill id;` 杀死一个线程，很有用
- **db** - 数据库
- **user** - 用户

- **host** - 连库的主机IP
- **command** - 当前执行的命令，比如最常见的：Sleep，Query，Connect 等
- **time** - 消耗时间，单位秒，很有用
- **state** - 执行状态

  sleep，线程正在等待客户端发送新的请求

  query，线程正在查询或者正在将结果发送到客户端

  Sorting result，线程正在对结果集进行排序

  Locked，线程正在等待锁
- **info** - 执行的SQL语句，很有用

> 最后给大家分享一个github仓库，上面放了**200多本经典的计算机书籍**，包括C语言、C++、Java、Python、前端、数据库、操作系统、计算机网络、数据结构和算法、机器学习、编程人生等，可以star一下，下次找书直接在上面搜索，仓库持续更新中~
>
> github地址：https://github.com/Tyson0314/java-books
>
> 如果github访问不了，可以访问gitee仓库。
>
> gitee地址：https://gitee.com/tysondai/java-books

Redis

1. Redis是什么?

Redis（`Remote Dictionary Server`）是一个使用 C 语言编写的，高性能非关系型的键值对数据库。与传统数据库不同的是，Redis 的数据是存在内存中的，所以读写速度非常快，被广泛应用于缓存方向。Redis可以将数据写入磁盘中，保证了数据的安全不丢失，而且Redis的操作是原子性的。

2. Redis优缺点?

优点:

1. **基于内存操作**，内存读写速度快。
2. Redis是**单线程**的，避免线程切换开销及多线程的竞争问题。单线程是指网络请求使用一个线程来处理，即一个线程处理所有网络请求，Redis 运行时不止有一个线程，比如数据持久化的过程会另起线程。
3. **支持多种数据类型**，包括String、Hash、List、Set、ZSet等。
4. **支持持久化。**Redis支持RDB和AOF两种持久化机制，持久化功能可以有效地避免数据丢失问题。
5. **支持事务。**Redis的所有操作都是原子性的，同时Redis还支持对几个操作合并后的原子性执行。
6. **支持主从复制。**主节点会自动将数据同步到从节点，可以进行读写分离。

缺点:

1. 对结构化查询的支持比较差。
2. 数据库容量受到物理内存的限制，不适合用作海量数据的高性能读写，因此Redis适合的场景主要局限在较小数据量的操作。
3. Redis 较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。

3. Redis为什么这么快?

- **基于内存**：Redis是使用内存存储，没有磁盘IO上的开销。数据存在内存中，读写速度快。
- **单线程实现**（Redis 6.0以前）：Redis使用单个线程处理请求，避免了多个线程之间线程切换和锁资源争用的开销。
- **IO多路复用模型**：Redis 采用 IO 多路复用技术。Redis 使用单线程来轮询描述符，将数据库的操作都转换成了事件，不在网络I/O上浪费过多的时间。
- **高效的数据结构**：Redis 每种数据类型底层都做了优化，目的就是为了追求更快的速度。

4. Redis为何选择单线程

- 避免过多的**上下文切换开销。**程序始终运行在进程中单个线程内，没有多线程切换的场景。
- **避免同步机制的开销**：如果 Redis选择多线程模型，需要考虑数据同步的问题，则必然会引入某些同步机制，会导致在操作数据过程中带来更多的开销，增加程序复杂度的同时还会降低性能。
- **实现简单，方便维护**：如果 Redis使用多线程模式，那么所有的底层数据结构的设计都必须考虑线程安全问题，那么 Redis 的实现将会变得更加复杂。

5. Redis6.0为何引入多线程?

Redis支持多线程主要有两个原因:

- 可以充分利用服务器 CPU 资源，单线程模型的主线程只能利用一个cpu;
- 多线程任务可以分摊 Redis 同步 IO 读写的负荷。

6. Redis应用场景有哪些?

1. **缓存热点数据**，缓解数据库的压力。
2. 利用 Redis 原子性的自增操作，可以实现**计数器**的功能，比如统计用户点赞数、用户访问数等。
3. **简单的消息队列**，可以使用Redis自身的发布/订阅模式或者List来实现简单的消息队列，实现异步操作。
4. **限速器**，可用于限制某个用户访问某个接口的频率，比如秒杀场景用于防止用户快速点击带来不必要的压力。
5. **好友关系**，利用集合的一些命令，比如交集、并集、差集等，实现共同好友、共同爱好之类的功能。

7. Memcached和Redis的区别?

1. Redis 只使用**单核**，而 Memcached 可以使用多核。
2. MemCached 数据结构单一，仅用来缓存数据，而 **Redis 支持多种数据类型。**
3. MemCached 不支持数据持久化，重启后数据会消失。**Redis 支持数据持久化。**
4. **Redis 提供主从同步机制和 cluster 集群部署能力**，能够提供高可用服务。Memcached 没有提供原生的集群模式，需要依靠客户端实现往集群中分片写入数据。
5. Redis 的速度比 Memcached 快很多。
6. Redis 使用**单线程的多路 IO 复用模型**，Memcached使用多线程的非阻塞 IO 模型。

8. Redis 数据类型有哪些?

基本数据类型:

1、**String**：最常用的一种数据类型，String类型的值可以是字符串、数字或者二进制，但值最大不能超过512MB。

2、**Hash**：Hash 是一个键值对集合。

3、**Set**：无序去重的集合。Set 提供了交集、并集等方法，对于实现共同好友、共同关注等功能特别方便。

4、**List**：有序可重复的集合，底层是依赖双向链表实现的。

5、**SortedSet**：有序Set。内部维护了一个`score`的参数来实现。适用于排行榜和带权重的消息队列等场景。

特殊的数据类型:

1、**Bitmap**：位图，可以认为是一个以位为单位数组，数组中的每个单元只能存0或者1，数组的下标在Bitmap 中叫做偏移量。Bitmap的长度与集合中元素个数无关，而是与基数的上限有关。

2、**Hyperloglog**。HyperLogLog 是用来做基数统计的算法，其优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定的、并且是很小的。典型的使用场景是统计独立访客。

3、**Geospatial** ：主要用于存储地理位置信息，并对存储的信息进行操作，适用场景如定位、附近的人等。

9. keys命令存在的问题?

redis的单线程的。keys指令会导致线程阻塞一段时间，直到执行完毕，服务才能恢复。scan采用渐进式遍历的方式来解决keys命令可能带来的阻塞问题，每次scan命令的时间复杂度是`o(1)`，但是要真正实现keys的功能，需要执行多次scan。

scan的缺点：在scan的过程中如果有键的变化（增加、删除、修改），遍历过程可能会有以下问题：新增的键可能没有遍历到，遍历出了重复的键等情况，也就是说scan并不能保证完整的遍历出来所有的键。

10. SortedSet和List异同点?

相同点:

1. 都是有序的;
2. 都可以获得某个范围内的元素。

不同点:

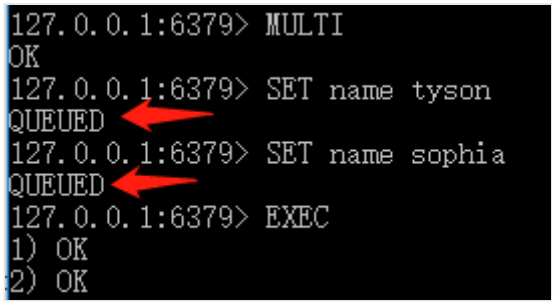
1. 列表基于链表实现，获取两端元素速度快，访问中间元素速度慢;
2. 有序集合基于散列表和跳跃表实现，访问中间元素时间复杂度是OlogN;
3. 列表不能简单的调整某个元素的位置，有序列表可以（更改元素的分数）;
4. 有序集合更耗内存。

11. Redis事务

事务的原理是将一个事务范围内的若干命令发送给Redis，然后再让Redis依次执行这些命令。

事务的生命周期:

1. 使用MULTI开启一个事务
2. 在开启事务的时候，每次操作的命令将会被插入到一个队列中，同时这个命令并不会被真的执行
3. EXEC命令进行提交事务



一个事务范围内某个命令出错不会影响其他命令的执行，不保证原子性:

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set a 1
QUEUED
127.0.0.1:6379> set b 1 2
QUEUED
127.0.0.1:6379> set c 3
QUEUED
127.0.0.1:6379> exec
1) OK
2) (error) ERR syntax error
3) OK
```

WATCH命令

`WATCH`命令可以监控一个或多个键，一旦其中有一个键被修改，之后的事务就不会执行（类似于乐观锁）。执行`EXEC`命令之后，就会自动取消监控。

```
127.0.0.1:6379> watch name
OK
127.0.0.1:6379> set name 1
OK
127.0.0.1:6379> multi
```

```
OK
127.0.0.1:6379> set name 2
QUEUED
127.0.0.1:6379> set gender 1
QUEUED
127.0.0.1:6379> exec
(nil)
127.0.0.1:6379> get gender
(nil)
```

比如上面的代码中：

1. `watch name`开启了对`name`这个`key`的监控
2. 修改`name`的值
3. 开启事务a
4. 在事务a中设置了`name`和`gender`的值
5. 使用`EXEC`命令进提交事务
6. 使用命令`get gender`发现不存在，即事务a没有执行

使用`UNWATCH`可以取消`WATCH`命令对`key`的监控，所有监控锁将会被取消。

12. 持久化机制

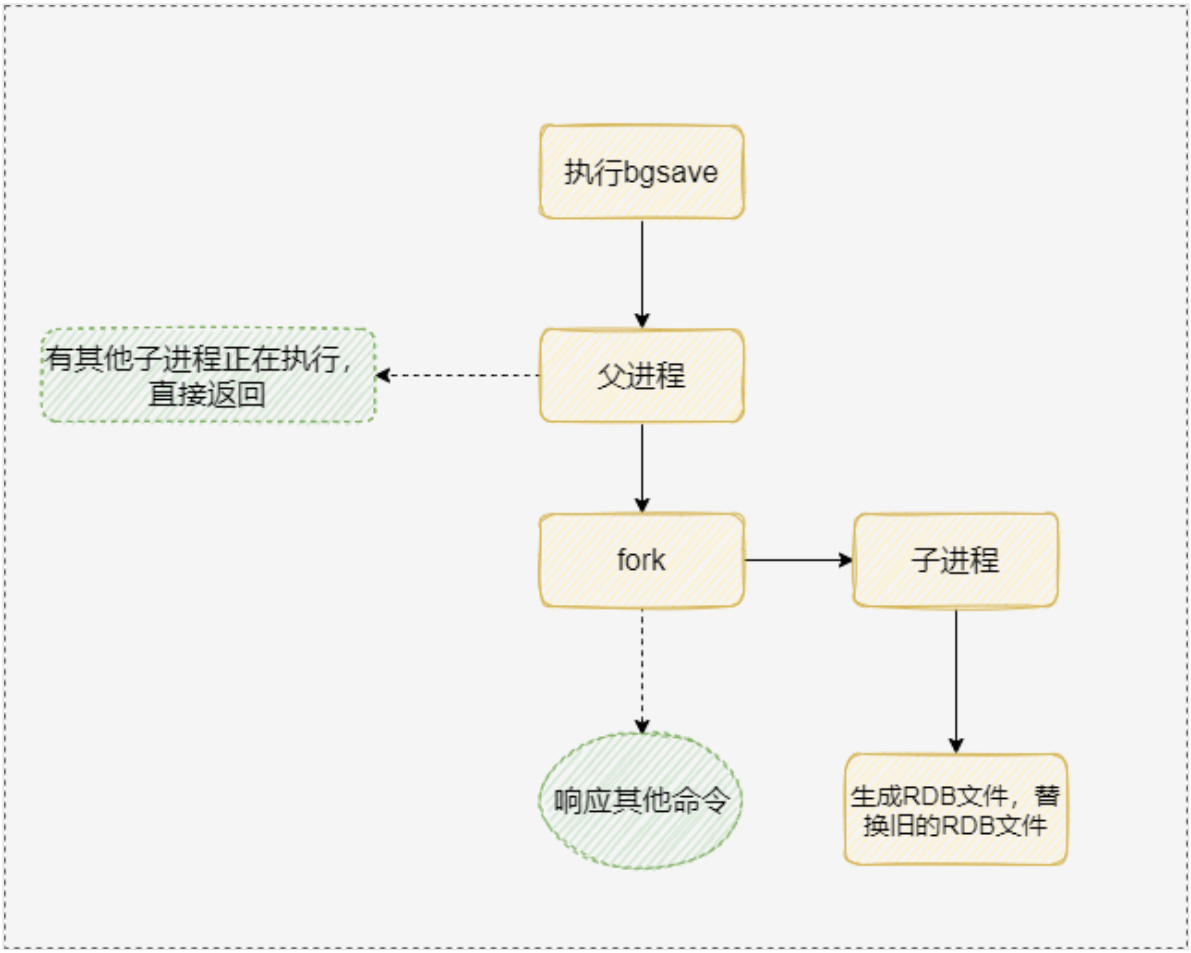
持久化就是把**内存的数据写到磁盘中**，防止服务宕机导致内存数据丢失。

Redis支持两种方式的持久化，一种是`RDB`的方式，一种是`AOF`的方式。**前者会根据指定的规则定时将内存中的数据存储在硬盘上**，而**后者在每次执行完命令后将命令记录下来**。一般将两者结合使用。

12.1. RDB方式

`RDB`是 Redis 默认的持久化方案。RDB持久化时会将内存中的数据写入到磁盘中，在指定目录下生成一个`dump.rdb`文件。Redis 重启会加载`dump.rdb`文件恢复数据。

`bgsave`是主流的触发 RDB 持久化的方式，执行过程如下：



- 执行`BGSAVE`命令
- Redis 父进程判断当前**是否存在正在执行的子进程**，如果存在，`BGSAVE`命令直接返回。
- 父进程执行`fork`操作**创建子进程**，fork操作过程中父进程会阻塞。
- 父进程`fork`完成后，**父进程继续接收并处理客户端的请求**，而**子进程开始将内存中的数据写进硬盘的临时文件**；
- 当子进程写完所有数据后会**用该临时文件替换旧的 RDB 文件。**

Redis启动时会读取RDB快照文件，将数据从硬盘载入内存。通过 RDB 方式的持久化，一旦Redis异常退出，就会丢失最近一次持久化以后更改的数据。

触发 RDB 持久化的方式：

1. **手动触发**：用户执行`SAVE`或`BGSAVE`命令。`SAVE`命令执行快照的过程会阻塞所有客户端的请求，应避免在生产环境使用此命令。`BGSAVE`命令可以在后台异步进行快照操作，快照的同时服务器还可以继续响应客户端的请求，因此需要手动执行快照时推荐使用`BGSAVE`命令。
2. **被动触发**：
    - 根据配置规则进行自动快照，如`SAVE 100 10`，100秒内至少有10个键被修改则进行快照。
    - 如果从节点执行全量复制操作，主节点会自动执行`BGSAVE`生成 RDB 文件并发送给从节点。
    - 默认情况下执行`shutdown`命令时，如果没有开启 AOF 持久化功能则自动执行·BGSAVE·。

优点：

1. **Redis 加载 RDB 恢复数据远远快于 AOF 的方式。**
2. 使用单独子进程来进行持久化，主进程不会进行任何 IO 操作，**保证了 Redis 的高性能。**

缺点：

1. **RDB方式数据无法做到实时持久化。**因为`BGSAVE`每次运行都要执行`fork`操作创建子进程，属于重量级操作，频繁执行成本比较高。

2. RDB 文件使用特定二进制格式保存，Redis 版本升级过程中有多个格式的 RDB 版本，**存在老版本 Redis 无法兼容新版 RDB 格式的问题。**

12.2. AOF方式

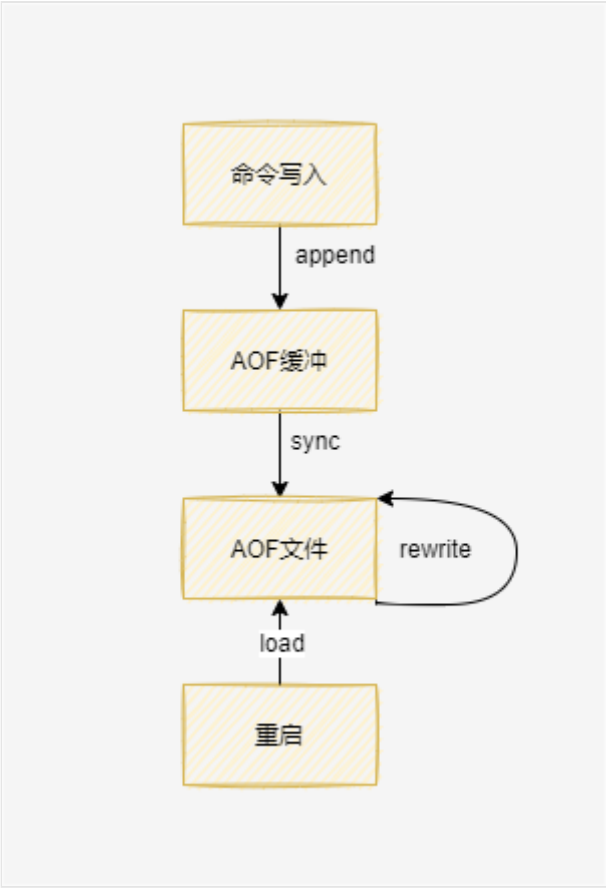
AOF（append only file）持久化：以独立日志的方式记录每次写命令，Redis重启时会重新执行AOF文件中的命令达到恢复数据的目的。AOF的主要作用是**解决了数据持久化的实时性**，AOF 是Redis持久化的主流方式。

默认情况下Redis没有开启AOF方式的持久化，可以通过`appendonly`参数启用：`appendonly yes`。开启AOF方式持久化后每执行一条写命令，Redis就会将该命令写进`aof_buf`缓冲区，AOF缓冲区根据对应的策略向硬盘做同步操作。

默认情况下系统**每30秒**会执行一次同步操作。为了防止缓冲区数据丢失，可以在Redis写入AOF文件后主动要求系统将缓冲区数据同步到硬盘上。可以通过`appendfsync`参数设置同步的时机。

```
appendfsync always //每次写入aof文件都会执行同步，最安全最慢，不建议配置
appendfsync everysec  //既保证性能也保证安全，建议配置
appendfsync no //由操作系统决定何时进行同步操作
```

接下来看一下 AOF 持久化执行流程：



1. 所有的写入命令会追加到 AOP 缓冲区中。
2. AOF 缓冲区根据对应的策略向硬盘同步。
3. 随着 AOF 文件越来越大，需要定期对 AOF 文件进行重写，达到压缩文件体积的目的。AOF文件重写是把Redis进程内的数据转化为写命令同步到新AOF文件的过程。
4. 当 Redis 服务器重启时，可以加载 AOF 文件进行数据恢复。

优点：

1. AOF可以更好的保护数据不丢失，可以配置 AOF 每秒执行一次`fsync`操作，如果Redis进程挂掉，最多丢失1秒的数据。

2. AOF以`append-only`的模式写入，所以没有磁盘寻址的开销，写入性能非常高。

缺点：

1. 对于同一份文件AOF文件比RDB数据快照要大。
2. 数据恢复比较慢。

13. RDB和AOF如何选择?

通常来说，应该同时使用两种持久化方案，以保证数据安全。

- 如果数据不敏感，且可以从其他地方重新生成，可以关闭持久化。
- 如果数据比较重要，且能够承受几分钟的数据丢失，比如缓存等，只需要使用RDB即可。
- 如果是用做内存数据，要使用Redis的持久化，建议是RDB和AOF都开启。
- 如果只用AOF，优先使用everysec的配置选择，因为它在可靠性和性能之间取了一个平衡。

当RDB与AOF两种方式都开启时，Redis会优先使用AOF恢复数据，因为AOF保存的文件比RDB文件更完整。

14. Redis常见的部署方式有哪些?

Redis的几种常见使用方式包括：

- 单机版
- Redis主从
- Redis Sentinel（哨兵）
- Redis Cluster

使用场景：

单机版：很少使用。存在的问题：1、内存容量有限 2、处理能力有限 3、无法高可用。

主从模式：master 节点挂掉后，需要手动指定新的 master，可用性不高，基本不用。

哨兵模式：master 节点挂掉后，哨兵进程会主动选举新的 master，可用性高，但是每个节点存储的数据是一样的，浪费内存空间。数据量不是很多，集群规模不是很大，需要自动容错容灾的时候使用。

Redis cluster：主要是针对海量数据+高并发+高可用的场景，如果是海量数据，如果你的数据量很大，那么建议就用Redis cluster，所有主节点的容量总和就是Redis cluster可缓存的数据容量。

15. 主从复制

Redis的复制功能是支持多个数据库之间的数据同步。主数据库可以进行读写操作，当主数据库的数据发生变化时会自动将数据同步到从数据库。从数据库一般是只读的，它会接收主数据库同步过来的数据。一个主数据库可以有多个从数据库，而一个从数据库只能有一个主数据库。

```
redis-server //启动Redis实例作为主数据库
redis-server --port 6380 --slaveof  127.0.0.1 6379  //启动另一个实例作为从数据库
slaveof 127.0.0.1 6379
SLAVEOF NO ONE //停止接收其他数据库的同步并转化为主数据库。
```

主从复制的原理?

1. 当启动一个从节点时，它会发送一个 `PSYNC` 命令给主节点；
2. 如果是从节点初次连接到主节点，那么会触发一次全量复制。此时主节点会启动一个后台线程，开始生成一份 `RDB` 快照文件；
3. 同时还会将从客户端 client 新收到的所有写命令缓存在内存中。`RDB` 文件生成完毕后， 主节点会将`RDB`文件发送给从节点，从节点会先将`RDB`文件**写入本地磁盘，然后再从本地磁盘加载到内存**

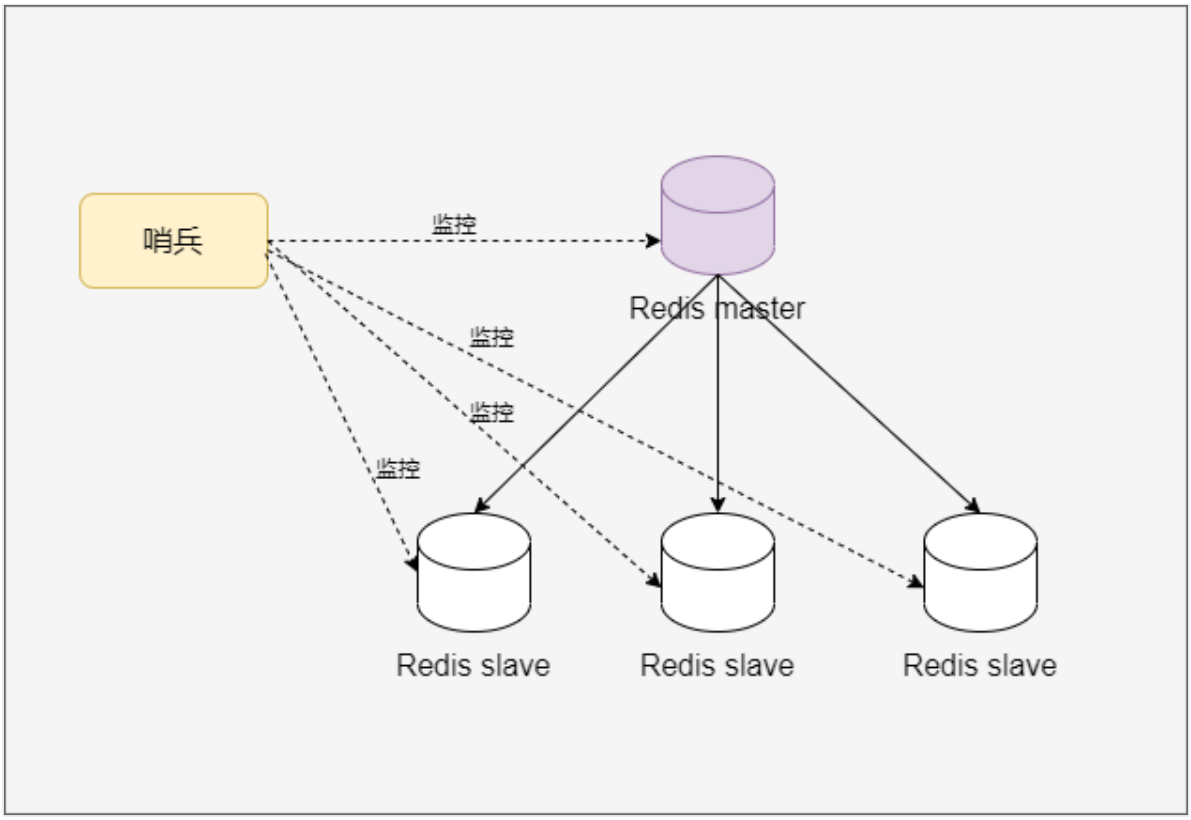
中；

4. 接着主节点会将内存中缓存的写命令发送到从节点，从节点同步这些数据；
5. 如果从节点跟主节点之间网络出现故障，连接断开了，会自动重连，连接之后主节点仅会将部分缺失的数据同步给从节点。

16. 哨兵Sentinel

主从复制存在不能自动故障转移、达不到高可用的问题。哨兵模式解决了这些问题。通过哨兵机制可以自动切换主从节点。

客户端连接Redis的时候，先连接哨兵，哨兵会告诉客户端Redis主节点的地址，然后客户端连接上Redis并进行后续的操作。当主节点宕机的时候，哨兵监测到主节点宕机，会重新推选出某个表现良好的从节点成为新的主节点，然后通过发布订阅模式通知其他的从服务器，让它们切换主机。



工作原理

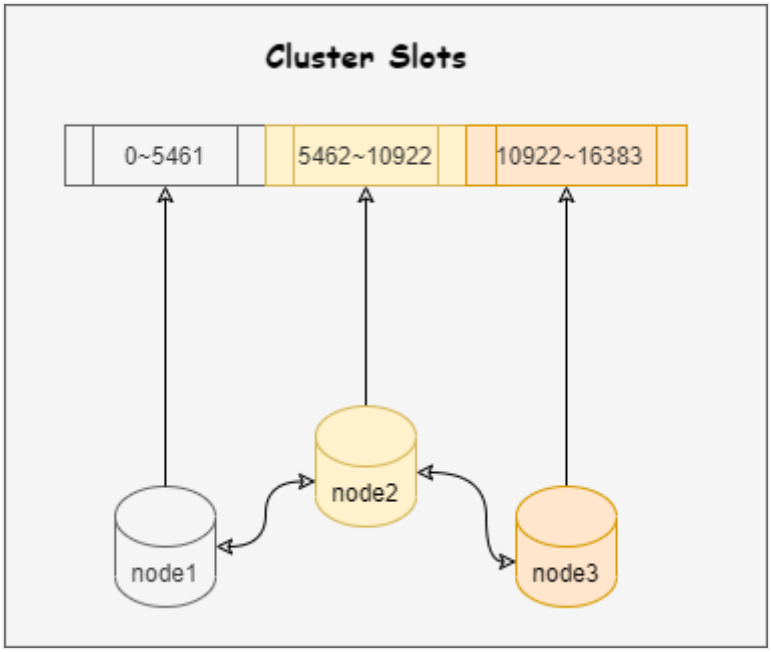
- 每个`Sentinel`以每秒钟一次的频率向它所知道的`Master`，`Slave`以及其他`Sentinel`实例发送一个`PING`命令。
- 如果一个实例距离最后一次有效回复`PING`命令的时间超过指定值， 则这个实例会被`Sentine`标记为主观下线。
- 如果一个`Master`被标记为主观下线，则正在监视这个`Master`的所有`Sentinel`要以每秒一次的频率确认`Master`是否真正进入主观下线状态。
- 当有足够数量的`Sentinel`（大于等于配置文件指定值）在指定的时间范围内确认`Master`的确进入了主观下线状态， 则`Master`会被标记为客观下线 。若没有足够数量的`Sentinel`同意`Master`已经下线， `Master`的客观下线状态就会被解除。 若`Master`重新向`Sentinel`的`PING`命令返回有效回复， `Master`的主观下线状态就会被移除。
- 哨兵节点会选举出哨兵 leader，负责故障转移的工作。
- 哨兵 leader 会推选出某个表现良好的从节点成为新的主节点，然后通知其他从节点更新主节点信息。

17. Redis cluster

哨兵模式解决了主从复制不能自动故障转移、达不到高可用的问题，但还是存在主节点的写能力、容量受限于单机配置的问题。而cluster模式实现了Redis的分布式存储，每个节点存储不同的内容，解决主节点的写能力、容量受限于单机配置的问题。

Redis cluster集群节点最小配置6个节点以上（3主3从），其中主节点提供读写操作，从节点作为备用节点，不提供请求，只作为故障转移使用。

Redis cluster采用**虚拟槽分区**，所有的键根据哈希函数映射到0 ~ 16383个整数槽内，每个节点负责维护一部分槽以及槽所映射的键值数据。



哈希槽是如何映射到 Redis 实例上的?

1. 对键值对的`key`使用`crc16`算法计算一个结果
2. 将结果对 16384 取余，得到的值表示`key`对应的哈希槽
3. 根据该槽信息定位到对应的实例

优点:

- 无中心架构，**支持动态扩**容;
- 数据按照`slot`存储分布在多个节点，节点间数据共享，**可动态调整数据分布**;
- **高可用性。**部分节点不可用时，集群仍可用。集群模式能够实现自动故障转移（failover），节点之间通过`gossip`协议交换状态信息，用投票机制完成`Slave`到`Master`的角色转换。

缺点:

- **不支持批量操作**（pipeline）。
- 数据通过异步复制，**不保证数据的强一致性。**
- **事务操作支持有限**，只支持多`key`在同一节点上的事务操作，当多个`key`分布于不同的节点上时无法使用事务功能。
- `key`作为数据分区的最小粒度，不能将一个很大的键值对象如`hash`、`list`等映射到不同的节点。
- **不支持多数据库空间**，单机下的Redis可以支持到16个数据库，集群模式下只能使用1个数据库空间。

17.1. 哈希分区算法有哪些?

节点取余分区。使用特定的数据，如Redis的键或用户ID，对节点数量N取余：hash（key）%N计算出哈希值，用来决定数据映射到哪一个节点上。 优点是简单性。扩容时通常采用翻倍扩容，避免数据映射全部被打乱导致全量迁移的情况。

一致性哈希分区。为系统中每个节点分配一个token，范围一般在0~232，这些token构成一个哈希环。数据读写执行节点查找操作时，先根据key计算hash值，然后顺时针找到第一个大于等于该哈希值的token节点。 这种方式相比节点取余最大的好处在于加入和删除节点只影响哈希环中相邻的节点，对其他节点无影响。

虚拟槽分区，所有的键根据哈希函数映射到0~16383整数槽内，计算公式：slot=CRC16（key）&16383。每一个节点负责维护一部分槽以及槽所映射的键值数据。**Redis Cluser采用虚拟槽分区算法。**

18. 过期键的删除策略?

1、**被动删除。**在访问key时，如果发现key已经过期，那么会将key删除。

2、**主动删除。**定时清理key，每次清理会依次遍历所有DB，从db随机取出20个key，如果过期就删除，如果其中有5个key过期，那么就继续对这个db进行清理，否则开始清理下一个db。

3、**内存不够时清理。**Redis有最大内存的限制，通过maxmemory参数可以设置最大内存，当使用的内存超过了设置的最大内存，就要进行内存释放， 在进行内存释放的时候，会按照配置的淘汰策略清理内存。

19. 内存淘汰策略有哪些?

当Redis的内存超过最大允许的内存之后，Redis 会触发内存淘汰策略，删除一些不常用的数据，以保证Redis服务器正常运行。

Redisv4.0前提供 6 种数据淘汰策略:

- **volatile-lru**：LRU（`Least Recently Used`），最近使用。利用LRU算法移除设置了过期时间的key
- **allkeys-lru**：当内存不足以容纳新写入数据时，从数据集中移除最近最少使用的key
- **volatile-ttl**：从已设置过期时间的数据集中挑选将要过期的数据淘汰
- **volatile-random**：从已设置过期时间的数据集中任意选择数据淘汰
- **allkeys-random**：从数据集中任意选择数据淘汰
- **no-eviction**：禁止删除数据，当内存不足以容纳新写入数据时，新写入操作会报错

Redisv4.0后增加以下两种:

- **volatile-lfu**：LFU，Least Frequently Used，最少使用，从已设置过期时间的数据集中挑选最不经常使用的数据淘汰。
- **allkeys-lfu**：当内存不足以容纳新写入数据时，从数据集中移除最不经常使用的key。

内存淘汰策略可以通过配置文件来修改，相应的配置项是`maxmemory-policy`，默认配置是`noeviction`。

20. 如何保证缓存与数据库双写时的数据一致性?

1、先删除缓存再更新数据库

进行更新操作时，先删除缓存，然后更新数据库，后续的请求再次读取时，会从数据库读取后再将新数据更新到缓存。

存在的问题：删除缓存数据之后，更新数据库完成之前，这个时间段内如果有新的读请求过来，就会从数据库读取旧数据重新写到缓存中，再次造成不一致，并且后续读的都是旧数据。

2、先更新数据库再删除缓存

进行更新操作时，先更新MySQL，成功之后，删除缓存，后续读取请求时再将新数据回写缓存。

存在的问题：更新MySQL和删除缓存这段时间内，请求读取的还是缓存的旧数据，不过等数据库更新完成，就会恢复一致，影响相对比较小。

3、异步更新缓存

数据库的更新操作完成后不直接操作缓存，而是把这个操作命令封装成消息扔到消息队列中，然后由Redis自己去消费更新数据，消息队列可以保证数据操作顺序一致性，确保缓存系统的数据正常。

21. 缓存穿透

缓存穿透是指查询一个**不存在的数据**，由于缓存是不命中时被动写的，如果从DB查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到DB去查询，失去了缓存的意义。在流量大时，可能DB就挂掉了。

1. **缓存空值**，不会查数据库。
2. 采用**布隆过滤器**，将所有可能存在的数据哈希到一个足够大的`bitmap`中，查询不存在的数据会被这个`bitmap`拦截掉，从而避免了对`DB`的查询压力。

布隆过滤器的原理：当一个元素被加入集合时，通过K个散列函数将这个元素映射成一个位数组中的K个点，把它们置为1。查询时，将元素通过散列函数映射之后会得到k个点，如果这些点有任何一个0，则被检元素一定不在，直接返回；如果都是1，则查询元素很可能存在，就会去查询Redis和数据库。

22. 缓存雪崩

缓存雪崩是指在我们设置缓存时采用了相同的过期时间，**导致缓存在某一时刻同时失效**，请求全部转发到DB，DB瞬时压力过重挂掉。

解决方法：在原有的失效时间基础上**增加一个随机值**，使得过期时间分散一些。

23. 缓存击穿

缓存击穿：大量的请求同时查询一个 key 时，此时这个 key 正好失效了，就会导致大量的请求都落到数据库。**缓存击穿是查询缓存中失效的 key，而缓存穿透是查询不存在的 key。**

解决方法：加分布式锁，第一个请求的线程可以拿到锁，拿到锁的线程查询到了数据之后设置缓存，其他的线程获取锁失败会等待50ms然后重新到缓存取数据，这样便可以避免大量的请求落到数据库。

```
public String get(String key) {
    String value = redis.get(key);
    if (value == null) { //缓存值过期
        String unique_key = systemId + ":" + key;
        //设置30s的超时
        if (redis.set(unique_key, 1, 'NX', 'PX', 30000) == 1) {  //设置成功
            value = db.get(key);
            redis.set(key, value, expire_secs);
            redis.del(unique_key);
        } else {  //其他线程已经到数据库取值并回写到缓存了，可以重试获取缓存值
            sleep(50);
            get(key);  //重试
        }
    } else {
```

```
            return value;
        }
    }
```

24. Redis 怎么实现消息队列?

使用一个列表，让生产者将任务使用LPUSH命令放进列表，消费者不断用RPOP从列表取出任务。

> BRPOP和RPOP命令相似，唯一的区别就是当列表没有元素时BRPOP命令会一直阻塞连接，直到有新元素加入。

```
BRPOP queue 0  //0表示不限制等待时间
```

优先级队列

如果多个键都有元素，则按照从左到右的顺序取元素。

```
BLPOP queue:1 queue:2 queue:3 0
```

发布/订阅模式

`PSUBSCRIBE channel?*` 按照规则订阅。 `PUNSUBSCRIBE channel?*` 退订通过PSUBSCRIBE命令按照某种规则订阅的频道。其中订阅规则要进行严格的字符串匹配，`PUNSUBSCRIBE *` 无法退订 `channel?*` 规则。

```
PUBLISH channel1 hi
SUBSCRIBE channel1
UNSUBSCRIBE channel1 //退订通过SUBSCRIBE命令订阅的频道。
```

缺点：在消费者下线的情况下，生产的消息会丢失。

延时队列

使用sortedset，拿时间戳作为score，消息内容作为key，调用zadd来生产消息，消费者用 `zrangebyscore` 指令获取N秒之前的数据轮询进行处理。

25. pipeline的作用?

redis客户端执行一条命令分4个过程： 发送命令、命令排队、命令执行、返回结果。使用 `pipeline` 可以批量请求，批量返回结果，执行速度比逐条执行要快。

使用 `pipeline` 组装的命令个数不能太多，不然数据量过大，增加客户端的等待时间，还可能造成网络阻塞，可以将大量命令的拆分多个小的 `pipeline` 命令完成。

原生批命令（mset和mget）与 `pipeline` 对比：

1. 原生批命令是原子性， `pipeline` 是**非原子性。**pipeline命令中途异常退出，之前执行成功的命令**不会回滚。**
2. 原生批命令只有一个命令，但 `pipeline` **支持多命令。**

26. LUA脚本

Redis 通过 LUA 脚本创建具有原子性的命令： 当lua脚本命令正在运行的时候，不会有其他脚本或 Redis 命令被执行，实现组合命令的原子操作。

在Redis中执行Lua脚本有两种方法：`eval`和`evalsha`。`eval`命令使用内置的 Lua 解释器，对 Lua 脚本进行求值。

```
//第一个参数是lua脚本，第二个参数是键名参数个数，剩下的是键名参数和附加参数> eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second1) "key1"2) "key2"3) "first"4) "second"
```

lua脚本作用

1、Lua脚本在Redis中是原子执行的，执行过程中间不会插入其他命令。

2、Lua脚本可以将多条命令一次性打包，有效地减少网络开销。

应用场景

举例：限制接口访问频率。

在Redis维护一个接口访问次数的键值对，`key`是接口名称，`value`是访问次数。每次访问接口时，会执行以下操作：

- 通过`aop`拦截接口的请求，对接口请求进行计数，每次进来一个请求，相应的接口访问次数`count`加1，存入redis。
- 如果是第一次请求，则会设置`count=1`，并设置过期时间。因为这里`set()`和`expire()`组合操作不是原子操作，所以引入`lua`脚本，实现原子操作，避免并发访问问题。
- 如果给定时间范围内超过最大访问次数，则会抛出异常。

```
private String buildLuaScript() {
    return "local c" +
        "\nc = redis.call('get',KEYS[1])" +
        "\nif c and tonumber(c) > tonumber(ARGV[1]) then" +
        "\nreturn c;" +
        "\nend" +
        "\nc = redis.call('incr',KEYS[1])" +
        "\nif tonumber(c) == 1 then" +
        "\nredis.call('expire',KEYS[1],ARGV[2])" +
        "\nend" +
        "\nreturn c;";
}

String luaScript = buildLuaScript();
RedisScript<Number> redisScript = new DefaultRedisScript<>(luaScript, Number.class);
Number count = redisTemplate.execute(redisScript, keys, limit.count(), limit.period());
```

PS：这种接口限流的实现方式比较简单，问题也比较多，一般不会使用，接口限流用的比较多的是令牌桶算法和漏桶算法。

27. 什么是RedLock?

Redis 官方站提出了一种权威的基于 Redis 实现分布式锁的方式名叫 *Redlock*，此种方式比原先的单节点的方法更安全。它可以保证以下特性：

1. 安全特性：互斥访问，即永远只有一个 client 能拿到锁
2. 避免死锁：最终 client 都可能拿到锁，不会出现死锁的情况，即使原本锁住某资源的 client 挂掉了
3. 容错性：只要大部分 Redis 节点存活就可以正常提供服务