

基础篇

1、Java语言有哪些特点

- 1、简单易学、有丰富的类库
- 2、面向对象（Java最重要的特性，让程序耦合度更低，内聚性更高）
- 3、与平台无关性（JVM是Java跨平台使用的根本）
- 4、可靠安全
- 5、支持多线程



关注【程序员生活志】
免费获取更多海量资源

2、面向对象和面向过程的区别

面向过程：是分析解决问题的步骤，然后用函数把这些步骤一步一步地实现，然后在使用的时候一一调用则可。性能较高，所以单片机、嵌入式开发等一般采用面向过程开发

面向对象：是把构成问题的事务分解成各个对象，而建立对象的目的也不是为了完成一个个步骤，而是为了描述某个事物在解决整个问题的过程中所发生的行为。面向对象有封装、继承、多态的特性，所以易维护、易复用、易扩展。可以设计出低耦合的系统。但是性能上来说，比面向过程要低。

3、八种基本数据类型的大小，以及他们的封装类

基本类型	大小（字节）	默认值	封装类
byte	1	(byte)0	Byte
short	2	(short)0	Short
int	4	0	Integer
long	8	0L	Long
float	4	0.0f	Float
double	8	0.0d	Double
boolean		false	Boolean
char	2	\u0000(null)	Character

注：

1.int是基本数据类型，Integer是int的封装类，是引用类型。int默认值是0，而Integer默认值是null，所以Integer能区分出0和null的情况。一旦java看到null，就知道这个引用还没有指向某个对象，再任何引用使用前，必须为其指定一个对象，否则会报错。

2. 基本数据类型在声明时系统会自动给它分配空间，而引用类型声明时只是分配了引用空间，必须通过实例化开辟数据空间之后才可以赋值。数组对象也是一个引用对象，将一个数组赋值给另一个数组时只是复制了一个引用，所以通过某一个数组所做的修改在另一个数组中也看的见。

虽然定义了boolean这种数据类型，但是只对它提供了非常有限的支持。在Java虚拟机中没有任何供boolean值专用的字节码指令，Java语言表达式所操作的boolean值，在编译之后都使用Java虚拟机中的int数据类型来代替，而boolean数组将会被编码成Java虚拟机的byte数组，每个元素boolean元素占8位。这样我们可以得出boolean类型占了单独使用是4个字节，在数组中又是1个字节。使用int的原因是，对于当下32位的处理器（CPU）来说，一次处理数据是32位（这里不是指的是32/64位系统，而是指CPU硬件层面），具有高效存取的特点。

4、标识符的命名规则。

标识符的含义：

是指在程序中，我们自己定义的内容，譬如，类的名字，方法名称以及变量名称等等，都是标识符。

命名规则：（硬性要求）

标识符可以包含英文字母，0-9的数字，\$以及_

标识符不能以数字开头

标识符不是关键字

命名规范：（非硬性要求）

类名规范：首字符大写，后面每个单词首字母大写（大驼峰式）。

变量名规范：首字母小写，后面每个单词首字母大写（小驼峰式）。

方法名规范：同变量名。

5、 instanceof 关键字的作用

instanceof 严格来说是Java中的一个双目运算符，用来测试一个对象是否为一个类的实例，用法为：

```
boolean result = obj instanceof Class
```

其中 obj 为一个对象，Class 表示一个类或者一个接口，当 obj 为 Class 的对象，或者是其直接或间接子类，或者是其接口的实现类，结果result 都返回 true，否则返回false。

注意：编译器会检查 obj 是否能转换成右边的class类型，如果不能转换则直接报错，如果不能确定类型，则通过编译，具体看运行时定。

```
int i = 0;  
System.out.println(i instanceof Integer); //编译不通过 i必须是引用类型，不能是基本类型  
System.out.println(i instanceof Object); //编译不通过
```

```
Integer integer = new Integer(1);  
System.out.println(integer instanceof Integer); //true
```

//false，在 JavaSE 规范 中对 instanceof 运算符的规定就是：如果 obj 为 null，那么将返回 false。
System.out.println(null instanceof Object);

6、Java自动装箱与拆箱

装箱就是自动将基本数据类型转换为包装器类型 (`int-->Integer`) ; 调用方法: `Integer`的 `valueOf(int)` 方法

拆箱就是自动将包装器类型转换为基本数据类型 (`Integer-->int`) 。调用方法: `Integer`的 `intValue`方法

在Java SE5之前, 如果要生成一个数值为10的`Integer`对象, 必须这样进行:

```
Integer i = new Integer(10);
```

而在从Java SE5开始就提供了自动装箱的特性, 如果要生成一个数值为10的`Integer`对象, 只需要这样就可以了:

```
Integer i = 10;
```

面试题1: 以下代码会输出什么?

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

        Integer i1 = 100;
        Integer i2 = 100;
        Integer i3 = 200;
        Integer i4 = 200;

        System.out.println(i1==i2);
        System.out.println(i3==i4);
    }
}
```

运行结果:

```
true  
false
```

为什么会出现这样的结果? 输出结果表明i1和i2指向的是同一个对象, 而i3和i4指向的是不同的对象。此时只需一看源码便知究竟, 下面这段代码是`Integer`的`valueOf`方法的具体实现:

```
public static Integer valueOf(int i) {
    if(i >= -128 && i <= IntegerCache.high)
        return IntegerCache.cache[i + 128];
    else
        return new Integer(i);
}
```

其中IntegerCache类的实现为：

```
private static class IntegerCache
    { static final int high;
    static final Integer cache[];

    static {
        final int low = -128;

        // high value may be configured by property
        int h = 127;
        if (integerCacheHighPropValue != null) {
            // Use Long.decode here to avoid invoking methods that
            // require Integer's autoboxing cache to be initialized
            int i = Long.decode(integerCacheHighPropValue).intValue();
            i = Math.max(i, 127);
            // Maximum array size is Integer.MAX_VALUE
            h = Math.min(i, Integer.MAX_VALUE - -low);
        }
        high = h;

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

    private IntegerCache() {}
}
```

从这2段代码可以看出，在通过valueOf方法创建Integer对象的时候，如果数值在[-128,127]之间，便返回指向IntegerCache.cache中已经存在的对象的引用；否则创建一个新的Integer对象。

上面的代码中i1和i2的数值为100，因此会直接从cache中取已经存在的对象，所以i1和i2指向的是同一个对象，而i3和i4则是分别指向不同的对象。

面试题2：以下代码输出什么？

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

        Double i1 = 100.0;
        Double i2 = 100.0;
        Double i3 = 200.0;
        Double i4 = 200.0;

        System.out.println(i1==i2);
        System.out.println(i3==i4);
    }
}
```

运行结果：

```
false  
false
```

原因： 在某个范围内的整型数值的个数是有限的，而浮点数却不是。

7、重载和重写的区别

重写(Override)

从字面上看，重写就是 重新写一遍的意思。其实就是在子类中把父类本身有的方法重新写一遍。子类继承了父类原有的方法，但有时子类并不想原封不动的继承父类中的某个方法，所以在方法名，参数列表，返回类型(除过子类中方法的返回值是父类中方法返回值的子类时)都相同的情况下， 对方法体进行修改或重写，这就是重写。但要注意子类函数的访问修饰权限不能少于父类的。

```
public class Father {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Son s = new Son();  
        s.sayHello();  
    }  
  
    public void sayHello()  
    { System.out.println("Hello"  
    );  
    }  
}  
  
class Son extends  
  
Father{ @Override  
public void sayHello() {  
    // TODO Auto-generated method stub  
    System.out.println("Hello by ");  
}  
  
}
```

重写 总结：

- 1.发生在父类与子类之间
- 2.方法名，参数列表，返回类型（除过子类中方法的返回类型是父类中返回类型的子类）必须相同
- 3.访问修饰符的限制一定要大于被重写方法的访问修饰符（public>protected>default>private）
- 4.重写方法一定不能抛出新的检查异常或者比被重写方法申明更加宽泛的检查型异常

重载 (Overload)

在一个类中，同名的方法如果有不同的参数列表（参数类型不同、参数个数不同甚至是参数顺序不同）则视为重载。同时，重载对返回类型没有要求，可以相同也可以不同，但不能通过返回类型是否相同来判断重载。

```
public class Father {
```

```

public static void main(String[] args) {
    // TODO Auto-generated method stub
    Father s = new Father();
    s.sayHello();
    s.sayHello("wintershi");
}

public void sayHello()
{
    System.out.println("Hello");
}

public void sayHello(String name)
{
    System.out.println("Hello" + " " +
}

```

重载 总结：

- 1.重载Overload是一个类中多态性的一种表现
- 2.重载要求同名方法的参数列表不同(参数类型, 参数个数甚至是参数顺序)
- 3.重载的时候, 返回值类型可以相同也可以不相同。无法以返回型别作为重载函数的区分标准

8、 equals与==的区别

`==` :

`==` 比较的是变量(栈)内存中存放的对象的(堆)内存地址, 用来判断两个对象的地址是否相同, 即是否是指同一个对象。比较的是真正意义上的指针操作。

- 1、比较的是操作符两端的操作数是否是同一个对象。
 - 2、两边的操作数必须是同一类型的 (可以是父子类之间) 才能编译通过。
 - 3、比较的是地址, 如果是具体的阿拉伯数字的比较, 值相等则为true, 如:
- `int a=10` 与 `long b=10L` 与 `double c=10.0`都是相同的 (为true) , 因为他们都指向地址为10的堆。

`equals`:

`equals`用来比较的是两个对象的内容是否相等, 由于所有的类都是继承自`java.lang.Object`类的, 所以适用于所有对象。如果没有对该方法进行覆盖的话, 调用的仍然是`Object`类中的方法, 而`Object`中的`equals`方法返回的却是`==`的判断。

总结:

所有比较是否相等时, 都是用`equals` 并且在对常量相比较时, 把常量写在前面, 因为使用`object.equals` `object`可能为`null` 则空指针

在阿里的代码规范中只使用`equals` , 阿里插件默认会识别, 并可以快速修改, 推荐安装阿里插件来排查老代码使用 “`==`” , 替换成`equals`

9、 Hashcode的作用

java的集合有两类，一类是List，还有一类是Set。前者有序可重复，后者无序不重复。当我们在set中插入的时候怎么判断是否已经存在该元素呢，可以通过equals方法。但是如果元素太多，用这样的方法就会比较慢。

于是有人发明了哈希算法来提高集合中查找元素的效率。这种方式将集合分成若干个存储区域，每个对象可以计算出一个哈希码，可以将哈希码分组，每组分别对应某个存储区域，根据一个对象的哈希码就可以确定该对象应该存储的那个区域。

hashCode方法可以这样理解：它返回的就是根据对象的内存地址换算出的一个值。这样一来，当集合要添加新的元素时，先调用这个元素的hashCode方法，就一下子能定位到它应该放置的物理位置上。如果这个位置上没有元素，它就可以直接存储在这个位置上，不用再进行任何比较了；如果这个位置上已经有元素了，就调用它的equals方法与新元素进行比较，相同的话就不存了，不相同就散列其它的地址。这样一来实际调用equals方法的次数就大大降低了，几乎只需要一两次。

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

String是只读字符串，它并不是基本数据类型，而是一个对象。从底层源码来看是一个final类型的字符数组，所引用的字符串不能被改变，一经定义，无法再增删改。每次对String的操作都会生成新的 String 对象。

```
private final char value[];
```

每次+操作：隐式在堆上new了一个跟原字符串相同的StringBuilder对象，再调用append方法 拼接+后面的字符。

StringBuffer和StringBuilder他们两都继承了AbstractStringBuilder抽象类，从AbstractStringBuilder抽象类中我们可以看到

```
/**  
 * The value is used for character storage.  
 */  
char[] value;
```

他们的底层都是可变的字符数组，所以在进行频繁的字符串操作时，建议使用StringBuffer和StringBuilder来进行操作。另外StringBuffer 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder 并没有对方法进行加同步锁，所以是非线程安全的。

11、ArrayList和linkedlist的区别

Array（数组）是基于索引(index)的数据结构，它使用索引在数组中搜索和读取数据是很快的。

Array获取数据的时间复杂度是O(1),但是要删除数据却是开销很大，因为这需要重排数组中的所有数据，(因为删除数据以后，需要把后面所有的数据前移)

缺点：数组初始化必须指定初始化的长度，否则报错

例如：

```
int[] a = new int[4]; //推介使用 int[] 这种方式初始化  
int c[] = {23, 43, 56, 78}; //长度: 4, 索引范围: [0, 3]
```

List—是一个有序的集合，可以包含重复的元素，提供了按索引访问的方式，它继承**Collection**。List有两个重要的实现类：**ArrayList**和**LinkedList**

ArrayList: 可以看作是能够自动增长容量的数组**ArrayList**的**toArray**方法返回一个数组 **ArrayList**的**asList**方法返回一个列表
ArrayList底层的实现是**Array**, 数组扩容实现

LinkedList是一个双链表，在添加和删除元素时具有比**ArrayList**更好的性能.但在**get**与**set**方面弱于**ArrayList**.当然,这些对比都是指数据量很大或者操作很频繁。

12、HashMap和HashTable的区别

1、两者父类不同

HashMap是继承自AbstractMap类，而Hashtable是继承自Dictionary类。不过它们都实现了同时实现了map、Cloneable（可复制）、Serializable（可序列化）这三个接口。

2、对外提供的接口不同

Hashtable比HashMap多提供了elements() 和contains() 两个方法。

elements() 方法继承自Hashtable的父类Dictionary。elements() 方法用于返回此Hashtable中的value的枚举。

contains()方法判断该Hashtable是否包含传入的value。它的作用与containsValue()一致。事实上，containsValue() 就只是调用了一下contains() 方法。

3、对null的支持不同

Hashtable: key和value都不能为null。

HashMap: key可以为null，但是这样的key只能有一个，因为必须保证key的唯一性；可以有多个key值对应的value为null。

4、安全性不同

HashMap是线程不安全的，在多线程并发的环境下，可能会产生死锁等问题，因此需要开发人员自己处理多线程的安全问题。

Hashtable是线程安全的，它的每个方法上都有synchronized关键字，因此可直接用于多线程中。

虽然HashMap是线程不安全的，但是它的效率远远高于Hashtable，这样设计是合理的，因为大部分的使用场景都是单线程。当需要多线程操作的时候可以使用线程安全的ConcurrentHashMap。

ConcurrentHashMap虽然也是线程安全的，但是它的效率比Hashtable要高好多倍。因为ConcurrentHashMap使用了分段锁，并不对整个数据进行锁定。

5、初始容量大小和每次扩充容量大小不同

6、计算hash值的方法不同

13、Collection包结构，与Collections的区别

Collection是集合类的上级接口，子接口有Set、List、LinkedList、ArrayList、Vector、Stack、Set；

Collections是集合类的一个帮助类，它包含有各种有关集合操作的静态多态方法，用于实现对各种集合的搜索、排序、线程安全化等操作。此类不能实例化，就像一个工具类，服务于Java的Collection框架。

14、Java的四种引用，强弱软虚

- 强引用

强引用是平常中使用最多的引用，强引用在程序内存不足（OOM）的时候也不会被回收，使用方式：

```
String str = new String("str");
```

- 软引用

软引用在程序内存不足时，会被回收，使用方式：

```
// 注意：wrf这个引用也是强引用，它是指向SoftReference这个对象的，  
// 这里的软引用指的是指向new String("str")的引用，也就是SoftReference类中T  
SoftReference<String> wrf = new SoftReference<String>(new String("str"));
```

可用场景：创建缓存的时候，创建的对象放进缓存中，当内存不足时，JVM就会回收早先创建的对象。

- 弱引用

弱引用就是只要JVM垃圾回收器发现了它，就会将之回收，使用方式：

```
WeakReference<String> wrf = new WeakReference<String>(str);
```

可用场景：Java源码中的java.util.WeakHashMap 中的key 就是使用弱引用，我的理解就是，一旦我不需要某个引用，JVM会自动帮我处理它，这样我就不需要做其它操作。

- 虚引用

虚引用的回收机制跟弱引用差不多，但是它被回收之前，会被放入ReferenceQueue 中。注意哦，其它引用是被JVM回收后才被传入ReferenceQueue 中的。由于这个机制，所以虚引用大多被用于引用销毁前的处理工作。还有就是，虚引用创建的时候，必须带有ReferenceQueue，使用例子：

```
PhantomReference<String> prf = new PhantomReference<String>(new  
String("str"), new ReferenceQueue<>());
```

可用场景：对象销毁前的一些操作，比如说资源释放等。`Object.finalize()` 虽然也可以做这类动作，但是这种方式即不安全又低效

上诉所说的几类引用，都是指对象本身的引用，而不是指 Reference 的四个子类的引用（SoftReference 等）。

15、泛型常用特点

泛型是Java SE 1.5之后的特性，《Java 核心技术》中对泛型的定义是：

“泛型” 意味着编写的代码可以被不同类型的对象所重用。

“泛型”，顾名思义，“泛指的类型”。我们提供了泛指的概念，但具体执行的时候却可以有具体的规则来约束，比如我们用的非常多的ArrayList就是个泛型类，ArrayList作为集合可以存放各种元素，如 Integer, String, 自定义的各种类型等，但在我们使用的时候通过具体的规则来约束，如我们可以约束集合中只存放Integer类型的元素，如

```
List<Integer> initData = new ArrayList<>()
```

使用泛型的好处？

以集合来举例，使用泛型的好处是我们不必因为添加元素类型的不同而定义不同类型的集合，如整型集合类，浮点型集合类，字符串集合类，我们可以定义一个集合来存放整型、浮点型，字符串型数据，而这并不是最重要的，因为我们只要把底层存储设置了Object即可，添加的数据全部都可向上转型为Object。更重要的是我们可以通过规则按照自己的想法控制存储的数据类型。

16、Java创建对象有几种方式？

java中提供了以下四种创建对象的方式：

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

17、有没有可能两个不相等的对象有相同的hashcode

有可能.在产生hash冲突时,两个不相等的对象就会有相同的 hashcode 值.当hash冲突产生时,一般有以下几种方式来处理:

- 拉链法:每个哈希表节点都有一个next指针,多个哈希表节点可以用next指针构成一个单向链表, 被分配到同一个索引上的多个节点可以用这个单向链表进行存储.
- 开放定址法:一旦发生了冲突,就去寻找下一个空的散列地址,只要散列表足够大,空的散列地址总能找到,并将记录存入
- 再哈希:又叫双哈希法,有多个不同的Hash函数.当发生冲突时,使用第二个,第三个....等哈希函数计算地址,直到无冲突.

18、深拷贝和浅拷贝的区别是什么？

- 浅拷贝:被复制对象的所有变量都含有与原来的对象相同的值,而所有的对其他对象的引用仍然指向原来的对象.换言之,浅拷贝仅仅复制所考虑的对象,而不复制它所引用的对象.
- 深拷贝:被复制对象的所有变量都含有与原来的对象相同的值.而那些引用其他对象的变量将指向被复制过的新对象.不再是原有的那些被引用的对象.换言之,深拷贝把要复制的对象所引用的对象都复制了一遍.

19、final有哪些用法？

final也是很多面试喜欢问的地方,但我觉得这个问题很无聊,通常能回答下以下5点就不错了:

- 被final修饰的类不可以被继承
- 被final修饰的方法不可以被重写
- 被final修饰的变量不可以被改变.如果修饰引用,那么表示引用不可变,引用指向的内容可变.
- 被final修饰的方法,JVM会尝试将其内联,以提高运行效率
- 被final修饰的常量,在编译阶段会存入常量池中.

除此之外,编译器对final域要遵守的两个重排序规则更好:

在构造函数内对一个final域的写入,与随后把这个被构造对象的引用赋值给一个引用变量,这两个操作之间不能重排序

初次读一个包含final域的对象的引用,与随后初次读这个final域,这两个操作之间不能重排序.

20、static都有哪些用法？

所有的人都知道static关键字这两个基本的用法:静态变量和静态方法.也就是被static所修饰的变量/方法都属于类的静态资源,类实例所共享.

除了静态变量和静态方法之外,static也用于静态块,多用于初始化操作:

```
public class PreCache{ static{  
    //执行相关操作  
}  
}
```

此外static也多用于修饰内部类,此时称之为静态内部类.

最后一种用法就是静态导包,即import static .import static是在JDK 1.5之后引入的新特性,可以用来指定导入某个类中的静态资源,并且不需要使用类名,可以直接使用资源名,比如:

```
import static java.lang.Math.*;  
  
public class Test{  
  
    public static void main(String[] args){  
        //System.out.println(Math.sin(20));传统做法  
        System.out.println(sin(20));  
    }  
}
```

21、3*0.1 == 0.3返回值是什么

false,因为有些浮点数不能完全精确的表示出来.

22、a=a+b与a+=b有什么区别吗?

+ = 操作符会进行隐式自动类型转换,此处a+=b隐式的将加操作的结果类型强制转换为持有结果的类型,而a=a+b则不会自动进行类型转换.如:

```
byte a = 127;
byte b = 127;
b = a + b; // 报编译错误:cannot convert from int to byte
b += a;
```

以下代码是否有错,有的话怎么改?

```
short s1= 1;
s1 = s1 + 1;
```

有错误.short类型在进行运算时会自动提升为int类型,也就是说 s1+1 的运算结果是int类型,而s1是short类型,此时编译器会报错.

正确写法:

```
short s1= 1;
s1 += 1;
```

+ =操作符会对右边的表达式结果强转匹配左边的数据类型,所以没错.

23、try catch finally, try里有return, finally还执行么?

执行, 并且finally的执行早于try里面的return

结论:

- 1、不管有木有出现异常, finally块中代码都会执行;
- 2、当try和catch中有return时, finally仍然会执行;
- 3、finally是在return后面的表达式运算后执行的 (此时并没有返回运算后的值, 而是先把要返回的值保存起来, 管finally中的代码怎么样, 返回的值都不会改变, 任然是之前保存的值), 所以函数返回值是 在finally执行前确定的;
- 4、finally中最好不要包含return, 否则程序会提前退出, 返回值不是try或catch中保存的返回值。

24、Exception与Error包结构

Java可抛出(Throwable)的结构分为三种类型: 被检查的异常(CheckedException), 运行时异常(RuntimeException), 错误(Error)。

1、运行时异常

定义:RuntimeException及其子类都被称为运行时异常。

特点:Java编译器不会检查它。也就是说, 当程序中可能出现这类异常时, 倘若既"没有通过throws声明抛出它", 也"没有用try-catch语句捕获它", 还是会编译通过。例如, 除数为零时产生的ArithmaticException异常, 数组越界时产生的IndexOutOfBoundsException异常, fail-fast机制产生的ConcurrentModificationException异常 (java.util包下面的所有集合类都是快速失败的, “快速失败” 也就是fail-fast, 它是Java集合的一种错误检测机制。当多个线程对集合进行结构上的改变的操作时, 有可能会产生fail-fast机制。记住是有可能, 而不是一定。例如: 假设存在两个线程 (线程1、线程2), 线程1通过Iterator在遍历集合A中的元素, 在某个时候线程2修改了集合A的结构 (是结构上面的修改, 而不是简单的修改集合元素的内容), 那么这个时候程序就会抛出

ConcurrentModificationException 异常，从而产生fail-fast机制，这个错叫并发修改异常。Fail-safe，java.util.concurrent包下面的所有的类都是安全失败的，在遍历过程中，如果已经遍历的数组上的内容变化了，迭代器不会抛出ConcurrentModificationException异常。如果未遍历的数组上的内容发生了变化，则有可能反映到迭代过程中。这就是ConcurrentHashMap迭代器弱一致的表现。

ConcurrentHashMap的弱一致性主要是为了提升效率，是一致性与效率之间的一种权衡。要成为强一致性，就得到处使用锁，甚至是全局锁，这就与Hashtable和同步的HashMap一样了。）等，都属于运行时异常。

常见的五种运行时异常：

ClassCastException（类转换异常）

IndexOutOfBoundsException（数组越界）

NullPointerException（空指针异常）

ArrayStoreException（数据存储异常，操作数组是类型不一致） BufferOverflowException

2、被检查异常

定义：Exception类本身，以及Exception的子类中除了“运行时异常”之外的其它子类都属于被检查异常。

特点：Java编译器会检查它。此类异常，要么通过throws进行声明抛出，要么通过try-catch进行捕获处理，否则不能通过编译。例如，CloneNotSupportedException就属于被检查异常。当通过clone()接口去克隆一个对象，而该对象对应的类没有实现Cloneable接口，就会抛出CloneNotSupportedException异常。被检查异常通常都是可以恢复的。

如：

IOException

FileNotFoundException

SQLException

被检查的异常适用于那些不是因程序引起的错误情况，比如：读取文件时文件不存在引发的FileNotFoundException。然而，不被检查的异常通常都是由于糟糕的编程引起的，比如：在对象引用时没有确保对象非空而引起的NullPointerException。

3、错误

定义：Error类及其子类。

特点：和运行时异常一样，编译器也不会对错误进行检查。

当资源不足、约束失败、或是其它程序无法继续运行的条件发生时，就产生错误。程序本身无法修复这些错误的。例如，VirtualMachineError就属于错误。出现这种错误会导致程序终止运行。

OutOfMemoryError、ThreadDeath。

Java虚拟机规范规定JVM的内存分为了好几块，比如堆，栈，程序计数器，方法区等

25、OOM你遇到过哪些情况，SOF你遇到过哪些情况

OOM：

1，OutOfMemoryError异常

除了程序计数器外，虚拟机内存的其他几个运行时区域都有可能发生OutOfMemoryError(OOM)异常的可能。

Java Heap 溢出：

一般的异常信息：java.lang.OutOfMemoryError:Java heap space.

java堆用于存储对象实例，我们只要不断的创建对象，并且保证GC Roots到对象之间有可达路径来避免垃圾回收机制清除这些对象，就会在对象数量达到最大堆容量限制后产生内存溢出异常。

出现这种异常，一般手段是先通过内存映像分析工具(如Eclipse Memory Analyzer)对dump出来的堆转存快照进行分析，重点是确认内存中的对象是否是必要的，先分清是因为内存泄漏(Memory Leak)还是内存溢出(Memory Overflow)。

如果是内存泄漏，可进一步通过工具查看泄漏对象到GC Roots的引用链。于是就能找到泄漏对象是通过怎样的路径与GC Roots相关联并导致垃圾收集器无法自动回收。

如果不存在泄漏，那就应该检查虚拟机的参数(-Xmx与-Xms)的设置是否适当。

2, 虚拟机栈和本地方法栈溢出

如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError异常。

如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出OutOfMemoryError异常

这里需要注意当栈的大小越大可分配的线程数就越少。

3, 运行时常量池溢出

异常信息：java.lang.OutOfMemoryError:PermGenspace

如果要向运行时常量池中添加内容，最简单的做法就是使用String.intern()这个Native方法。该方法的作用是：如果池中已经包含一个等于此String的字符串，则返回代表池中这个字符串的String对象；否则，将此String对象包含的字符串添加到常量池中，并且返回此String对象的引用。由于常量池分配在方法区内，我们可以通过-XX:PermSize和-XX:MaxPermSize限制方法区的大小，从而间接限制其中常量池的容量。

4, 方法区溢出

方法区用于存放Class的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述等。也有可能是方法区中保存的class对象没有被及时回收掉或者class信息占用的内存超过了我们配置。

异常信息：java.lang.OutOfMemoryError:PermGenspace

方法区溢出也是一种常见的内存溢出异常，一个类如果要被垃圾收集器回收，判定条件是很苛刻的。在经常动态生成大量Class的应用中，要特别注意这点。

SOF (堆栈溢出StackOverflow) :

StackOverflowError 的定义：当应用程序递归太深而发生堆栈溢出时，抛出该错误。

因为栈一般默认为1-2m，一旦出现死循环或者是大量的递归调用，在不断的压栈过程中，造成栈容量超过1m而导致溢出。

栈溢出的原因：递归调用，大量循环或死循环，全局变量是否过多，数组、List、map数据过大。

26、简述线程、程序、进程的基本概念。以及他们之间关系是什么？

线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享同一块内存空间和一组系统资源，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

程序是含有指令和数据的文件，被存储在磁盘或其他的数据存储设备中，也就是说程序是静态的代码。

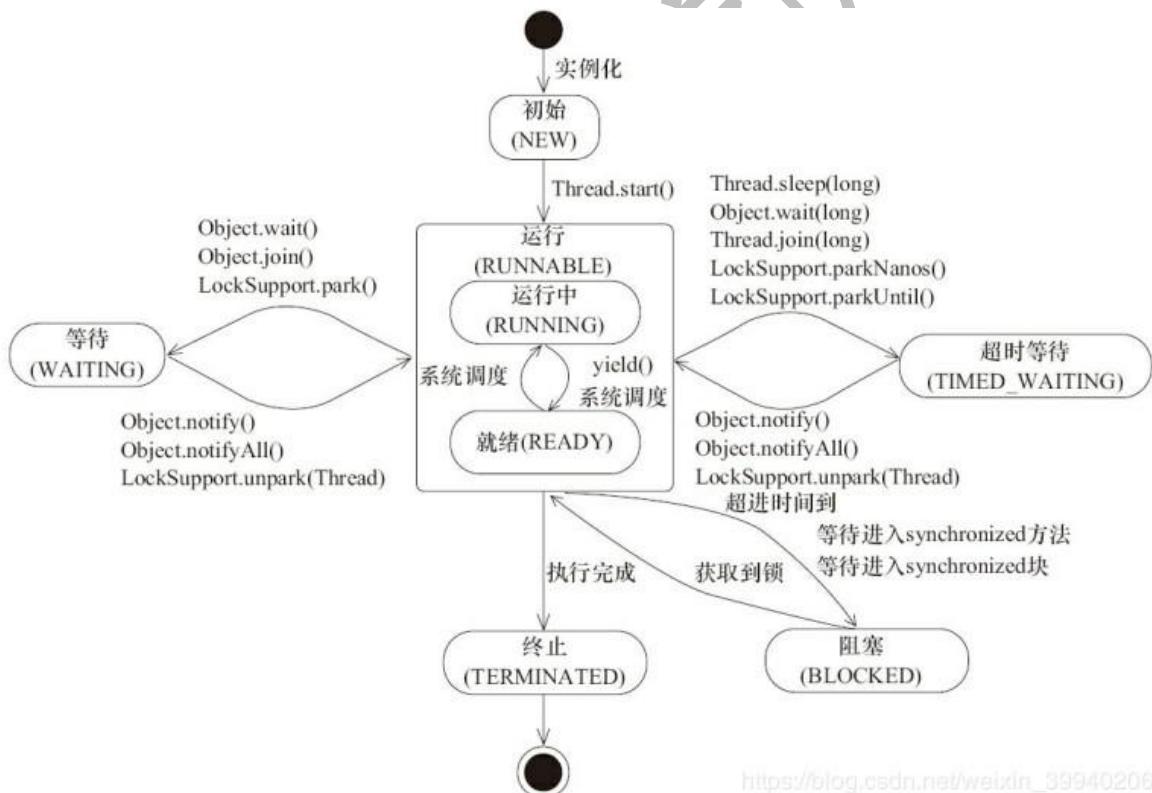
进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。简单来说，一个进程就是一个执行中的程序，它在计算机中一个指令接着一个指令地执行着，同时，每个进程还占有某些系统资源如 CPU 时间，内存空间，文件，输入输出设备的使用权等等。换句话说，当程序在执行时，将被操作系统载入内存中。线程是进程划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。从另一角度来说，进程属于操作系统的范畴，主要是同一段时间内，可以同时执行一个以上的程序，而线程则是在同一程序内几乎同时执行一个以上的程序段。

27、线程有哪些基本状态？

Java 线程在运行的生命周期中的指定时刻只可能处于下面6种不同状态的其中一个状态（图源《Java 并发编程艺术》4.1.4节）。

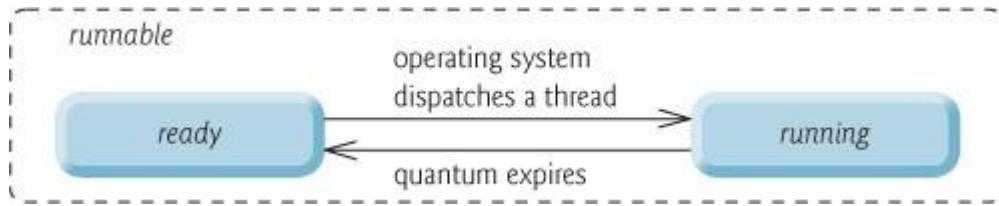
状态名称	说 明
NEW	初始状态，线程被构建，但是还没有调用 start() 方法
RUNNABLE	运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”
BLOCKED	阻塞状态，表示线程阻塞于锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕 https://blog.csdn.net/weixin_39940206

线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换。Java 线程状态变迁如下图所示（图源《Java 并发编程艺术》4.1.4节）：



操作系统隐藏 Java 虚拟机 (JVM) 中的 RUNNABLE 和 RUNNING 状态，它只能看到 RUNNABLE 状态（图源：HowToDoInJava：[Java Thread Life Cycle and Thread States](#)），所以 Java 系统一般将这两个状态统称为 RUNNABLE (运行中) 状态。

操作系统隐藏 Java 虚拟机 (JVM) 中的 RUNNABLE 和 RUNNING 状态，它只能看到 RUNNABLE 状态（图源：HowToDoInJava：[Java Thread Life Cycle and Thread States](#)），所以 Java 系统一般将这两个状态统称为 RUNNABLE (运行中) 状态。



当线程执行 `wait()` 方法之后，线程进入 **WAITING**（等待）状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而 **TIME_WAITING**（超时等待）状态相当于在等待状态的基础上增加了超时限制，比如通过 `sleep(long millis)` 方法或 `wait(long millis)` 方法可以将 Java 线程置于 **TIMED_WAITING** 状态。当超时时间到达后 Java 线程将会返回到 **RUNNABLE** 状态。当线程调用同步方法时，在没有获取到锁的情况下，线程将会进入到 **BLOCKED**（阻塞）状态。线程在执行 **Runnable** 的 `run()` 方法之后将会进入到 **TERMINATED**（终止）状态。

28、Java 序列化中如果有些字段不想进行序列化，怎么办？

对于不想进行序列化的变量，使用 `transient` 关键字修饰。

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

29、Java 中 IO 流

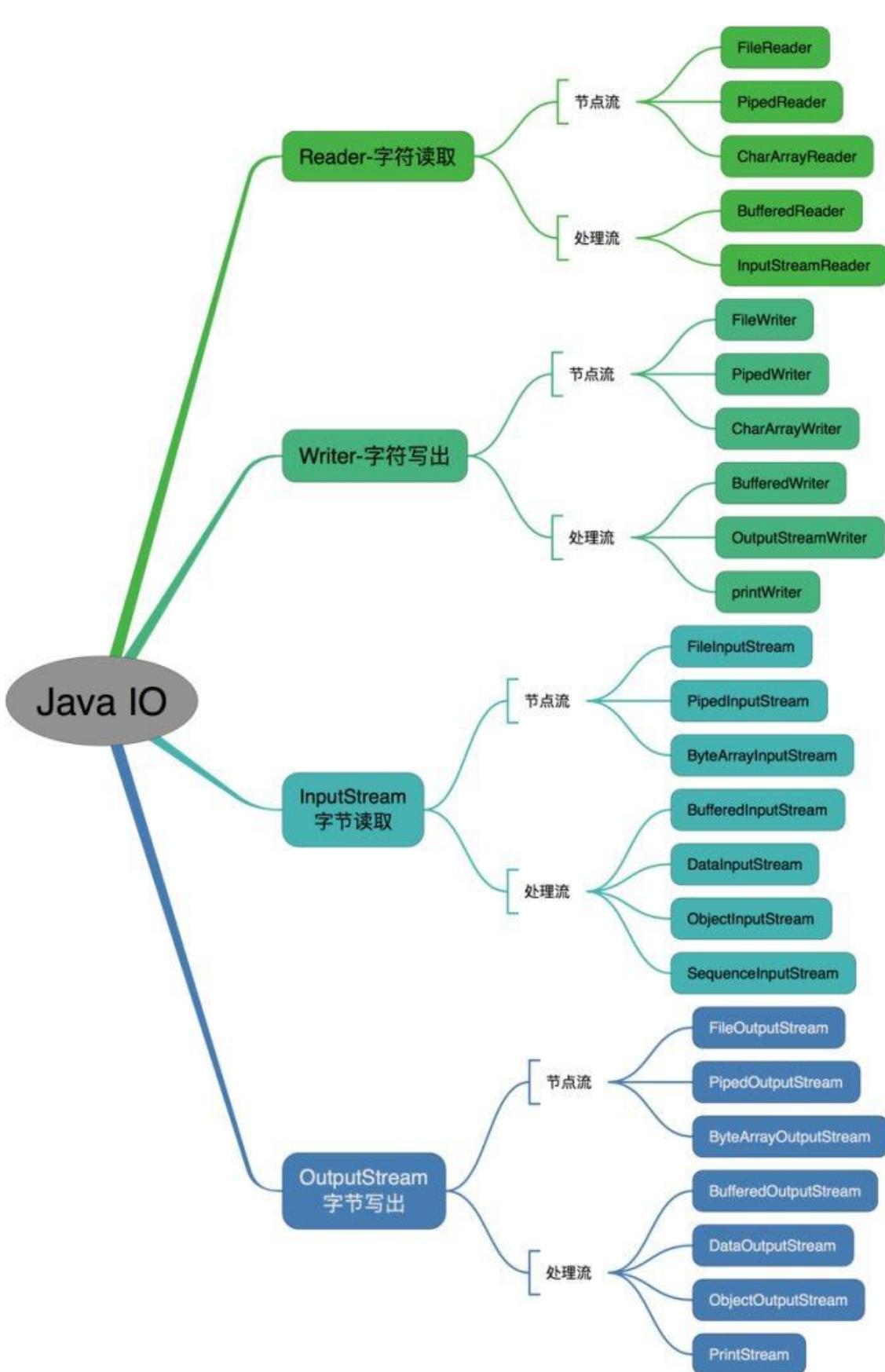
Java 中 **IO** 流分为几种？

- 按照流的流向分，可以分为输入流和输出流；
- 按照操作单元划分，可以划分为字节流和字符流；
- 按照流的角色划分为节点流和处理流。

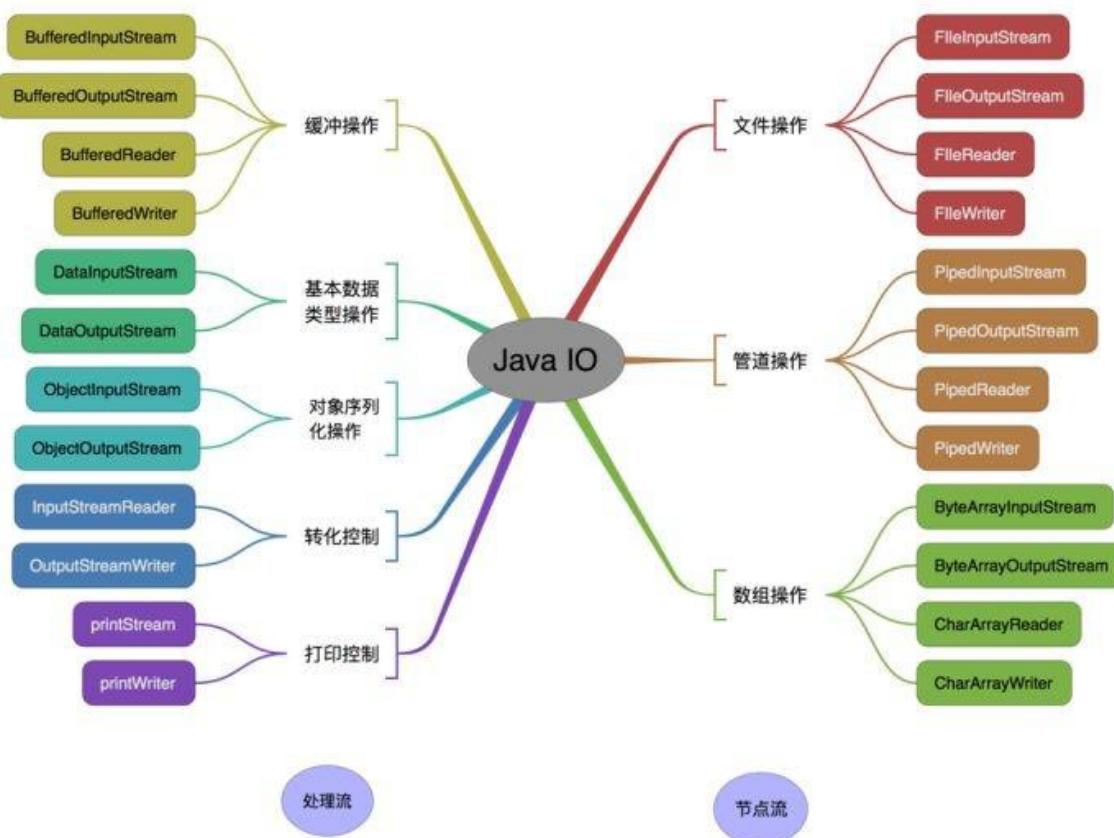
Java **Io** 流共涉及 40 多个类，这些类看上去很杂乱，但实际上很有规则，而且彼此之间存在非常紧密的联系，Java **Io** 流的 40 多个类都是从如下 4 个抽象类基类中派生出来的。

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

按操作方式分类结构图：



按操作对象分类结构图：



30、Java IO与NIO的区别

推荐阅读：<https://mp.weixin.qq.com/s/N1ojvByYmary65B6JM1ZWA>

31、java反射的作用与原理

1、定义：

反射机制是在运行时，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意个对象，都能够调用它的任意一个方法。在java中，只要给定类的名字，就可以通过反射机制来获得类的所有信息。

这种动态获取的信息以及动态调用对象的方法的功能称为Java语言的反射机制。

2、哪里会用到反射机制？

jdbc就是典型的反射

```
Class.forName("com.mysql.jdbc.Driver.class"); //加载MySQL的驱动类
```

这就是反射。如hibernate, struts等框架使用反射实现的。

3、反射的实现方式：

第一步：获取Class对象，有4中方法：

- 1) Class.forName(“类的路径”);
- 2) 类名.class
- 3) 对象名.getClass()
- 4) 基本类型的包装类，可以调用包装类的Type属性来获得该包装类的Class对象

4、实现Java反射的类：

- 1) Class：表示正在运行的Java应用程序中的类和接口

注意：所有获取对象的信息都需要Class类来实现。

- 2) Field：提供有关类和接口的属性信息，以及对它的动态访问权限。
- 3) Constructor：提供关于类的单个构造方法的信息以及它的访问权限
- 4) Method：提供类或接口中某个方法的信息

5、反射机制的优缺点：

优点：

- 1) 能够运行时动态获取类的实例，提高灵活性；
- 2) 与动态编译结合

缺点：

- 1) 使用反射性能较低，需要解析字节码，将内存中的对象进行解析。

解决方案：

- 1、通过setAccessible(true)关闭JDK的安全检查来提升反射速度；
- 2、多次创建一个类的实例时，有缓存会快很多
- 3、ReflectASM工具类，通过字节码生成的方式加快反射速度
- 2) 相对不安全，破坏了封装性（因为通过反射可以获得私有方法和属性）

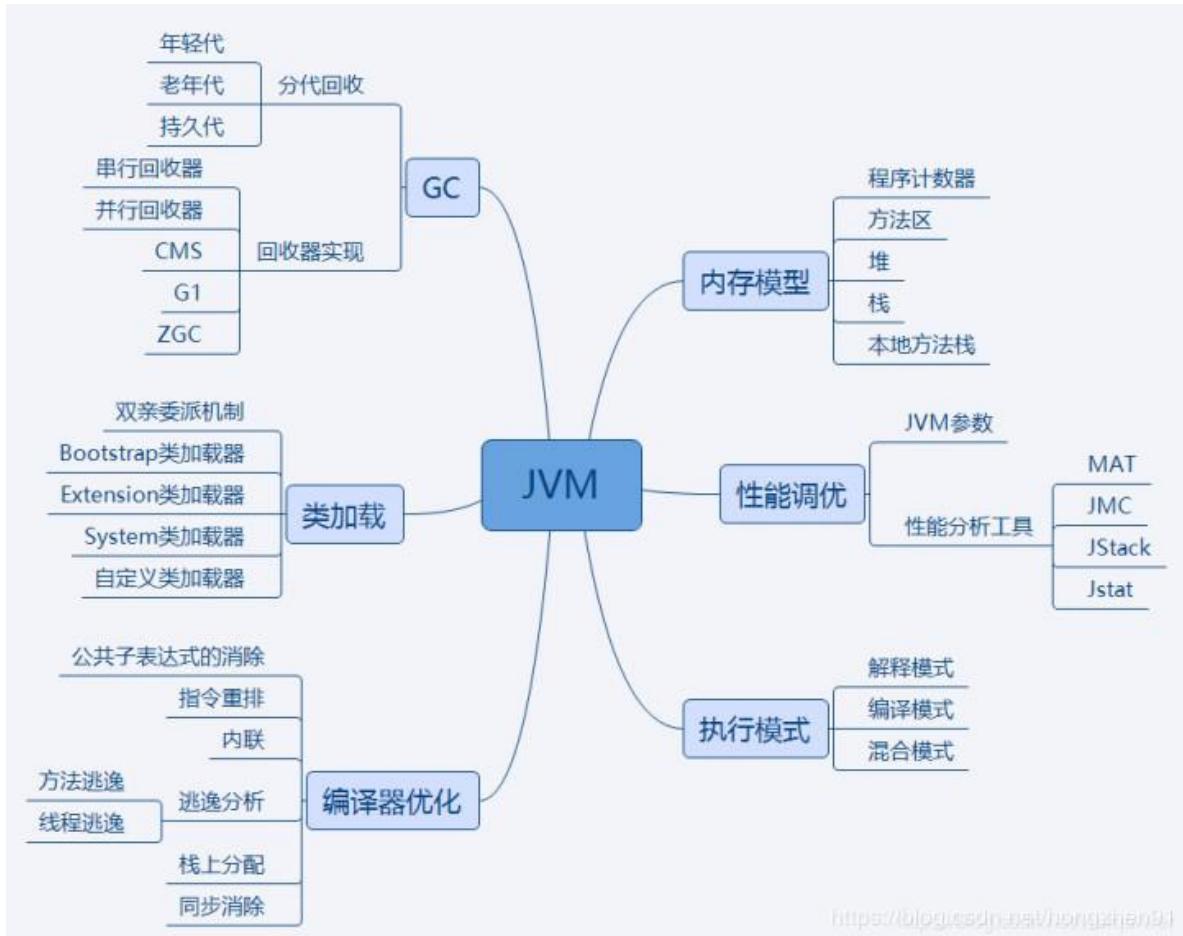
32、说说List,Set,Map三者的区别？

- **List(对付顺序的好帮手)：** List接口存储一组不唯一（可以有多个元素引用相同的对象），有序的对象
- **Set(注重独一无二的性质)：** 不允许重复的集合。不会有多个元素引用相同的对象。
- **Map(用Key来搜索的专家)：** 使用键值对存储。Map会维护与Key有关联的值。两个Key可以引用相同的对象，但Key不能重复，典型的Key是String类型，但也可以是任何对象。

JVM篇

1、知识点汇总

JVM是Java运行基础,面试时一定会遇到JVM的有关问题,内容相对集中,但对只是深度要求较高.



其中内存模型,类加载机制,GC是重点方面.性能调优部分更偏向应用,重点突出实践能力.编译器优化和执行模式部分偏向于理论基础,重点掌握知识点.

需了解

内存模型各部分作用,保存哪些数据.

类加载双亲委派加载机制,常用加载器分别加载哪种类型的类.

GC分代回收的思想和依据以及不同垃圾回收算法的回收思路和适合场景.

性能调优常有JVM优化参数作用,参数调优的依据,常用的JVM分析工具能分析哪些问题以及使用方法.

执行模式解释/编译/混合模式的优缺点,Java 7提供的分层编译技术,JIT即时编译技术,OSR栈上替换,C1/C2编译器针对的场景,C2针对的是server模式,优化更激进.新技术方面Java 10的graal编译器

编译器优化javac的编译过程,ast抽象语法树,编译器优化和运行器优化.

2、知识点详解：

1、JVM内存模型：

线程独占:栈,本地方法栈,程序计数器

线程共享:堆,方法区

2、栈：

又称方法栈,线程私有的,线程执行方法是都会创建一个栈帧,用来存储局部变量表,操作栈,动态链接,方法出口等信息.调用方法时执行入栈,方法返回式执行出栈.

3、本地方法栈

与栈类似,也是用来保存执行方法的信息.执行Java方法是使用栈,执行Native方法时使用本地方法栈.

4、程序计数器

保存着当前线程执行的字节码位置,每个线程工作时都有独立的计数器,只为执行Java方法服务,执行Native方法时,程序计数器为空.

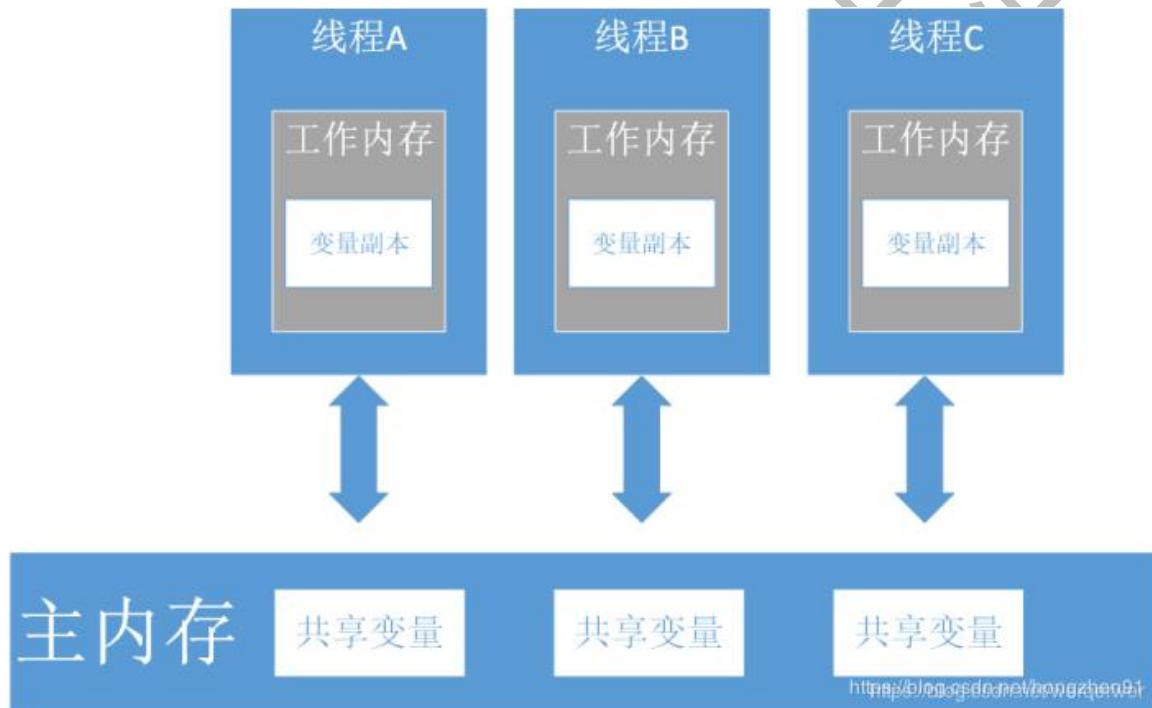
5、堆

JVM内存管理最大的一块,对被线程共享,目的是存放对象的实例,几乎所有的对象实例都会放在这里,当堆没有可用空间时,会抛出OOM异常.根据对象的存活周期不同,JVM把对象进行分代管理,由垃圾回收器进行垃圾的回收管理

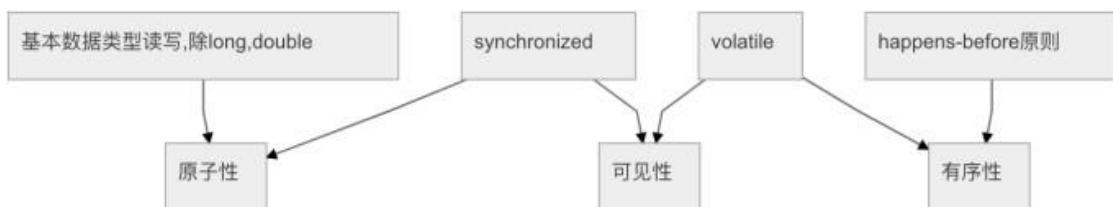
6、方法区:

又称非堆区,用于存储已被虚拟机加载的类信息,常量,静态变量,即时编译器优化后的代码等数据.1.7的永久代和1.8的元空间都是方法区的一种实现

7、JVM 内存可见性

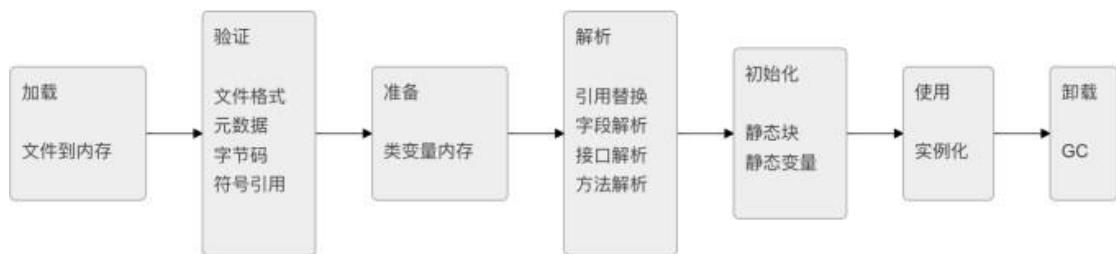


JMM是定义程序中变量的访问规则,线程对于变量的操作只能在自己的工作内存中进行,而不能直接对主内存操作.由于指令重排序,读写的顺序会被打乱,因此JMM需要提供原子性,可见性,有序性保证.



3、类加载与卸载

加载过程



<https://blog.csdn.net/hongzhentil>

其中验证,准备,解析合称链接

加载通过类的完全限定名,查找此类字节码文件,利用字节码文件创建Class对象.

验证确保Class文件符合当前虚拟机的要求,不会危害到虚拟机自身安全.

准备进行内存分配,为static修饰的类变量分配内存,并设置初始值(0或null).不包含final修饰的静态变量,因为final变量在编译时分配.

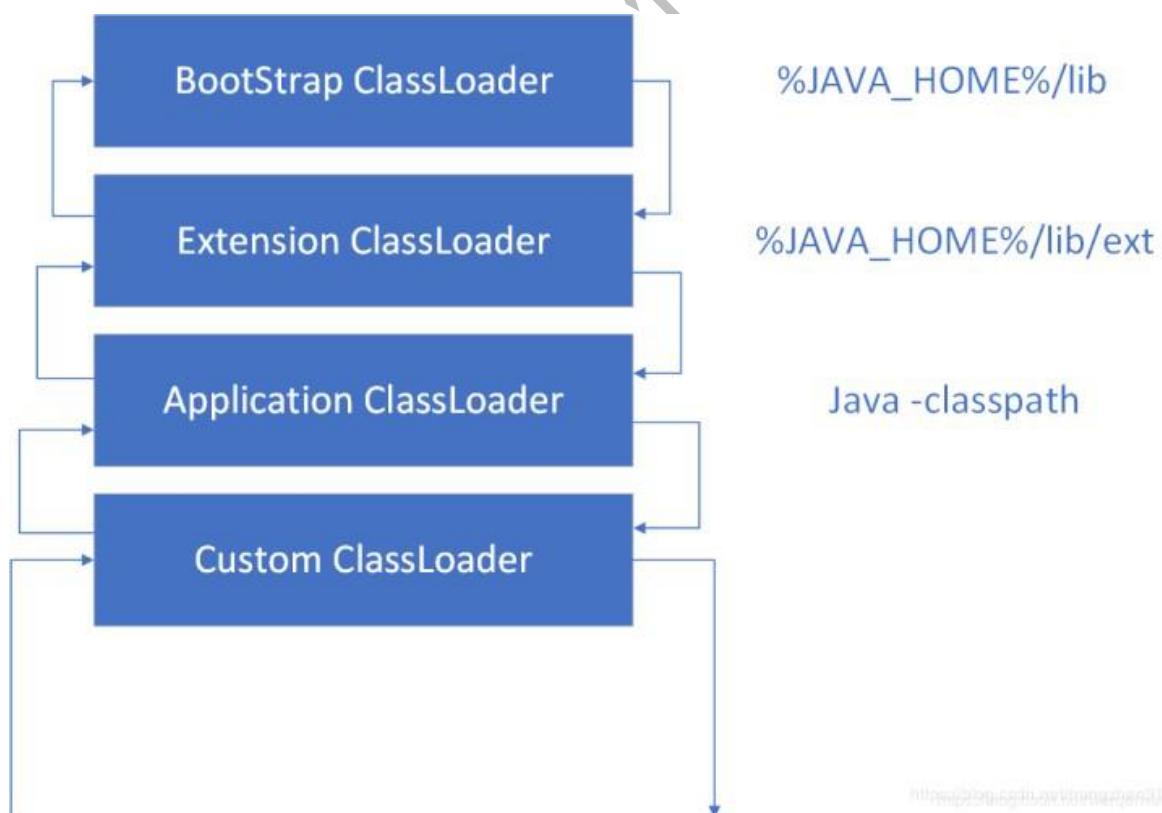
解析将常量池中的符号引用替换为直接引用的过程.直接引用为直接指向目标的指针或者相对偏移量等.

初始化主要完成静态块执行以及静态变量的赋值.先初始化父类,再初始化当前类.只有对类主动使用时才会初始化.

触发条件包括,创建类的实例时,访问类的静态方法或静态变量的时候,使用Class.forName反射类的时候,或者某个子类初始化的时候.

Java自带的加载器加载的类,在虚拟机的生命周期中是不会被卸载的,只有用户自定义的加载器加载的类才可以被卸.

1、加载机制-双亲委派模式



<https://blog.csdn.net/hongzhentil>

双亲委派模式,即加载器加载类时先把请求委托给自己的父类加载器执行,直到顶层的启动类加载器.父类加载器能够完成加载则成功返回,不能则子类加载器才自己尝试加载.*

优点:

1. 避免类的重复加载
2. 避免Java的核心API被篡改

2、分代回收

分代回收基于两个事实:大部分对象很快就不使用了,还有一部分不会立即无用,但也不会持续很长时间.

堆分代			
年轻代	Dden	Survivor1	Survivor2
老年代	Tenured	Tenured	Tenured
永久代	PremGen/MetaSpace	PremGen/MetaSpace	PremGen/MetaSpace

年轻代->标记-复制

老年代->标记-清除

3、回收算法

a、G1算法

1.9后默认的垃圾回收算法,特点保持高回收率的同时减少停顿.采用每次只清理一部分,而不是清理全部的增量式清理,以保证停顿时间不会过长

其取消了年轻代与老年代的物理划分,但仍属于分代收集器,算法将堆分为若干个逻辑区域(region),一部分用作年轻代,一部分用作老年代,还有用来存储巨型对象的分区.

同CMS相同,会遍历所有对象,标记引用情况,清除对象后会对区域进行复制移动,以整合碎片空间.

年轻代回收:

并行复制采用复制算法,并行收集,会StopTheWorld.

老年代回收:

会对年轻代一并回收

初始标记完成堆root对象的标记,会StopTheWorld.

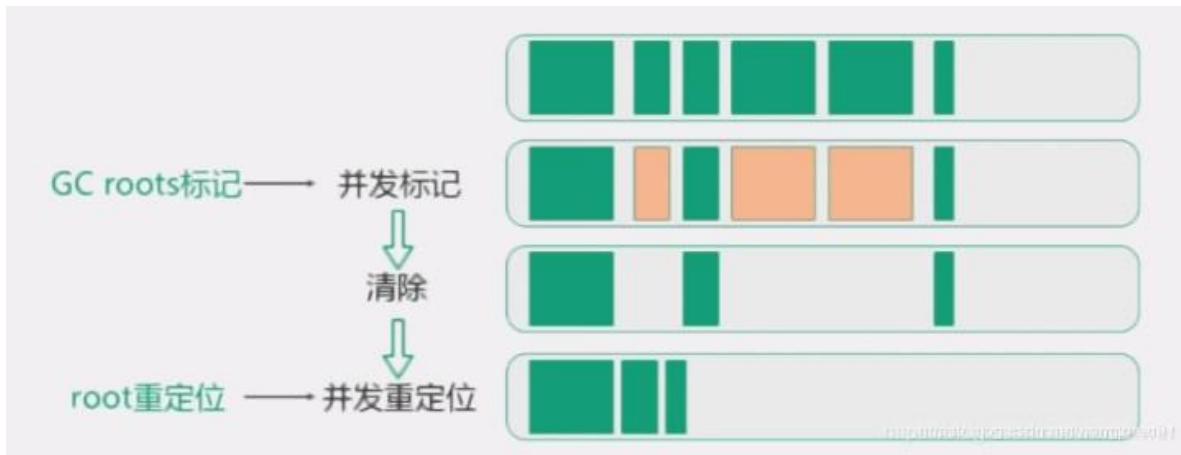
并发标记 GC线程和应用线程并发执行.

最终标记完成三色标记周期,会StopTheWorld.

复制/清楚会优先对可回收空间加大的区域进行回收

b、ZGC算法

前面提供的高效垃圾回收算法,针对大堆内存设计,可以处理TB级别的堆,可以做到10ms以下的回收停顿时间.



- 着色指针
- 读屏障并
- 处理
- 基于region
- 内存压缩(整理)

roots标记：标记root对象,会StopTheWorld.

并发标记：利用读屏障与应用线程一起运行标记,可能会发生StopTheWorld.

清除会清理标记为不可用的对象.

roots重定位：是对存活的对象进行移动,以腾出大块内存空间,减少碎片产生.重定位最开始会StopTheWorld,却取决于重定位集与对象总活动集的比例.

并发重定位与并发标记类似.

4、简述一下JVM的内存模型

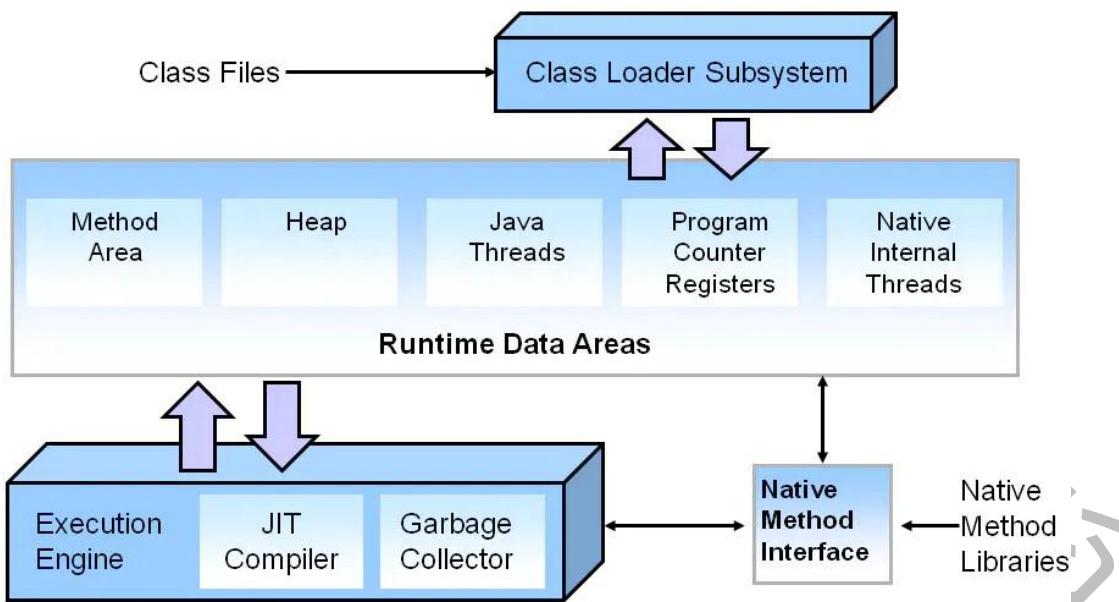
1.JVM内存模型简介

JVM定义了不同运行时数据区，他们是用来执行应用程序的。某些区域随着JVM启动及销毁，另外一些区域的数据是线程性独立的，随着线程创建和销毁。jvm内存模型总体架构图如下：

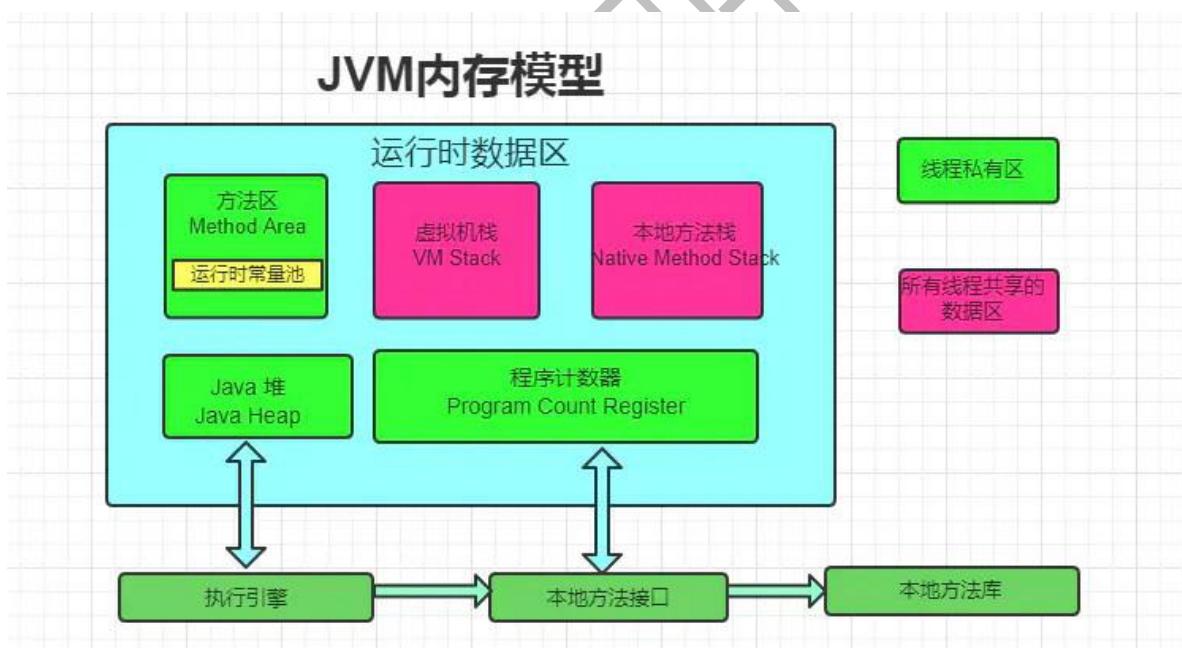


关注【程序员生活志】
免费获取更多海量资源

HotSpot JVM: Architecture



JVM在执行Java程序时，会把它管理的内存划分为若干个的区域，每个区域都有自己的用途和创建销毁时间。如下图所示，可以分为两大部分，线程私有区和共享区。下图是根据自己理解画的一个JVM内存模型架构图：



JVM内存分为线程私有区和线程共享区

线程私有区

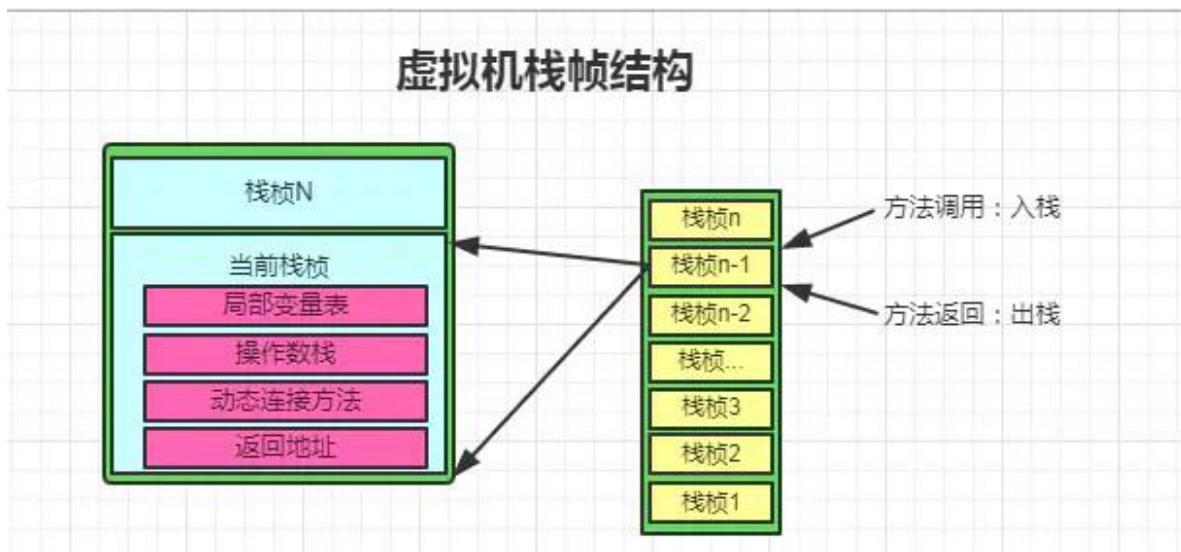
1、程序计数器

当同时进行的线程数超过CPU数或其内核数时，就要通过时间片轮询分派CPU的时间资源，不免发生线程切换。这时，每个线程就需要一个属于自己的计数器来记录下一条要运行的指令。如果执行的是JAVA方法，计数器记录正在执行的java字节码地址，如果执行的是native方法，则计数器为空。

2、虚拟机栈

线程私有的，与线程在同一时间创建。管理JAVA方法执行的内存模型。每个方法执行时都会创建一个桢栈来存储方法的的变量表、操作数栈、动态链接方法、返回值、返回地址等信息。栈的大小决定了方法调用的可达深度（递归多少层次，或嵌套调用多少层其他方法，-Xss参数可以设置虚拟机栈大小）。栈的大小可以是固定的，或者是动态扩展的。如果请求的栈深度大于最大可用深度，则抛出stackOverflowError；如果栈是可动态扩展的，但没有内存空间支持扩展，则抛出OutOfMemoryError。

使用jclasslib工具可以查看class类文件的结构。下图为栈帧结构图：



3、本地方法栈

与虚拟机栈作用相似。但它不是为Java方法服务的，而是本地方法（C语言）。由于规范对这块没有强制要求，不同虚拟机实现方法不同。

线程共享区

1、方法区

线程共享的，用于存放被虚拟机加载的类的元数据信息，如常量、静态变量和即时编译器编译后的代码。若要分代，算是永久代（老年代），以前类大多“static”的，很少被卸载或收集，现回收废弃常量和无用的类。其中运行时常量池存放编译生成的各种常量。（如果hotspot虚拟机确定一个类的定义信息不会被使用，也会将其回收。回收的基本条件至少有：所有该类的实例被回收，而且装载该类的ClassLoader被回收）

2、堆

存放对象实例和数组，是垃圾回收的主要区域，分为新生代和老年代。刚创建的对象在新生代的Eden区中，经过GC后进入新生代的S0区中，再经过GC进入新生代的S1区中，15次GC后仍存在就进入老年代。这是按照一种回收机制进行划分的，不是固定的。若堆的空间不够实例分配，则OutOfMemoryError。



Young Generation	即图中的Eden + From Space (s0) + To Space(s1)
Eden	存放新生的对象
Survivor Space	有两个，存放每次垃圾回收后存活的对象(s0+s1)
Old Generation	Tenured Generation 即图中的Old Space 主要存放应用程序中生命周期长的存活对象

5、堆和栈的区别

栈是运行时单位，代表着逻辑，内含基本数据类型和堆中对象引用，所在区域连续，没有碎片；堆是存储单位，代表着数据，可被多个栈共享（包括成员中基本数据类型、引用和引用对象），所在区域不连续，会有碎片。

1、功能不同

栈内存用来存储局部变量和方法调用，而堆内存用来存储Java中的对象。无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中。

2、共享性不同

栈内存是线程私有的。

堆内存是所有线程共有的。

3、异常错误不同

如果栈内存或者堆内存不足都会抛出异常。

栈空间不足：java.lang.StackOverflowError。

堆空间不足：java.lang.OutOfMemoryError。

4、空间大小

栈的空间大小远远小于堆的。

6、什么时候会触发FullGC

除直接调用System.gc外，触发Full GC执行的情况有如下四种。

1. 旧生代空间不足

旧生代空间只有在新生代对象转入及创建为大对象、大数据时才会出现不足的现象，当执行Full GC后空间仍然不足，则抛出如下错误：

java.lang.OutOfMemoryError: Java heap space

为避免以上两种状况引起的FullGC，调优时应尽量做到让对象在Minor GC阶段被回收、让对象在新生代多存活一段时间及不要创建过大的对象及数组。

2. Permanet Generation空间满

PermanetGeneration中存放的为一些class的信息等，当系统中要加载的类、反射的类和调用的方法较多时，Permanet Generation可能会被占满，在未配置为采用CMS GC的情况下会执行Full GC。如果经过Full GC仍然回收不了，那么JVM会抛出如下错误信息：

java.lang.OutOfMemoryError: PermGen space

为避免Perm Gen占满造成Full GC现象，可采用的方法为增大Perm Gen空间或转为使用CMS GC。

3. CMS GC时出现promotion failed和concurrent mode failure

对于采用CMS进行旧生代GC的程序而言，尤其要注意GC日志中是否有promotion failed和concurrent mode failure两种状况，当这两种状况出现时可能会触发Full GC。

promotion failed是在进行Minor GC时，survivor space放不下、对象只能放入旧生代，而此时旧生代也放不下造成的；concurrent mode failure是在执行CMS GC的过程中同时有对象要放入旧生代，而此时旧生代空间不足造成的。

应对措施为：增大survivorspace、旧生代空间或调低触发并发GC的比率，但在JDK 5.0+、6.0+的版本中有可能会由于JDK的bug29导致CMS在remark完毕后很久才触发sweeping动作。对于这种状况，可通过设置-XX:CMSMaxAbortablePrecleanTime=5（单位为ms）来避免。

4. 统计得到的Minor GC晋升到旧生代的平均大小大于旧生代的剩余空间

这是一个较为复杂的触发情况，Hotspot为了避免由于新生代对象晋升到旧生代导致旧生代空间不足的现象，在进行Minor GC时，做了一个判断，如果之前统计所得到的Minor GC晋升到旧生代的平均大小大于旧生代的剩余空间，那么就直接触发Full GC。

例如程序第一次触发MinorGC后，有6MB的对象晋升到旧生代，那么当下一次Minor GC发生时，首先检查旧生代的剩余空间是否大于6MB，如果小于6MB，则执行Full GC。

当新生代采用PSGC时，方式稍有不同，PS GC是在Minor GC后也会检查，例如上面的例子中第一次Minor GC后，PS GC会检查此时旧生代的剩余空间是否大于6MB，如小于，则触发对旧生代的回收。除了以上4种状况外，对于使用RMI来进行RPC或管理的Sun JDK应用而言，默认情况下会一小时执行一次Full GC。可通过在启动时通过- java-Dsun.rmi.dgc.client.gcInterval=3600000来设置Full GC执行的间隔时间或通过-XX:+ DisableExplicitGC来禁止RMI调用System.gc。

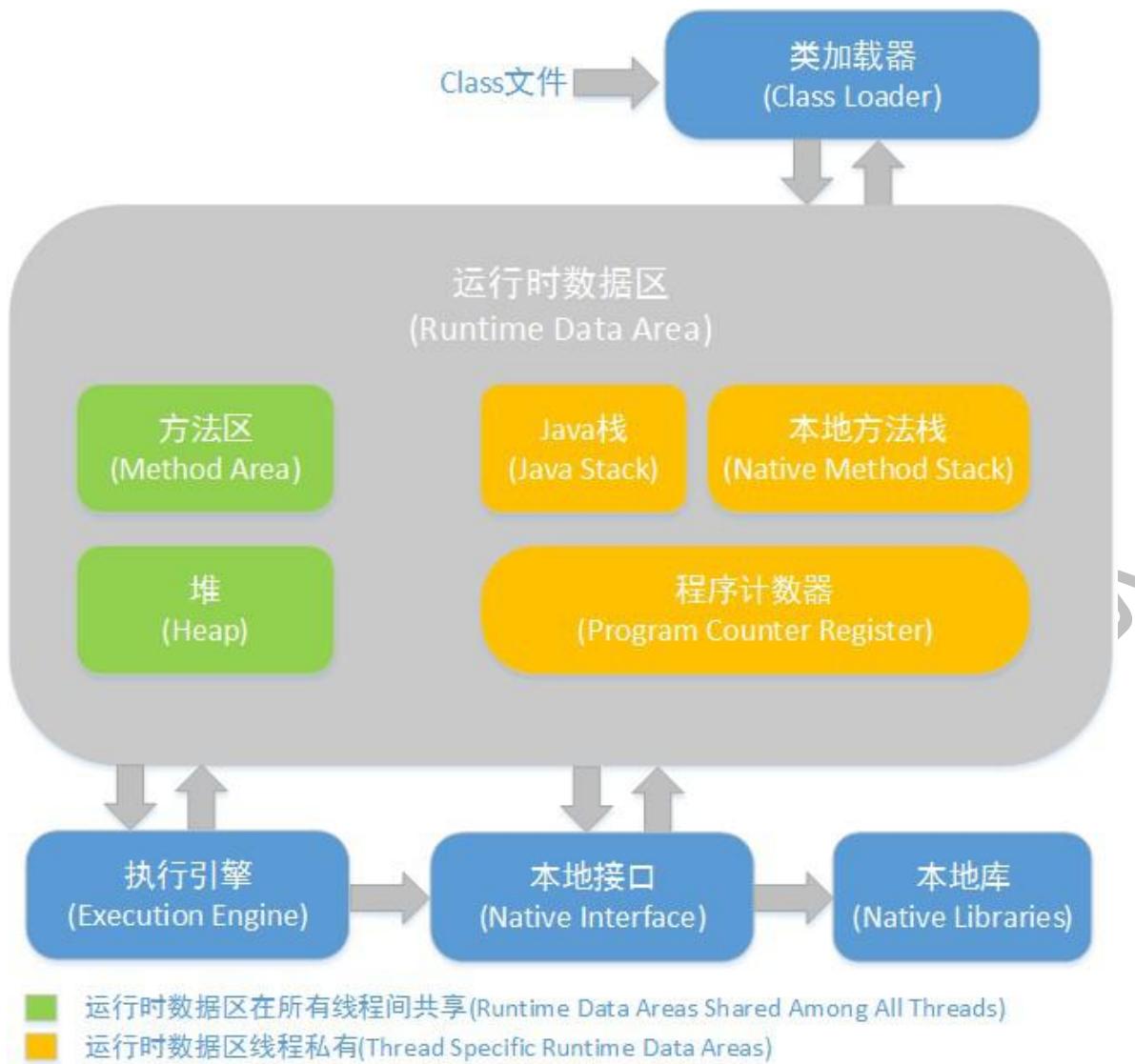
7、什么是Java虚拟机？为什么Java被称作是“平台无关的编程语言”？

Java虚拟机是一个可以执行Java字节码的虚拟机进程。Java源文件被编译成能被Java虚拟机执行的字节码文件。Java被设计成允许应用程序可以运行在任意的平台，而不需要程序员为每一个平台单独重写或者是重新编译。Java虚拟机让这个变为可能，因为它知道底层硬件平台的指令长度和其他特性。

8、Java内存结构



关注【程序员生活志】
免费获取更多海量资源



方法区和堆是所有线程共享的内存区域；而Java栈、本地方法栈和程序员计数器是运行时线程私有的内存区域。

- Java堆 (Heap), 是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。
- 方法区 (Method Area), 方法区 (Method Area) 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。
- 程序计数器 (Program Counter Register), 程序计数器 (Program Counter Register) 是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。
- JVM栈 (JVM Stacks), 与程序计数器一样，Java虚拟机栈 (Java Virtual Machine Stacks) 也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧 (Stack Frame) 用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。
- 本地方法栈 (Native Method Stacks), 本地方法栈 (Native Method Stacks) 与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。

9、对象分配规则

- 对象优先分配在Eden区，如果Eden区没有足够的空间时，虚拟机执行一次Minor GC。
- 大对象直接进入老年区（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在Eden区和两个Survivor区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。
- 长期存活的对象进入老年区。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了1次Minor GC那么对象会进入Survivor区，之后每经过一次Minor GC那么对象的年龄加1，知道达到阀值对象进入老年区。
- 动态判断对象的年龄。如果Survivor区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象可以直接进入老年区。
- 空间分配担保。每次进行Minor GC时，JVM会计算Survivor区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次Full GC，如果小于检查HandlePromotionFailure设置，如果true则只进行Monitor GC,如果false则进行Full GC。

10、描述一下JVM加载class文件的原理机制？

JVM中类的装载是由类加载器（ClassLoader）和它的子类来实现的，Java中的类加载器是一个重要的Java运行时系统组件，它负责在运行时查找和装入类文件中的类。由于Java的跨平台性，经过编译的Java源程序并不是一个可执行程序，而是一个或多个类文件。当Java程序需要使用某个类时，JVM会确保这个类已经被加载、连接（验证、准备和解析）和初始化。类的加载是指把类的.class文件中的数据读入到内存中，通常是创建一个字节数组读入.class文件，然后产生与所加载类对应的Class对象。加载完成后，Class对象还不完整，所以此时的类还不可用。当类被加载后就进入连接阶段，这一阶段包括验证、准备（为静态变量分配内存并设置默认的初始值）和解析（将符号引用替换为直接引用）三个步骤。最后JVM对类进行初始化，包括：1)如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；2)如果类中存在初始化语句，就依次执行这些初始化语句。类的加载是由类加载器完成的，类加载器包括：根加载器（BootStrap）、扩展加载器（Extension）、系统加载器（System）和用户自定义类加载器（java.lang.ClassLoader的子类）。从Java 2 (JDK 1.2) 开始，类加载过程采取了父亲委托机制（PDM）。PDM更好的保证了Java平台的安全性，在该机制中，JVM自带的Bootstrap是根加载器，其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载，父类加载器无能为力时才由其子类加载器自行加载。JVM不会向Java程序提供对Bootstrap的引用。下面是关于几个类加载器的说明：

- Bootstrap：一般用本地代码实现，负责加载JVM基础核心类库（rt.jar）；
- Extension：从java.ext.dirs系统属性所指定的目录中加载类库，它的父加载器是Bootstrap；
- System：又叫应用类加载器，其父类是Extension。它是应用最广泛的类加载器。它从环境变量classpath或者系统属性java.class.path所指定的目录中加载类，是用户自定义加载器的默认父加载器。

11、Java对象创建过程

1.JVM遇到一条新建对象的指令时首先去检查这个指令的参数是否能在常量池中定义到一个类的符号引用。然后加载这个类（类加载过程在后边讲）

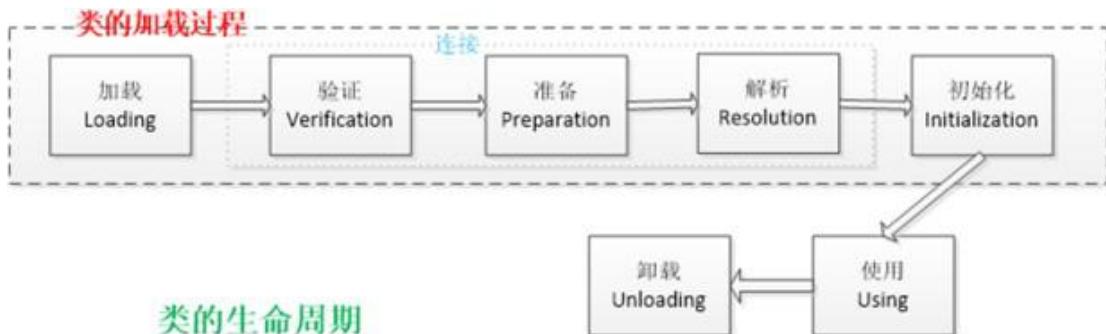
2.为对象分配内存。一种办法“指针碰撞”、一种办法“空闲列表”，最终常用的办法“本地线程缓冲分配(TLAB)”

3.将除对象头外的对象内存空间初始化为0

4.对对象头进行必要设置

12、类的生命周期

类的生命周期包括这几个部分，加载、连接、初始化、使用和卸载，其中前三部是类的加载的过程,如下图；



- 加载，查找并加载类的二进制数据，在Java堆中也创建一个java.lang.Class类的对象
- 连接，连接又包含三块内容：验证、准备、初始化。1) 验证，文件格式、元数据、字节码、符号引用验证；2) 准备，为类的静态变量分配内存，并将其初始化为默认值；3) 解析，把类中的符号引用转换为直接引用
- 初始化，为类的静态变量赋予正确的初始值
- 使用，new出对象程序中使用
- 卸载，执行垃圾回收

13、简述Java的对象结构

Java对象由三个部分组成：对象头、实例数据、对齐填充。

对象头由两部分组成，第一部分存储对象自身的运行时数据：哈希码、GC分代年龄、锁标识状态、线程持有的锁、偏向线程ID（一般占32/64 bit）。第二部分是指针类型，指向对象的类元数据类型（即对象代表哪个类）。如果是数组对象，则对象头中还有一部分用来记录数组长度。

实例数据用来存储对象真正的有效信息（包括父类继承下来的和自己定义的）

对齐填充：JVM要求对象起始地址必须是8字节的整数倍（8字节对齐）

14、如何判断对象可以被回收？

判断对象是否存活一般有两种方式：

- 引用计数：每个对象有一个引用计数属性，新增一个引用时计数加1，引用释放时计数减1，计数为0时可以回收。此方法简单，无法解决对象相互循环引用的问题。
- 可达性分析（Reachability Analysis）：从GC Roots开始向下搜索，搜索所走过的路径称为引用链。当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的，不可达对象。

15、JVM的永久代中会发生垃圾回收么？

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免Full GC是非常重要的原因。请参考下Java8：从永久代到元数据区(注：Java8中已经移除了永久代，新加了一个叫做元数据区的native内存区)

16、垃圾收集算法

GC最基础的算法有三种：标记 -清除算法、复制算法、标记-压缩算法，我们常用的垃圾回收器一般都采用分代收集算法。

- 标记 -清除算法，“标记-清除”（Mark-Sweep）算法，如它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。
- 复制算法，“复制”（Copying）的收集算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。
- 标记-压缩算法，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，然后直接清理掉端边界以外的内存
- 分代收集算法，“分代收集”（Generational Collection）算法，把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

17、调优命令有哪些？

Sun JDK监控和故障处理命令有jps jstat jmap jhat jstack jinfo

- jps, JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程。
- jstat, JVM statistics Monitoring是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。
- jmap, JVM Memory Map命令用于生成heap dump文件
- jhat, JVM Heap Analysis Tool命令是与jmap搭配使用，用来分析jmap生成的dump，jhat内置了一个微型的HTTP/HTML服务器，生成dump的分析结果后，可以在浏览器中查看
- jstack, 用于生成java虚拟机当前时刻的线程快照。
- jinfo, JVM Configuration info 这个命令作用是实时查看和调整虚拟机运行参数。

18、调优工具

常用调优工具分为两类，jdk自带监控工具：jconsole和jvisualvm，第三方有：MAT(Memory Analyzer Tool)、GChisto。

- jconsole, Java Monitoring and Management Console是从java5开始，在JDK中自带的java监控和管理控制台，用于对JVM中内存、线程和类等的监控
- jvisualvm, jdk自带全能工具，可以分析内存快照、线程快照；监控内存变化、GC变化等。MAT，
- Memory Analyzer Tool, 一个基于Eclipse的内存分析工具，是一个快速、功能丰富的Java heap分析工具，它可以帮助我们查找内存泄漏和减少内存消耗
- GChisto, 一款专业分析gc日志的工具

19、Minor GC与Full GC分别在什么时候发生？

新生代内存不够用时候发生MGC也叫YGC，JVM内存不够的时候发生FGC

20、你知道哪些JVM性能调优

- 设定堆内存大小
-Xmx: 堆内存最大限制。
- 设定新生代大小。新生代不宜太小，否则会有大量对象涌入老年代

- XX:NewSize：新生代大小
- XX:NewRatio 新生代和老生代占比
- XX:SurvivorRatio：伊甸园空间和幸存者空间的占比
- 设定垃圾回收器 年轻代用 -XX:+UseParNewGC 年老代用-XX:+UseConcMarkSweepGC

多线程&并发篇

1、Java中实现多线程有几种方法

继承Thread类；
实现Runnable接口；
实现Callable接口通过FutureTask包装器来创建Thread线程；
使用ExecutorService、Callable、Future实现有返回结果的多线程（也就是使用了ExecutorService来管理前面的三种方式）。

2、如何停止一个正在运行的线程

- 1、使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。
- 2、使用stop方法强行终止，但是不推荐这个方法，因为stop和suspend及resume一样都是过期作废的方法。
- 3、使用interrupt方法中断线程。

```
class MyThread extends Thread
{
    volatile boolean stop = false;

    public void run()
    {
        while (!stop) {
            System.out.println(getName() + " is running");
            try {
                sleep(1000);
            } catch (InterruptedException e)
            {
                System.out.println("wake up from
block..."); stop = true; // 在异常处理代码中修改
                共享变量的状态
            }
        }
        System.out.println(getName() + " is exiting...");
    }
}

class InterruptThreadDemo3 {
    public static void main(String[] args) throws InterruptedException
    {
        MyThread m1 = new MyThread();
        System.out.println("Starting thread...");
        m1.start();
        Thread.sleep(3000);
        System.out.println("Interrupt thread...: " + m1.getName());
        m1.stop = true; // 设置共享变量为true
    }
}
```

```
m1.interrupt(); // 阻塞时退出阻塞状态  
Thread.sleep(3000); // 主线程休眠3秒以便观察线程m1的中断情况  
System.out.println("Stopping application...");  
}  
}
```

3、notify()和notifyAll()有什么区别？

notify可能会导致死锁，而notifyAll则不会

任何时候只有一个线程可以获得锁，也就是说只有一个线程可以运行synchronized 中的代码

使用notifyAll,可以唤醒

所有处于wait状态的线程，使其重新进入锁的争夺队列中，而notify只能唤醒一个。

wait() 应配合while循环使用，不应使用if，务必在wait()调用前后都检查条件，如果不满足，必须调用notify()唤醒另外的线程来处理，自己继续wait()直至条件满足再往下执行。

notify() 是对notifyAll()的一个优化，但它有很精确的应用场景，并且要求正确使用。不然可能导致死锁。正确的场景应该是 WaitSet中等待的是相同的条件，唤醒任一个都能正确处理接下来的事项，如果唤醒的线程无法正确处理，务必确保继续notify()下一个线程，并且自身需要重新回到WaitSet中。

4、sleep()和wait() 有什么区别？

对于sleep()方法，我们首先要知道该方法是属于Thread类中的。而wait()方法，则是属于Object类中的。

sleep()方法导致了程序暂停执行指定的时间，让出cpu给其他线程，但是他的监控状态依然保持者，当指定的时间到了又会自动恢复运行状态。在调用sleep()方法的过程中，线程不会释放对象锁。

当调用wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify()方法后本线程才进入对象锁定池准备，获取对象锁进入运行状态。

5、volatile 是什么?可以保证有序性吗?

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

1) 保证了不同线程对这个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,volatile关键字会强制将修改的值立即写入主存。

2) 禁止进行指令重排序。

volatile 不是原子性操作什

么叫保证部分有序性？

当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；

```
x = 2;          //语句1  
y = 0;          //语句2  
flag = true;   //语句3  
x = 4;          //语句4  
y = -1;         //语句5
```

由于flag变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句3放到语句1、语句2前面，也不会讲语句3放到语句4、语句5后面。但是要注意语句1和语句2的顺序、语句4和语句5的顺序是不作任何保证的。

使用 Volatile 一般用于 状态标记量 和 单例模式的双检锁

6、Thread 类中的start() 和 run() 方法有什么区别？

start()方法被用来启动新创建的线程，而且start()内部调用了run()方法，这和直接调用run()方法的效果不一样。当你调用run()方法的时候，只会是在原来的线程中调用，没有新的线程启动，start()方法才会启动新线程。

7、为什么wait, notify 和 notifyAll这些方法不在thread类里面？

明显的原因是JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的wait()方法就有意义了。如果wait()方法定义在Thread类中，线程正在等待的是哪个锁就不明显了。简单的说，由于wait, notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象。

8、为什么wait和notify方法要在同步块中调用？

1. 只有在调用线程拥有某个对象的独占锁时，才能够调用该对象的wait(),notify()和notifyAll()方法。
2. 如果你不这么做，你的代码会抛出IllegalMonitorStateException异常。
3. 还有一个原因是为了避免wait和notify之间产生竞态条件。

wait()方法强制当前线程释放对象锁。这意味着在调用某对象的wait()方法之前，当前线程必须已经获得该对象的锁。因此，线程必须在某个对象的同步方法或同步代码块中才能调用该对象的wait()方法。

在调用对象的notify()和notifyAll()方法之前，调用线程必须已经得到该对象的锁。因此，必须在某个对象的同步方法或同步代码块中才能调用该对象的notify()或notifyAll()方法。

调用wait()方法的原因通常是，调用线程希望某个特殊的状态(或变量)被设置之后再继续执行。调用notify()或notifyAll()方法的原因通常是，调用线程希望告诉其他等待中的线程：“特殊状态已经被设置”。这个状态作为线程间通信的通道，它必须是一个可变的共享状态(或变量)。

9、Java中interrupted 和 isInterrupted方法的区别？

interrupted() 和 isInterrupted()的主要区别是前者会将中断状态清除而后者不会。Java多线程的中断机制是用内部标识来实现的，调用Thread.interrupt()来中断一个线程就会设置中断标识为true。当中断线程调用静态方法Thread.interrupted()来检查中断状态时，中断状态会被清零。而非静态方法isInterrupted()用来查询其它线程的中断状态且不会改变中断状态标识。简单的说就是任何抛出InterruptedException异常的方法都会将中断状态清零。无论如何，一个线程的中断状态有可能被其它线程调用中断来改变。

10、Java中synchronized 和 ReentrantLock 有什么不同？

相似点：

这两种同步方式有很多相似之处，它们都是加锁方式同步，而且都是阻塞式的同步，也就是说当如果一个线程获得了对象锁，进入了同步块，其他访问该同步块的线程都必须阻塞在同步块外面等待，而进行线程阻塞和唤醒的代价是比较高的。

区别：

这两种方式最大区别就是对于Synchronized来说，它是java语言的关键字，是原生语法层面的互斥，需要jvm实现。而ReentrantLock它是JDK 1.5之后提供的API层面的互斥锁，需要lock()和unlock()方法配合try/finally语句块来完成。

Synchronized进过编译，会在同步块的前后分别形成monitorenter和monitorexit这个两个字节码指令。在执行monitorenter指令时，首先要尝试获取对象锁。如果这个对象没被锁定，或者当前线程已经拥有了那个对象锁，把锁的计算器加1，相应的，在执行monitorexit指令时会将锁计算器减1，当计算器为0时，锁就被释放了。如果获取对象锁失败，那当前线程就要阻塞，直到对象锁被另一个线程释放为止。

由于ReentrantLock是java.util.concurrent包下提供的一套互斥锁，相比Synchronized，

ReentrantLock类提供了一些高级功能，主要有以下3项：

1. 等待可中断，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于Synchronized来说可以避免出现死锁的情况。
2. 公平锁，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，Synchronized锁非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，可以通过参数true设为公平锁，但公平锁表现的性能不是很好。
3. 锁绑定多个条件，一个ReentrantLock对象可以同时绑定对个对象。

11、有三个线程T1,T2,T3,如何保证顺序执行？

在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3调用T2， T2调用T1)，这样T1就会先完成而T3最后完成。

实际上先启动三个线程中哪一个都行，

因为在每个线程的run方法中用join方法限定了三个线程的执行顺序。

```
public class JoinTest2 {  
  
    // 1. 现在有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  
    public static void main(String[] args) {  
  
        final Thread t1 = new Thread(new Runnable() {  
  
            @Override  
            public void run()  
            { System.out.println("t1");  
            };  
        });  
        final Thread t2 = new Thread(new Runnable() {  
    
```

```

@Override
public void run() {
    { try {
        // 引用t1线程，等待t1线程执行完
        t1.join();
    } catch (InterruptedException e)
    { e.printStackTrace();
    }
    System.out.println("t2");
    }
}
};

Thread t3 = new Thread(new Runnable() {

@Override
public void run() {
    { try {
        // 引用t2线程，等待t2线程执行完
        t2.join();
    } catch (InterruptedException e)
    { e.printStackTrace();
    }
    System.out.println("t3");
    }
}
});

t3.start(); //这里三个线程的启动顺序可以任意，大家可以试下！
t2.start();
t1.start();
}
}
}

```

12、SynchronizedMap和ConcurrentHashMap有什么区别？

SynchronizedMap()和Hashtable一样，实现上在调用map所有方法时，都对整个map进行同步。而ConcurrentHashMap的实现却更加精细，它对map中的所有桶加了锁。所以，只要有一个线程访问map，其他线程就无法进入map，而如果一个线程在访问ConcurrentHashMap某个桶时，其他线程，仍然可以对map执行某些操作。

所以，ConcurrentHashMap在性能以及安全性方面，明显比Collections.synchronizedMap()更加有优势。同时，同步操作精确控制到桶，这样，即使在遍历map时，如果其他线程试图对map进行数据修改，也不会抛出ConcurrentModificationException。

13、什么是线程安全

线程安全就是说多线程访问同一代码，不会产生不确定的结果。

在多线程环境中，当各线程不共享数据的时候，即都是私有（private）成员，那么一定是线程安全的。但这种情况并不多见，在多数情况下需要共享数据，这时就需要进行适当的同步控制了。

线程安全一般都涉及到synchronized，就是一段代码同时只能有一个线程来操作不然中间过程可能会产生不可预知的结果。

如果你的代码所在的进程中多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的一样的，就是线程安全的。

14、Thread类中的yield方法有什么作用？

Yield方法可以暂停当前正在执行的线程对象，让其它有相同优先级的线程执行。它是一个静态方法而且只保证当前线程放弃CPU占用而不能保证使其它线程一定能占用CPU，执行yield()的线程有可能在进入到暂停状态后马上又被执行。

15、Java线程池中submit() 和 execute()方法有什么区别？

两个方法都可以向线程池提交任务，execute()方法的返回类型是void，它定义在Executor接口中，而submit()方法可以返回持有计算结果的Future对象，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法。

16、说一说自己对于 synchronized 关键字的了解

synchronized关键字解决的是多个线程之间访问资源的同步性，synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

另外，在 Java 早期版本中，synchronized属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态 转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的 synchronized 效率低的原因。庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

17、说说自己是怎么使用 synchronized 关键字，在项目中用到了吗synchronized关键字最主要的三种使用方式：

修饰实例方法：作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁

修饰静态方法：也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（static 表明这是该类的一个静态资源，不管new了多少个对象，只有一份）。所以如果一个线程A调用一个实例对象的非静态 synchronized 方法，而线程B需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁。

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。

总结：synchronized 关键字加到 static 静态方法和 synchronized(class)代码块上都是给 Class 类上锁。synchronized 关键字加到实例方法上是给对象实例上锁。尽量不要使用 synchronized(String a) 因为JVM中，字符串常量池具有缓存功能！

18、什么是线程安全？Vector是一个线程安全类吗？

如果你的代码所在的进程中多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量 的值也和预期的一样的，就是线程安全的。一个线程安全的计数器类的同一个实例对象在被多个线程使用的情况下也不会出现计算失误。很显然你可以将集合类分成两组，线程安全和非线程安全的。Vector 是用同步方法来实现线程安全的，而和它相似

的ArrayList不是线程安全的。

19、 volatile关键字的作用？

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

- 保证了不同线程对这个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。
- 禁止进行指令重排序。
- volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取； synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。
- volatile仅能使用在变量级别； synchronized则可以使用在变量、方法、和类级别的。
- volatile仅能实现变量的修改可见性，并不能保证原子性； synchronized则可以保证变量的修改可见性和原子性。
- volatile不会造成线程的阻塞； synchronized可能会造成线程的阻塞。

volatile标记的变量不会被编译器优化； synchronized标记的变量可以被编译器优化。

20、常用的线程池有哪些？

- newSingleThreadExecutor：创建一个单线程的线程池，此线程池保证所有任务的执行顺序按照任务的提交顺序执行。
- newFixedThreadPool：创建固定大小的线程池，每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。
- newCachedThreadPool：创建一个可缓存的线程池，此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。
- newScheduledThreadPool：创建一个大小无限的线程池，此线程池支持定时以及周期性执行任务的需求。
- newSingleThreadExecutor：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

21、简述一下你对线程池的理解

（如果问到了这样的问题，可以展开的说一下线程池如何用、线程池的好处、线程池的启动策略）合理利用线程池能够带来三个好处。

第一：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。

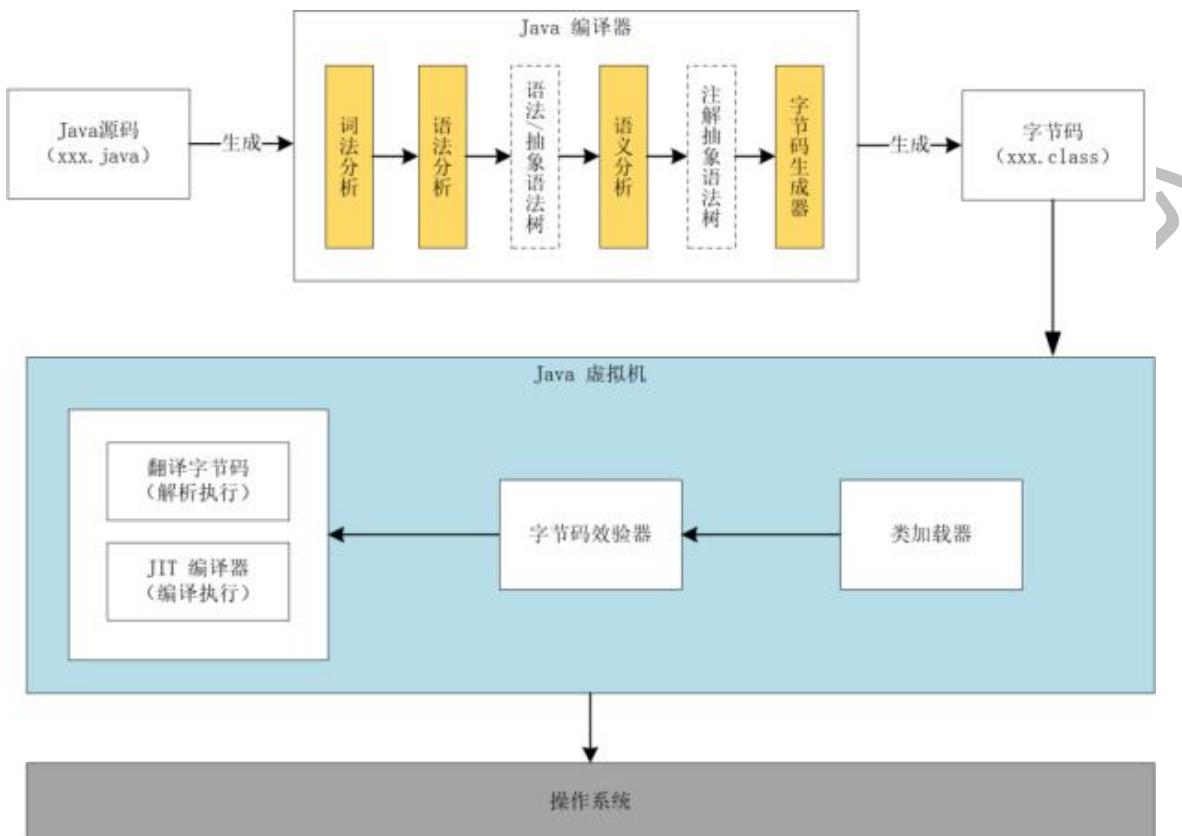
第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

22、Java程序是如何执行的

我们日常的工作中都使用开发工具（IntelliJ IDEA 或 Eclipse 等）可以很方便的调试程序，或者是通过打包工具把项目打包成 jar 包或者 war 包，放入 Tomcat 等 Web 容器中就可以正常运行了，但你有没有想过 Java 程序内部是如何执行的？其实不论是在开发工具中运行还是在 Tomcat 中运行，Java 程序的执行流程基本都是相同的，它的执行流程如下：

- 先把 Java 代码编译成字节码，也就是把 .java 类型的文件编译成 .class 类型的文件。这个过程的大致执行流程：Java 源代码 -> 词法分析器 -> 语法分析器 -> 语义分析器 -> 字符码生成器 -> 最终生成字节码，其中任何一个节点执行失败就会造成编译失败；
- 把 class 文件放置到 Java 虚拟机，这个虚拟机通常指的是 Oracle 官方自带的 Hotspot JVM；
- Java 虚拟机使用类加载器（Class Loader）装载 class 文件；
- 类加载完成之后，会进行字节码效验，字节码效验通过之后 JVM 解释器会把字节码翻译成机器码交由操作系统执行。但不是所有代码都是解释执行的，JVM 对此做了优化，比如，以 Hotspot 虚拟机来说，它本身提供了 JIT（Just In Time）也就是我们通常所说的动态编译器，它能够在运行时将热点代码编译为机器码，这个时候字节码就变成了编译执行。Java 程序执行流程图如下：



Spring篇

1、 Spring的IOC和AOP机制？

我们是在使用Spring框架的过程中，其实就是为了使用IOC，依赖注入，和AOP，面向切面编程，这两个是Spring的灵魂。

主要用到的设计模式有工厂模式和代理模式。

IOC就是典型的工厂模式，通过sessionfactory去注入实例。

AOP就是典型的代理模式的体现。

代理模式是常用的java设计模式，他的特征是代理类与委托类有同样的接口，代理类主要负责为委托类 预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等。代理类与委托类之间通常会存在 关联关系，一个代理类的对象与一个委托类的对象关联，代理类的对象本身并不真正实现服务，而是通 过调用委托类的对象的相关方法，来提供特定的服务。

spring的IoC容器是spring的核心，spring AOP是spring框架的重要组成部分。

在传统的程序设计中，当调用者需要被调用者的协助时，通常由调用者来创建被调用者的实例。但在spring里创建被调用者的工作不再由调用者来完成，因此控制反转（IoC）；创建被调用者实例的工作通常由spring容器来完成，然后注入调用者，因此也被称为依赖注入（DI），依赖注入和控制反转是同一个概念。

面向方面编程（AOP）是以另一个角度来考虑程序结构，通过分析程序结构的关注点来完善面向对象编程（OOP）。OOP将应用程序分解成各个层次的对象，而AOP将程序分解成多个切面。spring AOP 只实现了方法级别的连接点，在J2EE应用中，AOP拦截到方法级别的操作就已经足够。在spring中，未来使IoC方便地使用健壮、灵活的企业服务，需要利用spring AOP实现为IoC和企业服务之间建立联系。

IOC:控制反转也叫依赖注入。利用了工厂模式

将对象交给容器管理，你只需要在**spring**配置文件总配置相应的**bean**，以及设置相关的属性，让**spring**容器来生成类的实例对象以及管理对象。在**spring**容器启动的时候，**spring**会把你配置文件中配置的**bean**都初始化好，然后在你需要调用的时候，就把它已经初始化好的那些**bean**分配给你需要调用这些**bean**的类（假设这个类名是**A**），分配的方法就是调用**A**的**setter**方法来注入，而不需要你在**A**里面new这些**bean**了。

注意：面试的时候，如果有条件，画图，这样更加显得你懂了。

spring ioc初始化流程



AOP:面向切面编程。（Aspect-Oriented Programming）

AOP可以说是对**OOP**的补充和完善。**OOP**引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候，**OOP**则显得无能为力。也就是说，**OOP**允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能。日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。在**OOP**设计中，它导致了大量代码的重复，而不利于各个模块的重用。

将程序中的交叉业务逻辑（比如安全，日志，事务等），封装成一个切面，然后注入到目标对象（具体业务逻辑）中去。

实现**AOP**的技术，主要分为两大类：一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

简单点解释，比方说你想在你的biz层所有类中都加上一个打印‘你好’的功能，这时就可以用aop思想来做。你先写个类写个类方法，方法经实现打印‘你好’，然后在类 ref = “biz.*” 让每个类都注入即可实现。

2、Spring中**Autowired**和**Resource**关键字的区别？

@Resource和@Autowired都是做bean的注入时使用，其实@Resource并不是Spring的注解，它的包是javax.annotation.Resource，需要导入，但是Spring支持该注解的注入。

1、共同点

两者都可以写在字段和setter方法上。两者如果都写在字段上，那么就不需要再写setter方法。

2、不同点

- (1) @Autowired

@Autowired为Spring提供的注解，需要导入包org.springframework.beans.factory.annotation.Autowired;只按照byType注入。

```
public class TestServiceImpl {
    // 下面两种@Autowired只要使用一种即可
    @Autowired
    private UserDao userDao; // 用于字段上

    @Autowired
    public void setUserDao(UserDao userDao) { // 用于属性的方法上
        this.userDao = userDao;
    }
}
```

@Autowired注解是按照类型（byType）装配依赖对象，默认情况下它要求依赖对象必须存在，如果允许null值，可以设置它的required属性为false。如果我们想使用按名称（byName）来装配，可以结合@Qualifier注解一起使用。如下：

```
public class TestServiceImpl {
    @Autowired
    @Qualifier("userDao")
    private UserDao userDao;
}
```

(2) @Resource

@Resource默认按照ByName自动注入，由J2EE提供，需要导入包javax.annotation.Resource。
@Resource有两个重要的属性：name和type，而Spring将@Resource注解的name属性解析为bean的名字，而type属性则解析为bean的类型。所以，如果使用name属性，则使用ByName的自动注入策略，而使用type属性时则使用byType自动注入策略。如果既不制定name也不制定type属性，这时将通过反射机制使用ByName自动注入策略。

```
public class TestServiceImpl {
    // 下面两种@Resource只要使用一种即可
    @Resource(name="userDao")
    private UserDao userDao; // 用于字段上

    @Resource(name="userDao")
    public void setUserDao(UserDao userDao) { // 用于属性的setter方法上
        this.userDao = userDao;
    }
}
```

注：最好是将@Resource放在setter方法上，因为这样更符合面向对象的思想，通过set、get去操作属性，而不是直接去操作属性。

@Resource装配顺序：

- ①如果同时指定了name和type，则从Spring上下文中找到唯一匹配的bean进行装配，找不到则抛出异常。
- ②如果指定了name，则从上下文中查找名称（id）匹配的bean进行装配，找不到则抛出异常。
- ③如果指定了type，则从上下文中找到类似匹配的唯一bean进行装配，找不到或是找到多个，都会抛

出异常。



关注【程序员生活志】
免费获取更多海量资源

云朵里·程序员生活志

④如果既没有指定name，又没有指定type，则自动按照byName方式进行装配；如果没有匹配，则回退为一个原始类型进行匹配，如果匹配则自动装配。

@Resource的作用相当于@Autowired，只不过@Autowired按照byType自动注入。

3、依赖注入的方式有几种，各是什么？

一、构造器注入

将被依赖对象通过构造函数的参数注入给依赖对象，并且在初始化对象的时候注入。

优点：

对象初始化完成后便可获得可使用的对象。

缺点：

当需要注入的对象很多时，构造器参数列表将会很长；

不够灵活。若有多种注入方式，每种方式只需注入指定几个依赖，那么就需要提供多个重载的构造函数，麻烦。

二、setter方法注入

IoC Service Provider通过调用成员变量提供的setter函数将被依赖对象注入给依赖类。

优点：

灵活。可以选择性地注入需要的对象。

缺点：

依赖对象初始化完成后由于尚未注入被依赖对象，因此还不能使用。

三、接口注入

依赖类必须要实现指定的接口，然后实现该接口中的一个函数，该函数就是用于依赖注入。该函数的参数就是要注入的对象。

优点

接口注入中，接口的名字、函数的名字都不重要，只要保证函数的参数是要注入的对象类型即可。

缺点：

侵入行太强，不建议使用。

PS：什么是侵入行？

如果类A要使用别人提供的一个功能，若为了使用这功能，需要在自己的类中增加额外的代码，这就是侵入性。

4、讲一下什么是Spring

Spring是一个轻量级的IoC和AOP容器框架。是为Java应用程序提供基础性服务的一套框架，目的是用于简化企业应用程序的开发，它使得开发者只需要关心业务需求。常见的配置方式有三种：基于XML的配置、基于注解的配置、基于Java的配置。

主要由以下几个模块组成：

Spring Core：核心类库，提供IOC服务；

Spring Context：提供框架式的Bean访问方式，以及企业级功能（JNDI、定时任务等）；

Spring AOP：AOP服务；

Spring DAO：对JDBC的抽象，简化了数据访问异常的处理；

Spring ORM：对现有的ORM框架的支持；

Spring Web：提供了基本的面向Web的综合特性，例如多方文件上传；

Spring MVC：提供面向Web应用的Model-View-Controller实现。

- 1、 用户发送请求至前端控制器DispatcherServlet。
- 2、 DispatcherServlet收到请求调用HandlerMapping处理器映射器。
- 3、 处理器映射器找到具体的处理器(可以根据xml配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
- 4、 DispatcherServlet调用HandlerAdapter处理器适配器。
- 5、 HandlerAdapter经过适配调用具体的处理器(Controller，也叫后端控制器)。
- 6、 Controller执行完成返回ModelAndView。
- 7、 HandlerAdapter将controller执行结果 ModelAndView 返回给DispatcherServlet。
- 8、 DispatcherServlet将 ModelAndView 传给 ViewReslover 视图解析器。
- 9、 ViewReslover解析后返回具体 View。
- 10、 DispatcherServlet根据 View 进行渲染视图 (即将模型数据填充至视图中)。
- 11、 DispatcherServlet响应用户。

组件说明：

以下组件通常使用框架提供实现：

DispatcherServlet：作为前端控制器，整个流程控制的中心，控制其它组件执行，统一调度，降低组件之间的耦合性，提高每个组件的扩展性。



关注【程序员生活志】
免费获取更多海量资源

HandlerMapping: 通过扩展处理器映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

HandlerAdapter: 通过扩展处理器适配器，支持更多类型的处理器。

ViewResolver: 通过扩展视图解析器，支持更多类型的视图解析，例如：jsp、freemarker、pdf、excel等。

组件：

1、前端控制器**DispatcherServlet**（不需要工程师开发），由框架提供

作用：接收请求，响应结果，相当于转发器，中央处理器。有了dispatcherServlet减少了其它组件之间的耦合度。

用户请求到达前端控制器，它就相当于mvc模式中的c，dispatcherServlet是整个流程控制的中心，由它调用其它组件处理用户的请求，dispatcherServlet的存在降低了组件之间的耦合性。

2、处理器映射器**HandlerMapping**（不需要工程师开发），由框架提供

作用：根据请求的url查找Handler

HandlerMapping负责根据用户请求找到Handler即处理器，springmvc提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

3、处理器适配器**HandlerAdapter**

作用：按照特定规则（HandlerAdapter要求的规则）去执行Handler

通过HandlerAdapter对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

4、处理器**Handler**（需要工程师开发）

注意：编写Handler时按照HandlerAdapter的要求去做，这样适配器才可以去正确执行Handler

Handler 是继DispatcherServlet前端控制器的后端控制器，在DispatcherServlet的控制下Handler对具体的用户请求进行处理。

由于Handler涉及到具体的用户业务请求，所以一般情况需要工程师根据业务需求开发Handler。

5、视图解析器**View resolver**（不需要工程师开发），由框架提供

作用：进行视图解析，根据逻辑视图名解析成真正的视图（view）

View Resolver负责将处理结果生成View视图，View Resolver首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成View视图对象，最后对View进行渲染将处理结果通过页面展示给用户。

springmvc框架提供了很多的View视图类型，包括：jstlView、freemarkerView、pdfView等。

一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由工程师根据业务需求开发具体的页面。

6、视图**View**（需要工程师开发jsp...）

View是一个接口，实现类支持不同的View类型（jsp、freemarker、pdf...）

核心架构的具体流程步骤如下：

1、首先用户发送请求——>DispatcherServlet，前端控制器收到请求后自己不进行处理，而是委托给其他的解析器进行处理，作为统一访问点，进行全局的流程控制；

2、DispatcherServlet——>HandlerMapping，HandlerMapping 将会把请求映射为HandlerExecutionChain 对象（包含一个Handler 处理器（页面控制器）对象、多个HandlerInterceptor 拦截器）对象，通过这种策略模式，很容易添加新的映射策略；

3、DispatcherServlet——>HandlerAdapter，HandlerAdapter 将会把处理器包装为适配器，从而支持多种类型的处理器，即适配器设计模式的应用，从而很容易支持很多类型的处理器；

4、HandlerAdapter——>处理器功能处理方法的调用，HandlerAdapter 将会根据适配的结果调用真正的处理器的功能处理方法，完成功能处理；并返回一个 ModelAndView 对象（包含模型数据、逻辑视图名）；

5、ModelAndView的逻辑视图名——> ViewResolver，ViewResolver 将把逻辑视图名解析为具体的View，通过这种策略模式，很容易更换其他视图技术；

6、View——>渲染，View会根据传进来的Model模型数据进行渲染，此处的Model实际是一个Map数

据结构，因此很容易支持其他视图技术；

7、返回控制权给DispatcherServlet，由DispatcherServlet返回响应给用户，到此一个流程结束。

下边两个组件通常情况下需要开发：

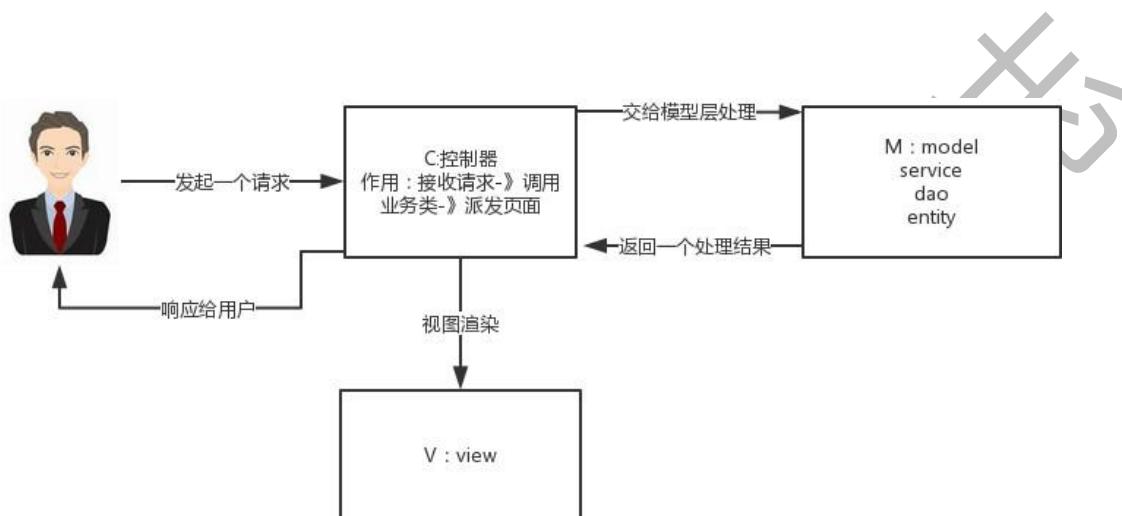
Handler：处理器，即后端控制器用controller表示。

View：视图，即展示给用户的界面，视图中通常需要标签语言展示模型数据。

在讲**SpringMVC**之前我们先来看一下什么是**MVC模式**

MVC：MVC是一种设计模

式MVC的原理图：



分析：

M-Model 模型（完成业务逻辑：有javaBean构成，service+dao+entity）

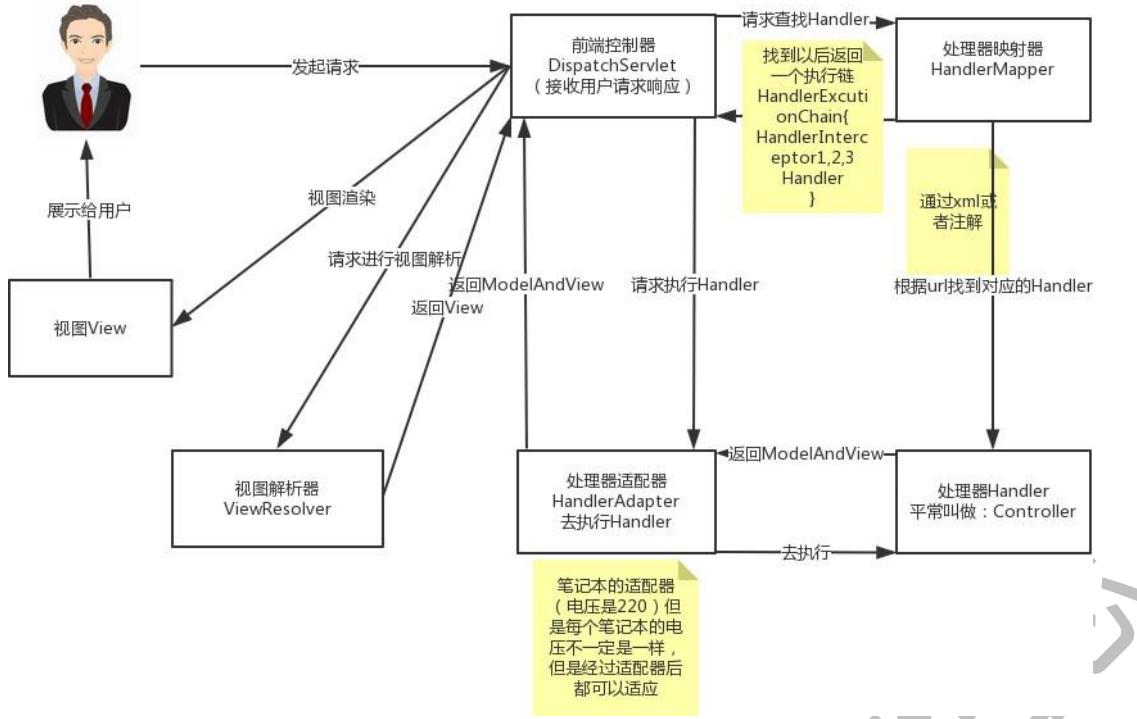
V-View 视图（做界面的展示 jsp, html.....）

C-Controller 控制器（接收请求—>调用模型—>根据结果派发页面）

springMVC是什么：

springMVC是一个MVC的开源框架，springMVC=struts2+spring，springMVC就相当于Struts2加上spring的整合，但是这里有一个疑惑就是，springMVC和spring是什么样的关系呢？这个在百度百科上有一个很好的解释：意思是说，springMVC是spring的一个后续产品，其实就是spring在原有基础上，又提供了web应用的MVC模块，可以简单的把springMVC理解为是spring的一个模块（类似AOP，IOC这样的模块），网络上经常会说springMVC和spring无缝集成，其实springMVC就是spring的一个子模块，所以根本不需要同spring进行整合。

SpringMVC的原理图：



看到这个图大家可能会有很多的疑惑，现在我们来看一下这个图的步骤：（可以对比MVC的原理图进行理解）

第一步：用户发起请求到前端控制器（DispatcherServlet）

第二步：前端控制器请求处理器映射器（HandlerMapping）去查找处理器（Handler）：通过xml配置或者注解进行查找

第三步：找到以后处理器映射器（HandlerMapping）像前端控制器返回执行链（HandlerExecutionChain）

第四步：前端控制器（DispatcherServlet）调用处理器适配器（HandlerAdapter）去执行处理器（Handler）

第五步：处理器适配器去执行Handler

第六步：Handler执行完给处理器适配器返回ModelAndView

第七步：处理器适配器向前端控制器返回ModelAndView

第八步：前端控制器请求视图解析器（ViewResolver）去进行视图解析

第九步：视图解析器像前端控制器返回View

第十步：前端控制器对视图进行渲染

第十一步：前端控制器向用户响应结果

看到这些步骤我相信大家很感觉非常的乱，这是正常的，但是这里主要是要大家理解springMVC中的几个组件：

前端控制器（DispatcherServlet）：接收请求，响应结果，相当于电脑的CPU。

处理器映射器（HandlerMapping）：根据URL去查找处理器

处理器（Handler）：（需要程序员去写代码处理逻辑的）

处理器适配器（HandlerAdapter）：会把处理器包装成适配器，这样就可以支持多种类型的处理器，类比笔记本的适配器（适配器模式的应用）

视图解析器（ViewResovler）：进行视图解析，多返回的字符串，进行处理，可以解析成对应的页面

6、SpringMVC怎么样设定重定向和转发的？

- (1) 转发：在返回值前面加"forward："，譬如"forward:user.do?name=method4"
- (2) 重定向：在返回值前面加"redirect："，譬如"redirect:<http://www.baidu.com>"

7、SpringMVC常用的注解有哪些？

@RequestMapping：用于处理请求 url 映射的注解，可用于类或方法上。用于类上，则表示类中的所有响应请求的方法都是以该地址作为父路径。

@RequestBody：注解实现接收http请求的json数据，将json转换为java对象。

@ResponseBody：注解实现将controller方法返回对象转化为json对象响应给客户。

8、Spring的AOP理解：

OOP面向对象，允许开发者定义纵向的关系，但不适用于定义横向的关系，导致了大量代码的重复，而不利于各个模块的重用。

AOP，一般称为面向切面，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证、日志、事务处理。

AOP实现的关键在于代理模式，AOP代理主要分为静态代理和动态代理。静态代理的代表为AspectJ；动态代理则以Spring AOP为代表。

(1) AspectJ是静态代理的增强，所谓静态代理，就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强，它会在编译阶段将AspectJ(切面)织入到Java字节码中，运行的时候就是增强之后的AOP对象。

(2) Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：

①JDK动态代理只提供接口的代理，不支持类的代理。核心InvocationHandler接口和Proxy类，InvocationHandler通过invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy利用InvocationHandler动态创建一个符合某一接口的实例，生成目标类的代理对象。

②如果代理类没有实现InvocationHandler接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

(3) 静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。

9、Spring的IOC理解

(1) IOC就是控制反转，是指创建对象的控制权的转移，以前创建对象的主动权和时机是由自己把控的，而现在这种权力转移到Spring容器中，并由容器根据配置文件去创建实例和管理各个实例之间的依赖关系，对象与对象之间松散耦合，也利于功能的复用。DI依赖注入，和控制反转是同一个概念的不同角度的描述，即 应用程序在运行时依赖IoC容器来动态注入对象需要的外部资源。

(2) 最直观的表达就是，IOC让对象的创建不用去new了，可以由spring自动生产，使用java的反射机制，根据配置文件在运行时动态的去创建对象以及管理对象，并调用对象的方法的。

(3) Spring的IOC有三种注入方式：构造器注入、setter方法注入、根据注解注入。

IoC让相互协作的组件保持松散的耦合，而AOP编程允许你把遍布于应用各层的功能分离出来形成可重用的功能组件。

10、解释一下spring bean的生命周期

首先说一下Servlet的生命周期：实例化，初始init，接收请求service，销毁destroy；

Spring上下文中的Bean生命周期也类似，如下：

(1) 实例化Bean：

对于BeanFactory容器，当客户向容器请求一个尚未初始化的bean时，或初始化bean的时候需要注入另一个尚未初始化的依赖时，容器就会调用createBean进行实例化。对于ApplicationContext容器，当容器启动结束后，通过获取BeanDefinition对象中的信息，实例化所有的bean。

(2) 设置对象属性（依赖注入）：

实例化后的对象被封装在BeanWrapper对象中，紧接着，Spring根据BeanDefinition中的信息 以及 通过BeanWrapper提供的设置属性的接口完成依赖注入。

(3) 处理Aware接口：

接着，Spring会检测该对象是否实现了xxxAware接口，并将相关的xxxAware实例注入给Bean：

①如果这个Bean已经实现了BeanNameAware接口，会调用它实现的setBeanName(String beanId)方法，此处传递的就是Spring配置文件中Bean的id值；

②如果这个Bean已经实现了BeanFactoryAware接口，会调用它实现的setBeanFactory()方法，传递的是Spring工厂自身。

③如果这个Bean已经实现了ApplicationContextAware接口，会调用setApplicationContext(ApplicationContext)方法，传入Spring上下文；

(4) BeanPostProcessor：

如果想对Bean进行一些自定义的处理，那么可以让Bean实现了BeanPostProcessor接口，那将会调用postProcessBeforeInitialization(Object obj, String s)方法。

(5) InitializingBean 与 init-method：

如果Bean在Spring配置文件中配置了 init-method 属性，则会自动调用其配置的初始化方法。

(6) 如果这个Bean实现了BeanPostProcessor接口，将会调用postProcessAfterInitialization(Object obj, String s)方法；由于这个方法是在Bean初始化结束时调用的，所以可以被应用于内存或缓存技术；

以上几个步骤完成后，Bean就已经被正确创建了，之后就可以使用这个Bean了。

(7) DisposableBean:

当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean这个接口，会调用其实现的destroy()方法；

(8) destroy-method:

最后，如果这个Bean的Spring配置中配置了destroy-method属性，会自动调用其配置的销毁方法。

11、解释Spring支持的几种bean的作用域。

Spring容器中的bean可以分为5个范围：

- (1) singleton：默认，每个容器中只有一个bean的实例，单例的模式由BeanFactory自身来维护。
- (2) prototype：为每一个bean请求提供一个实例。
- (3) request：为每一个网络请求创建一个实例，在请求完成以后，bean会失效并被垃圾回收器回收。
- (4) session：与request范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。
- (5) global-session：全局作用域，global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet共用全局的存储变量的话，那么这全局变量需要存储在global-session中。全局作用域与Servlet中的session作用域效果相同。

12、Spring基于xml注入bean的几种方式：

- (1) Set方法注入；
- (2) 构造器注入：①通过index设置参数的位置；②通过type设置参数类型；
- (3) 静态工厂注入；
- (4) 实例工厂；

详细内容可以阅读：<https://blog.csdn.net/a745233700/article/details/89307518>

13、Spring框架中都用到了哪些设计模式？

- (1) 工厂模式：BeanFactory就是简单工厂模式的体现，用来创建对象的实例；
- (2) 单例模式：Bean默认为单例模式。
- (3) 代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术；
- (4) 模板方法：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。
- (5) 观察者模式：定义对象间一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知并更新，如Spring中listener的实现--ApplicationListener。

MyBatis篇

1、什么是MyBatis

(1) Mybatis是一个半ORM（对象关系映射）框架，它内部封装了JDBC，开发时只需要关注SQL语句本身，不需要花费精力去处理加载驱动、创建连接、创建statement等繁杂的过程。程序员直接编写原生sql，可以严格控制sql执行性能，灵活度高。

(2) MyBatis 可以使用 XML 或注解来配置和映射原生信息，将 POJO 映射成数据库中的记录，避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。

(3) 通过xml 文件或注解的方式将要执行的各种 statement 配置起来，并通过java对象和 statement 中sql的动态参数进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射为java对象并返回。（从执行sql到返回result的过程）。

2、MyBatis的优点和缺点

优点：

(1) 基于SQL语句编程，相当灵活，不会对应用程序或者数据库的现有设计造成任何影响，SQL写在XML里，解除sql与程序代码的耦合，便于统一管理；提供XML标签，支持编写动态SQL语句，并可重用。

(2) 与JDBC相比，减少了50%以上的代码量，消除了JDBC大量冗余的代码，不需要手动开关连接；

(3) 很好的与各种数据库兼容（因为MyBatis使用JDBC来连接数据库，所以只要JDBC支持的数据库MyBatis都支持）。

(4) 能够与Spring很好的集成；

(5) 提供映射标签，支持对象与数据库的ORM字段关系映射；提供对象关系映射标签，支持对象关系组件维护。

缺点

(1) SQL语句的编写工作量较大，尤其当字段多、关联表多时，对开发人员编写SQL语句的功底有一定要求。

(2) SQL语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。

3、#{}和\${}的区别是什么？

#{}是预编译处理，\${}是字符串替换。

Mybatis在处理#{}时，会将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值；

Mybatis在处理\${}时，就是把\${}替换成变量的值。

使用#{}可以有效的防止SQL注入，提高系统安全性。

4、当实体类中的属性名和表中的字段名不一样，怎么办？

第1种：通过在查询的sql语句中定义字段名的别名，让字段名的别名和实体类的属性名一致。

```
<select id="selectorder" parameterType="int"
resultType="me.gacl.domain.order">
    select order_id id, order_no orderno ,order_price price from orders where
    order_id=#{id};
</select>
```

第2种：通过来映射字段名和实体类属性名的一一对应的关系。

```
<select id="getOrder" parameterType="int" resultMap="orderresultmap">
    select * from orders where order_id=#{id}
</select>

<resultMap type="me.gacl.domain.order" id="orderresultmap">
    <!--用id属性来映射主键字段-->
    <id property="id" column="order_id">

    <!--用result属性来映射非主键字段，property为实体类属性名，column为数据表中的属性-->
    <result property="orderno" column="order_no"/>
    <result property="price" column="order_price" />
</resultMap>
```

5、Mybatis是如何进行分页的？分页插件的原理是什么？

Mybatis使用RowBounds对象进行分页，它是针对ResultSet结果集执行的内存分页，而非物理分页。可以在sql内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。

分页插件的基本原理是使用Mybatis提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的sql，然后重写sql，根据dialect方言，添加对应的物理分页语句和物理分页参数。

6、Mybatis是如何将sql执行结果封装为目标对象并返回的？都有哪些映射形式？

第一种是使用标签，逐一定义数据库列名和对象属性名之间的映射关系。第

二种是使用sql列的别名功能，将列的别名书写为对象属性名。

有了列名与属性名的映射关系后，Mybatis通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。

7、如何执行批量插入？

首先，创建一个简单的insert语句：

然后在java代码中像下面这样执行批处理插入：

```
<insert id="insertname">
    insert into names (name) values (#{value})
</insert>
```

```
list<string> names = new arrayList();
names.add("fred");
names.add("barney");
names.add("betty");
names.add("wilma");

// 注意这里 executorType.Batch
sqlSession sqlSession =
sqlSessionFactory.openSession(executorType.Batch); try {
namemapper mapper =
sqlSession.getMapper(namemapper.class); for (String
name : names) {
    mapper.insertName(name);
}
sqlSession.commit();
} catch(Exce
ption
e){ e.print
StackTrace
();sqlSess
ion.rollba
ck();
throw e;
}
finally {
    sqlSession.close();
}
```



8、Xml映射文件中，除了常见的select|insert|update|delete标 签之外，还有哪些标签？

、、、、，加上动态sql的9个标签，其中为sql片段标签，通过标签引入sql片段，为不支持自增的主键生成策略标签。

9、MyBatis实现一对一有几种方式?具体怎么操作的?

有联合查询和嵌套查询,联合查询是几个表联合查询,只查询一次,通过在resultMap里面配置association节点配置一对一的类就可以完成;

嵌套查询是先查一个表,根据这个表里面的结果的外键id,去再另外一个表里面查询数据,也是通过association配置,但另外一个表的查询通过select属性配置。

10、Mybatis是否支持延迟加载？如果支持，它的实现原理是什么？

Mybatis仅支持association关联对象和collection关联集合对象的延迟加载，association指的就是一对一，collection指的就是一对多查询。在Mybatis配置文件中，可以配置是否启用延迟加载lazyLoadingEnabled=true|false。

它的原理是，使用CGLIB创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用a.getB().getName()，拦截器invoke()方法发现a.getB()是null值，那么就会单独发送事先保存好的查询关联B对象的sql，把B查询上来，然后调用a.setB(b)，于是a的对象b属性就有值了，接着完成a.getB().getName()方法的调用。这就是延迟加载的基本原理。

当然了，不光是Mybatis，几乎所有的包括Hibernate，支持延迟加载的原理都是一样的。

11、Mybatis的一级、二级缓存：

- 1) 一级缓存:基于PerpetualCache的HashMap本地缓存，其存储作用域为Session，当Session flush或close之后，该Session中的所有Cache就将清空，默认打开一级缓存。
- 2) 二级缓存与一级缓存其机制相同，默认也是采用PerpetualCache，HashMap存储，不同在于其存储作用域为Mapper(Namespace)，并且可自定义存储源，如Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现Serializable序列化接口(可用来保存对象的状态),可在它的映射文件中配置；



关注【程序员生活志】
免费获取更多海量资源

3) 对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存Namespaces)的进行了C/U/D操作后，默认该作用域下所有 select 中的缓存将被 clear 掉并重新更新，如果开启了二级缓存，则只根据配置判断是否刷新。

SpringBoot篇

1、什么是SpringBoot? 为什么要用SpringBoot

用来简化spring应用的初始搭建以及开发过程 使用特定的方式来进行配置 (properties或yml文件)

创建独立的spring引用程序 main方法运行

嵌入的Tomcat 无需部署war文件

简化maven配置

自动配置spring添加对应功能starter自动化配置

spring boot来简化**spring**应用开发，约定大于配置，去繁从简，**just run**就能创建一个独立的，产品级别的应用

Spring Boot 优点非常多，

如：一、独立运行

Spring Boot而且内嵌了各种servlet容器，Tomcat、Jetty等，现在不再需要打成war包部署到容器中，Spring Boot只要打成一个可执行的jar包就能独立运行，所有的依赖包都在一个jar包内。

二、简化配置

spring-boot-starter-web启动器自动依赖其他组件，简少了maven的配置。

三、自动配置

Spring Boot能根据当前类路径下的类、jar包来自动配置bean，如添加一个spring-boot-starter-web启动器就能拥有web的功能，无需其他配置。

四、无代码生成和XML配置

Spring Boot配置过程中无代码生成，也无需XML配置文件就能完成所有配置工作，这一切都是借助于条件注解完成的，这也是Spring4.x的核心功能之一。

五、应用监控

Spring Boot提供一系列端点可以监控服务及应用，做健康检测。

2、Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的？

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下3个注解：

@SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。

@EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。

@ComponentScan: Spring组件扫描。

3、运行Spring Boot有哪几种方式？

- 1) 打包用命令或者放到容器中运行
- 2) 用 Maven/Gradle 插件运行
- 3) 直接执行 main 方法运行

4、如何理解 Spring Boot 中的 Starters？

Starters是什么：

Starters可以理解为启动器，它包含了一系列可以集成到应用里面的依赖包，你可以一站式集成Spring及其他技术，而不需要到处找示例代码和依赖包。如你想使用Spring JPA访问数据库，只要加入spring-boot-starter-data-jpa启动器依赖就能使用了。Starters包含了许多项目中需要用到的依赖，它们能快速持续的运行，都是一系列得到支持的管理传递性依赖。

Starters命名：

Spring Boot官方的启动器都是以spring-boot-starter-命名的，代表了一个特定的应用类型。第三方的启动器不能以spring-boot开头命名，它们都被Spring Boot官方保留。一般一个第三方的应该这样命名，像mybatis的mybatis-spring-boot-starter。

Starters分类：

1. Spring Boot应用类启动器

启动器名称	功能描述
spring-boot-starter	包含自动配置、日志、YAML的支持。
spring-boot-starter-web	使用Spring MVC构建web 工程，包含restful，默认使用Tomcat容器。
...	...

https://blog.csdn.net/Kevin_Gu8

2. Spring Boot生产启动器

启动器名称	功能描述
spring-boot-starter-actuator	提供生产环境特性，能监控管理应用。

3. Spring Boot技术类启动器

启动器名称	功能描述
spring-boot-starter-json	提供对JSON的读写支持。
spring-boot-starter-logging	默认的日志启动器，默认使用Logback。
...	...

4. 其他第三方启动器

5、如何在**Spring Boot**启动的时候运行一些特定的代码？

如果你想在Spring Boot启动的时候运行一些特定的代码，你可以实现接口**ApplicationRunner**或者**CommandLineRunner**，这两个接口实现方式一样，它们都只提供了一个run方法。

CommandLineRunner: 启动获取命令行参数

6、**Spring Boot**需要独立的容器运行吗？

可以不需要，内置了Tomcat/Jetty等容器。

7、**Spring Boot**中的监视器是什么？

Spring boot actuator是spring启动框架中的重要功能之一。Spring boot监视器可帮助您访问生产环境中正在运行的应用程序的当前状态。有几个指标必须在生产环境中进行检查和监控。即使一些外部应用程序可能正在使用这些服务来向相关人员触发警报消息。监视器模块公开了一组可直接作为HTTP URL访问的REST端点来检查状态。

8、如何使用**Spring Boot**实现异常处理？

Spring提供了一种使用ControllerAdvice处理异常的非常有用的方法。我们通过实现一个ControllerAdvice类，来处理控制器类抛出的所有异常。

9、你如何理解**Spring Boot**中的**Starters**？

Starters可以理解为启动器，它包含了一系列可以集成到应用里面的依赖包，你可以一站式集成Spring及其他技术，而不需要到处找示例代码和依赖包。如你想使用Spring JPA访问数据库，只要加入spring-boot-starter-data-jpa启动器依赖就能使用了。

10、**springboot**常用的**starter**有哪些

spring-boot-starter-web 嵌入tomcat和web开发需要servlet与jsp支持

spring-boot-starter-data-jpa 数据库支持

spring-boot-starter-data-redis redis数据库支持

spring-boot-starter-data-solr solr支持

mybatis-spring-boot-starter 第三方的mybatis集成starter

11、**SpringBoot**实现热部署有哪几种方式？

主要有两种方式：

- Spring Loaded
- Spring-boot-devtools

12、如何理解 Spring Boot 配置加载顺序？

在 Spring Boot 里面，可以使用以下几种方式来加载配置。

- 1) properties文件；
- 2) YAML文件；
- 3) 系统环境变量；
- 4) 命令行参数；

等等.....

13、Spring Boot 的核心配置文件有哪几个？它们的区别是什么？

Spring Boot 的核心配置文件是 application 和 bootstrap 配置文件。

application 配置文件这个容易理解，主要用于 Spring Boot 项目的自动化配置。

bootstrap 配置文件有以下几个应用场景。

- 使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中添加连接到配置中心的配置属性来加载外部配置中心的配置信息；
- 一些固定的不能被覆盖的属性；
- 一些加密/解密的场景；

14、如何集成 Spring Boot 和 ActiveMQ？

对于集成 Spring Boot 和 ActiveMQ，我们使

用spring-boot-starter-activemq

依赖关系。它只需要很少的配置，并且不需要样板代码。

15、如何重新加载Spring Boot上的更改，而无需重新启动服务器？

这可以使用DEV工具来实现。通过这种依赖关系，您可以节省任何更改，嵌入式tomcat将重新启动。

Spring Boot有一个开发工具（DevTools）模块，它有助于提高开发人员的生产力。Java开发人员面临的一个主要挑战是将文件更改自动部署到服务器并自动重启服务器。

开发人员可以重新加载Spring Boot上的更改，而无需重新启动服务器。这将消除每次手动部署更改的需要。Spring Boot在发布它的第一个版本时没有这个功能。

这是开发人员最需要的功能。DevTools模块完全满足开发人员的需求。该模块将在生产环境中被禁用。它还提供H2数据库控制台以更好地测试应用程序。

org.springframework.boot

spring-boot-devtools

true

16、Spring Boot、Spring MVC 和 Spring 有什么区别？

1、Spring

Spring最重要的特征是依赖注入。所有 SpringModules 不是依赖注入就是 IOC 控制反转。

当我们恰当的使用 DI 或者是 IOC 的时候，我们可以开发松耦合应用。松耦合应用的单元测试可以很容易的进行。

2、Spring MVC

Spring MVC 提供了一种分离式的方法来开发 Web 应用。通过运用像 DispatcherServlet, MoudlAndView 和 ViewResolver 等一些简单的概念，开发 Web 应用将会变的非常简单。

3、SpringBoot

Spring 和 SpringMVC 的问题在于需要配置大量的参数。

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix">
        <value>/WEB-INF/views/</value>
    </property>
    <property name="suffix">
        <value>.jsp</value>
    </property>
</bean>

<mvc:resources mapping="/webjars/**" location="/webjars/" />
```

https://blog.csdn.net/Dome_

Spring Boot 通过一个自动配置和启动的项来解决这个问题。为了更快的构建产品就绪应用程序，Spring Boot 提供了一些非功能性特征。

17、能否举一个例子来解释更多 Staters 的内容？

让我们来思考一个 Stater 的例子 -Spring Boot Stater Web。

如果你想开发一个 web 应用程序或者是公开 REST 服务的应用程序。Spring Boot Start Web 是首选。让我们使用 Spring Initializr 创建一个 Spring Boot Start Web 的快速项目。

Spring Boot Start Web 的依赖项

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

下面的截图是添加进我们应用程序的不同的依赖项

Maven Dependencies

- ▶ `spring-boot-starter-web-1.4.4.RELEASE.jar` - /Users/rangaraokaranam/.m2/repository/org/springframework/boot/starter-web/1.4.4.RELEASE/spring-boot-starter-web-1.4.4.RELEASE.jar
- ▶ `spring-boot-starter-1.4.4.RELEASE.jar` - /Users/rangaraokaranam/.m2/repository/org/springframework/boot/starter/1.4.4.RELEASE/spring-boot-starter-1.4.4.RELEASE.jar
- ▶ `spring-boot-1.4.4.RELEASE.jar` - /Users/rangaraokaranam/.m2/repository/org/springframework/boot/1.4.4.RELEASE/spring-boot-1.4.4.RELEASE.jar
- ▶ `spring-boot-autoconfigure-1.4.4.RELEASE.jar` - /Users/rangaraokaranam/.m2/repository/org/springframework/boot/autoconfigure/spring-boot-autoconfigure-1.4.4.RELEASE.jar
- ▶ `spring-boot-starter-logging-1.4.4.RELEASE.jar` - /Users/rangaraokaranam/.m2/repository/org/springframework/boot/starter/logging/spring-boot-starter-logging-1.4.4.RELEASE.jar
- ▶ `logback-classic-1.1.9.jar` - /Users/rangaraokaranam/.m2/repository/ch/qos/logback/logback-classic/1.1.9/logback-classic-1.1.9.jar
- ▶ `logback-core-1.1.9.jar` - /Users/rangaraokaranam/.m2/repository/ch/qos/logback/logback-core/1.1.9/logback-core-1.1.9.jar
- ▶ `slf4j-api-1.7.22.jar` - /Users/rangaraokaranam/.m2/repository/org/slf4j/slf4j-api/1.7.22/slf4j-api-1.7.22.jar
- ▶ `jcl-over-slf4j-1.7.22.jar` - /Users/rangaraokaranam/.m2/repository/org/slf4j/jcl-over-slf4j/1.7.22/jcl-over-slf4j-1.7.22.jar
- ▶ `jul-to-slf4j-1.7.22.jar` - /Users/rangaraokaranam/.m2/repository/org/slf4j/jul-to-slf4j/1.7.22/jul-to-slf4j-1.7.22.jar
- ▶ `log4j-over-slf4j-1.7.22.jar` - /Users/rangaraokaranam/.m2/repository/org/slf4j/log4j-over-slf4j/1.7.22/log4j-over-slf4j-1.7.22.jar
- ▶ `spring-core-4.3.6.RELEASE.jar` - /Users/rangaraokaranam/.m2/repository/org/springframework/core/spring-core-4.3.6.RELEASE.jar
- ▶ `snakeyaml-1.17.jar` - /Users/rangaraokaranam/.m2/repository/org/yaml/snakeyaml/1.17/snakeyaml-1.17.jar
- ▶ `spring-boot-starter-tomcat-1.4.4.RELEASE.jar` - /Users/rangaraokaranam/.m2/repository/org/springframework/boot/starter/tomcat/spring-boot-starter-tomcat-1.4.4.RELEASE.jar
- ▶ `tomcat-embed-core-8.5.11.jar` - /Users/rangaraokaranam/.m2/repository/org/apache/tomcat/embed/tomcat-embed-core/8.5.11/tomcat-embed-core-8.5.11.jar
- ▶ `tomcat-embed-el-8.5.11.jar` - /Users/rangaraokaranam/.m2/repository/org/apache/tomcat/embed/tomcat-embed-el/8.5.11/tomcat-embed-el-8.5.11.jar
- ▶ `tomcat-embed-websocket-8.5.11.jar` - /Users/rangaraokaranam/.m2/repository/org/apache/tomcat/embed/tomcat-embed-websocket/8.5.11/tomcat-embed-websocket-8.5.11.jar
- ▶ `hibernate-validator-5.2.4.Final.jar` - /Users/rangaraokaranam/.m2/repository/org/hibernate/validator/hibernate-validator/5.2.4.Final/hibernate-validator-5.2.4.Final.jar
- ▶ `validation-api-1.1.0.Final.jar` - /Users/rangaraokaranam/.m2/repository/javax/validation/validation-api/1.1.0.Final/validation-api-1.1.0.Final.jar
- ▶ `jboss-logging-3.3.0.Final.jar` - /Users/rangaraokaranam/.m2/repository/org/jboss/logging/jboss-logging/3.3.0.Final/jboss-logging-3.3.0.Final.jar
- ▶ `classmate-1.3.3.jar` - /Users/rangaraokaranam/.m2/repository/com/fasterxml/classmate/1.3.3/classmate-1.3.3.jar
- ▶ `jackson-databind-2.8.6.jar` - /Users/rangaraokaranam/.m2/repository/com/fasterxml/jackson/databind/jackson-databind-2.8.6.jar
- ▶ `jackson-annotations-2.8.6.jar` - /Users/rangaraokaranam/.m2/repository/com/fasterxml/jackson/annotations/jackson-annotations-2.8.6.jar
- ▶ `jackson-core-2.8.6.jar` - /Users/rangaraokaranam/.m2/repository/com/fasterxml/jackson/core/jackson-core-2.8.6.jar
- ▶ `spring-web-4.3.6.RELEASE.jar` - /Users/rangaraokaranam/.m2/repository/org/springframework/web/spring-web-4.3.6.RELEASE.jar
- ▶ `spring-aop-4.3.6.RELEASE.jar` - /Users/rangaraokaranam/.m2/repository/org/springframework/aop/spring-aop-4.3.6.RELEASE.jar
- ▶ `spring-beans-4.3.6.RELEASE.jar` - /Users/rangaraokaranam/.m2/repository/org/springframework/beans/spring-beans-4.3.6.RELEASE.jar
- ▶ `spring-context-4.3.6.RELEASE.jar` - /Users/rangaraokaranam/.m2/repository/org/springframework/context/spring-context-4.3.6.RELEASE.jar
- ▶ `spring-webmvc-4.3.6.RELEASE.jar` - /Users/rangaraokaranam/.m2/repository/org/springframework/webmvc/spring-webmvc-4.3.6.RELEASE.jar
- ▶ `spring-expression-4.3.6.RELEASE.jar` - /Users/rangaraokaranam/.m2/repository/org/springframework/expression/spring-expression-4.3.6.RELEASE.jar

https://blog.csdn.net/Dorne_

依赖项可以被分为：

- Spring - core, beans, context, aop
- Web MVC - (Spring MVC)
- Jackson - for JSON Binding
- Validation - Hibernate,Validation API
- Embedded Servlet Container -
- Tomcat Logging - logback,slf4j

任何经典的 Web 应用程序都会使用所有这些依赖项。Spring Boot Starter Web 预先打包了这些依赖项。

作为一个开发者，我不需要再担心这些依赖项和它们的兼容版本。

18、Spring Boot 还提供了其它的哪些 Starter Project Options?

Spring Boot 也提供了其它的启动器项目包括，包括用于开发特定类型应用程序的典型依赖项。

- `spring-boot-starter-web-services` - SOAP Web Services;
- `spring-boot-starter-web` - Web 和 RESTful 应用程序；
- `spring-boot-starter-test` - 单元测试和集成测试；
- `spring-boot-starter-jdbc` - 传统的 JDBC；
- `spring-boot-starter-hateoas` - 为服务添加 HATEOAS 功能；
- `spring-boot-starter-security` - 使用 SpringSecurity 进行身份验证和授权；
- `spring-boot-starter-data-jpa` - 带有 Hibernate 的 Spring Data JPA；
- `spring-boot-starter-data-rest` - 使用 Spring Data REST 公布简单的 REST 服务；

MySQL篇

1、数据库的三范式是什么

第一范式：列不可再分

第二范式：行可以唯一区分，主键约束

第三范式：表的非主属性不能依赖与其他表的非主属性 外键约束

且三大范式是一级一级依赖的，第二范式建立在第一范式上，第三范式建立第一第二范式上。

2、数据库引擎有哪些

如何查看mysql提供的所有存储引擎

```
mysql> show engines;
```

The screenshot shows the MySQL 5.7 Command Line Client interface. The command 'show engines;' is entered, and the output displays various storage engines with their support status and transaction characteristics.

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

mysql常用引擎包括：MYISAM、Innodb、Memory、MERGE

- MYISAM：全表锁，拥有较高的执行速度，不支持事务，不支持外键，并发性能差，占用空间相对较小，对事务完整性没有要求，以select、insert为主的应用基本上可以使用这引擎
- Innodb：行级锁，提供了具有提交、回滚和崩溃恢复能力的事务安全，支持自动增长列，支持外键约束，并发能力强，占用空间是MYISAM的2.5倍，处理效率相对会差一些
- Memory：全表锁，存储在内存中，速度快，但会占用和数据量成正比的内存空间且数据在mysql重启时会丢失，默认使用HASH索引，检索效率非常高，但不适用于精确查找，主要用于那些内容变化不频繁的代码表
- MERGE：是一组MYISAM表的组合

3、InnoDB与MyISAM的区别

1. InnoDB支持事务，MyISAM不支持，对于InnoDB每一条SQL语言都默认封装成事务，自动提交，这样会影响速度，所以最好把多条SQL语言放在begin和commit之间，组成一个事务；
2. InnoDB支持外键，而MyISAM不支持。对一个包含外键的InnoDB表转为MYISAM会失败；
3. InnoDB是聚集索引，数据文件是和索引绑在一起的，必须要有主键，通过主键索引效率很高。但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，因为主键太大，其他索引也都会很大。而MyISAM是非聚集索引，数据文件是分离的，索引保存的是数据文件的指针。主键索引和辅助索引是独立的。
4. InnoDB不保存表的具体行数，执行select count(*) from table时需要全表扫描。而MyISAM用一个变量保存了整个表的行数，执行上述语句时只需要读出该变量即可，速度很快；
5. Innodb不支持全文索引，而MyISAM支持全文索引，查询效率上MyISAM要高；

如何选择引擎？

如果没有特别的需求，使用默认的Innodb即可。

MyISAM：以读写插入为主的应用程序，比如博客系统、新闻门户网站。

Innodb：更新（删除）操作频率也高，或者要保证数据的完整性；并发量高，支持事务和外键。比如OA自动化办公系统。

4、数据库的事务

什么是事务？：多条sql语句，要么全部成功，要么全部失败。

事务的特性：

数据库事务特性：原子性(Atomic)、一致性(Consistency)、隔离性(Isolation)、持久性(Durability)。简称ACID。

- 原子性：组成一个事务的多个数据库操作是一个不可分割的原子单元，只有所有操作都成功，整个事务才会提交。任何一个操作失败，已经执行的任何操作都必须撤销，让数据库返回初始状态。
- 一致性：事务操作成功后，数据库所处的状态和它的业务规则是一致的。即数据不会被破坏。如A转账100元给B，不管操作是否成功，A和B的账户总额是不变的。
- 隔离性：在并发数据操作时，不同的事务拥有各自的数据空间，它们的操作不会对彼此产生干扰。
- 持久性：一旦事务提交成功，事务中的所有操作都必须持久化到数据库中。

5、索引问题

索引是对数据库表中一个或多个列的值进行排序的结构，建立索引有助于快速获取信息。

你也可以这样理解：索引就是加快检索表中数据的方法。数据库的索引类似于书籍的索引。在书籍中，索引允许用户不必翻阅完整个书就能迅速地找到所需要的信息。在数据库中，索引也允许数据库程序迅速地找到表中的数据，而不必扫描整个数据库。

mysql有4种不同的索引：

- 主键索引 (PRIMARY)

数据列不允许重复，不允许为NULL，一个表只能有一个主键。

- 唯一索引 (UNIQUE)

数据列不允许重复，允许为NULL值，一个表允许多个列创建唯一索引。

- 可以通过 `ALTER TABLE table_name ADD UNIQUE (column);` 创建唯一索引
- 可以通过 `ALTER TABLE table_name ADD UNIQUE (column1,column2);` 创建唯一组合索引

- 普通索引 (INDEX)

- 可以通过 `ALTER TABLE table_name ADD INDEX index_name(column);` 创建普通索引
- 可以通过 `ALTER TABLE table_name ADD INDEX index_name(column1, column2, column3);` 创建组合索引

- 全文索引 (FULLTEXT)

可以通过 `ALTER TABLE table_name ADD FULLTEXT (column);` 创建全文索引

索引并非是越多越好，创建索引也需要耗费资源，一是增加了数据库的存储空间，二是在插入和删除时要花费较多的时间维护索引。

- 索引加快数据库的检索速度
- 索引降低了插入、删除、修改等维护任务的速度
- 唯一索引可以确保每一行数据的唯一性
- 通过使用索引，可以在查询的过程中使用优化器，提高系统的性能
- 索引需要占物理和数据空间

6、SQL优化

- 1、查询语句中不要使用select *
- 2、尽量减少子查询，使用关联查询（left join,right join,inner join）替代
- 3、减少使用IN或者NOT IN ,使用exists, not exists或者关联查询语句替代
- 4、or 的查询尽量用 union或者union all 替代(在确认没有重复数据或者不用剔除重复数据时，union all会更好)
- 5、应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。
- 6、应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如： select id from t where num is null 可以在num上设置默认值0，确保表中num列没有null值，然后这样查询： select id from t where num=0

7、简单说一说drop、 delete与truncate的区别

SQL中的drop、 delete、 truncate都表示删除，但是三者有一些差别

delete和truncate只删除表的数据不删除表的结构

速度,一般来说: drop> truncate >delete

delete语句是dml,这个操作会放到rollback segment中,事务提交之后才生效;

如果有相应的trigger,执行的时候将被触发. truncate,drop是ddl, 操作立即生效,原数据不放到rollback segment中,不能回滚. 操作不触发trigger.

8、什么是视图

视图是一种虚拟的表，具有和物理表相同的功能。可以对视图进行增，改，查，操作，试图通常是一个表或者多个表的行或列的子集。对视图的修改不影响基本表。它使得我们获取数据更容易，相比多表查询。

9、什么是内联接、左外联接、右外联接？

- 内联接（Inner Join）：匹配2张表中相关联的记录。
- 左外联接（Left Outer Join）：除了匹配2张表中相关联的记录外，还会匹配左表中剩余的记录，右表中未匹配到的字段用NULL表示。
- 右外联接（Right Outer Join）：除了匹配2张表中相关联的记录外，还会匹配右表中剩余的记录，左表中未匹配到的字段用NULL表示。在判定左表和右表时，要根据表名出现在Outer Join的左右位置关系。

10、并发事务带来哪些问题？

在典型的应用程序中，多个事务并发运行，经常会操作相同的数据来完成各自的任务（多个用户对同一数据进行操作）。并发虽然是必须的，但可能会导致以下的问题。

- 脏读（**Dirty read**）：当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。
- 丢失修改（**Lost to modify**）：指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。例如：事务1读取某表中的数据A=20，事务2也读取A=20，事务1修改A=A-1，事务2也修改A=A-1，最终结果A=19，事务1的修改被丢失。
- 不可重复读（**Unrepeatable read**）：指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可重复读。
- 幻读（**Phantom read**）：幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

不可重复读和幻读区别：

不可重复读的重点是修改比如多次读取一条记录发现其中某些列的值被修改，幻读的重点在于新增或者删除比如多次读取一条记录发现记录增多或减少了。

11、事务隔离级别有哪些？MySQL的默认隔离级别是？

SQL 标准定义了四个隔离级别：

- **READ-UNCOMMITTED(读取未提交)**：最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读。
- **READ-COMMITTED(读取已提交)**：允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生。
- **REPEATABLE-READ(可重复读)**：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。
- **SERIALIZABLE(可串行化)**：最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。

隔离级别	脏读	不可重复读	幻影读
READ-UNCOMMITTED	√	√	√
READ-COMMITTED	✗	√	√
REPEATABLE-READ	✗	✗	✗
SERIALIZABLE	✗	✗	✗

MySQL InnoDB 存储引擎的默认支持的隔离级别是 **REPEATABLE-READ**（可重读）。我们可以通过 `SELECT @@tx_isolation;` 命令来查看

```
mysql> SELECT @@tx_isolation;
+-----+
| @@tx_isolation |
+-----+
| REPEATABLE-READ |
+-----+
```

这里需要注意的是：与 SQL 标准不同的地方在于 InnoDB 存储引擎在 **REPEATABLE-READ**（可重读）事务隔离级别下使用的是 Next-Key Lock 锁算法，因此可以避免幻读的产生，这与其他数据库系统(如 SQL Server)是不同的。所以说InnoDB 存储引擎默认支持的隔离级别是 **REPEATABLE-READ**（可重读）已经可以完全保证事务的隔离性要求，即达到了 SQL 标准的 **SERIALIZABLE**(可串行化) 隔离级别。因为隔离级别越低，事务请求的锁越少，所以大部分数据库系统的隔离级别都是 **READ-COMMITTED**(读取提交内容)，但是你要知道的是InnoDB 存储引擎默认使用 **REPEATABLE-READ**（可重读）并不会有任何性能损失。

InnoDB 存储引擎在 分布式事务 的情况下一般会用到 **SERIALIZABLE**(可串行化) 隔离级别。

12、大表如何优化？

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

1. 限定数据的范围

务必禁止不带任何限制数据范围条件的查询语句。比如：我们当用户在查询订单历史的时候，我们可以控制在一个月的范围内；

2. 读/写分离

经典的数据库拆分方案，主库负责写，从库负责读；

3. 垂直分区

根据数据库里面数据表的相关性进行拆分。例如，用户表中既有用户的登录信息又有用户的基本信息，可以将用户表拆分成两个单独的表，甚至放到单独的库做分库。

简单来说垂直拆分是指数据表列的拆分，把一张列比较多的表拆分为多张表。如下图所示，这样来说大家应该就更容易理解了。

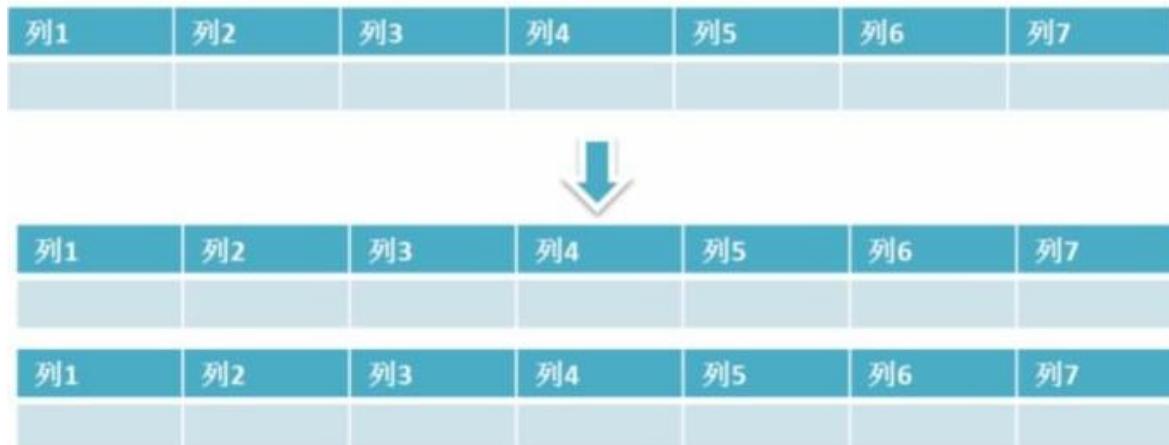


- 垂直拆分的优点： 可以使得列数据变小，在查询时减少读取的Block数，减少I/O次数。此外，垂直分区可以简化表的结构，易于维护。
- 垂直拆分的缺点： 主键会出现冗余，需要管理冗余列，并会引起Join操作，可以通过在应用层进行Join来解决。此外，垂直分区会让事务变得更加复杂；

4. 水平分区

保持数据表结构不变，通过某种策略存储数据分片。这样每一片数据分散到不同的表或者库中，达到了分布式的目的。水平拆分可以支撑非常大的数据量。

水平拆分是指数据表行的拆分，表的行数超过200万行时，就会变慢，这时可以把一张的表的数据拆成多张表来存放。举个例子：我们可以将用户信息表拆分成多个用户信息表，这样就可以避免单一表数据量过大对性能造成影响。



水平拆分可以支持非常大的数据量。需要注意的一点是：分表仅仅是解决了单一表数据过大的问题，但由于表的数据还是在同一台机器上，其实对于提升MySQL并发能力没有什么意义，所以 水平拆分最好分库。

水平拆分能够 支持非常大的数据量存储，应用端改造也少，但 分片事务难以解决，跨节点Join性能较差，逻辑复杂。《Java工程师修炼之道》的作者推荐 尽量不要对数据进行分片，因为拆分会带来逻辑、部署、运维的各种复杂度 ，一般的数据表在优化得当的情况下支撑千万以下的数据量是没有太大问题的。如果实在要分片，尽量选择客户端分片架构，这样可以减少一次和中间件的网络I/O。

下面补充一下数据库分片的两种常见方案：

- 客户端代理：分片逻辑在应用端，封装在jar包中，通过修改或者封装JDBC层来实现。当当网的 **Sharding-JDBC** 、阿里的TDDL是两种比较常用的实现。
- 中间件代理：在应用和数据中间加了一个代理层。分片逻辑统一维护在中间件服务中。我们现在谈的 **Mycat** 、360的Atlas、网易的DDB等等都是这种架构的实现。

详细内容可以参考： MySQL大表优化方案: <https://segmentfault.com/a/1190000006158186>

13、分库分表之后,id 主键如何处理？

因为要是分成多个表之后，每个表都是从 1 开始累加，这样是不对的，我们需要一个全局唯一的 id 来支持。

生成全局 id 有下面这几种方式：

- **UUID**: 不适合作为主键，因为太长了，并且无序不可读，查询效率低。比较适合用于生成唯一的名字的标示比如文件的名字。
- 数据库自增 **id** : 两台数据库分别设置不同步长，生成不重复ID的策略来实现高可用。这种方式生成的 id 有序，但是需要独立部署数据库实例，成本高，还会有性能瓶颈。
- 利用 **redis** 生成 **id** : 性能比较好，灵活方便，不依赖于数据库。但是，引入了新的组件造成系统更加复杂，可用性降低，编码更加复杂，增加了系统成本。
- Twitter的**snowflake**算法 : Github 地址: <https://github.com/twitter-archive/snowflake>。
- 美团的**Leaf**分布式ID生成系统 : Leaf 是美团开源的分布式ID生成器，能保证全局唯一性、趋势递增、单调递增、信息安全，里面也提到了几种分布式方案的对比，但也需要依赖关系数据库、Zookeeper等中间件。感觉还不错。美团技术团队的一篇文章：<https://tech.meituan.com/2017/04/21/mt-leaf.html>。

14、mysql有关权限的表都有哪几个

MySQL服务器通过权限表来控制用户对数据库的访问，权限表存放在mysql数据库里，由mysql_install_db脚本初始化。这些权限表分别user, db, table_priv, columns_priv和host。下面分别介绍一下这些表的结构和内容：

- user权限表：记录允许连接到服务器的用户帐号信息，里面的权限是全局级的。
- db权限表：记录各个帐号在各个数据库上的操作权限。
- table_priv权限表：记录数据表级的操作权限。
- columns_priv权限表：记录数据列级的操作权限。
- host权限表：配合db权限表对给定主机上数据库级操作权限作更细致的控制。这个权限表不受GRANT和REVOKE语句的影响。

15、mysql有哪些数据类型

1、整数类型，包括TINYINT、SMALLINT、MEDIUMINT、INT、BIGINT，分别表示1字节、2字节、3字节、4字节、8字节整数。任何整数类型都可以加上UNSIGNED属性，表示数据是无符号的，即非负整数。

长度：整数类型可以被指定长度，例如：INT(11)表示长度为11的INT类型。长度在大多数场景是没有意义的，它不会限制值的合法范围，只会影响显示字符的个数，而且需要和UNSIGNED ZEROFILL属性配合使用才有意义。

例子，假定类型设定为INT(5)，属性为UNSIGNED ZEROFILL，如果用户插入的数据为12的话，那么数据库实际存储数据为00012。

2、实数类型，包括FLOAT、DOUBLE、DECIMAL。

DECIMAL可以用于存储比BIGINT还大的整型，能存储精确的小数。

而FLOAT和DOUBLE是有取值范围的，并支持使用标准的浮点进行近似计算。

计算时FLOAT和DOUBLE相比DECIMAL效率更高一些，DECIMAL你可以理解成是用字符串进行处理。

3、字符串类型，包括VARCHAR、CHAR、TEXT、BLOB

VARCHAR用于存储可变长字符串，它比定长类型更节省空间。

VARCHAR使用额外1或2个字节存储字符串长度。列长度小于255字节时，使用1字节表示，否则使用2字节表示。

VARCHAR存储的内容超出设置的长度时，内容会被截断。

CHAR是定长的，根据定义的字符串长度分配足够的空间。

CHAR会根据需要使用空格进行填充方便比较。

CHAR适合存储很短的字符串，或者所有值都接近同一个长度。

CHAR存储的内容超出设置的长度时，内容同样会被截断。

使用策略：

对于经常变更的数据来说，CHAR比VARCHAR更好，因为CHAR不容易产生碎片。

对于非常短的列，CHAR比VARCHAR在存储空间上更有效率。

使用时要注意只分配需要的空间，更长的列排序时会消耗更多内存。

尽量避免使用TEXT/BLOB类型，查询时会使用临时表，导致严重的性能开销。

4、枚举类型（ENUM），把不重复的数据存储为一个预定义的集合。

有时可以使用ENUM代替常用的字符串类型。

ENUM存储非常紧凑，会把列表值压缩到一个或两个字节。

ENUM在内部存储时，其实存的是整数。

尽量避免使用数字作为ENUM枚举的常量，因为容易混乱。

排序是按照内部存储的整数

5、日期和时间类型，尽量使用timestamp，空间效率高于datetime，用整数保存时间戳通常不方便处理。
如果需要存储微妙，可以使用bigint存储。
看到这里，这道真题是不是就比较容易回答了。

16、创建索引的三种方式，删除索引

第一种方式：在执行CREATE TABLE时创建索引

```
CREATE TABLE user_index2 (
    id INT auto_increment PRIMARY KEY,
    first_name VARCHAR (16),
    last_name VARCHAR (16),
    id_card VARCHAR (18),
    information text,
    KEY name (first_name, last_name),
    FULLTEXT KEY (information),
    UNIQUE KEY (id_card)
);
```

第二种方式：使用ALTER TABLE命令去增加索引

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

ALTER TABLE用来创建普通索引、UNIQUE索引或PRIMARY KEY索引。

其中table_name是要增加索引的表名，column_list指出对哪些列进行索引，多列时各列之间用逗号分隔。

索引名index_name可自己命名，缺省时，MySQL将根据第一个索引列赋一个名称。另外，ALTER TABLE允许在单个语句中更改多个表，因此可以在同时创建多个索引。

第三种方式：使用CREATE INDEX命令创建

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

CREATE INDEX可对表增加普通索引或UNIQUE索引。（但是，不能创建PRIMARY KEY索引）

删除索引

根据索引名删除普通索引、唯一索引、全文索引：`alter table 表名 drop KEY 索引名`

```
alter table user_index drop KEY name;
alter table user_index drop KEY id_card;
alter table user_index drop KEY information;
```

删除主键索引：`alter table 表名 drop primary key`（因为主键只有一个）。这里值得注意的是，如果主键自增长，那么不能直接执行此操作（自增长依赖于主键索引）：

```
1 alter table user_index drop primary key
```

信息 概况 状态

[SQL]alter table user_index drop primary key

[Err] 1075 - Incorrect table definition; there can be only one auto column and it must be defined as a key

需要取消自增长再行删除：

```
alter table user_index
-- 重新定义字段
MODIFY id int,
drop PRIMARY KEY
```

但通常不会删除主键，因为设计主键一定与业务逻辑无关。

Redis篇

1、Redis持久化机制

Redis是一个支持持久化的内存数据库，通过持久化机制把内存中的数据同步到硬盘文件来保证数据持久化。当Redis重启后通过把硬盘文件重新加载到内存，就能达到恢复数据的目的。

实现：单独创建fork()一个子进程，将当前父进程的数据库数据复制到子进程的内存中，然后由子进程写入到临时文件中，持久化的过程结束了，再用这个临时文件替换上次的快照文件，然后子进程退出，内存释放。

RDB是Redis默认的持久化方式。按照一定的时间周期策略把内存的数据以快照的形式保存到硬盘的二进制文件。即Snapshot快照存储，对应产生的数据文件为dump.rdb，通过配置文件中的save参数来定义快照的周期。（快照可以是其所表示的数据的一个副本，也可以是数据的一个复制品。）AOF：Redis会将每一个收到的写命令都通过Write函数追加到文件最后，类似于MySQL的binlog。当Redis重启是会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。

当两种方式同时开启时，数据恢复Redis会优先选择AOF恢复。

2、缓存雪崩、缓存穿透、缓存预热、缓存更新、缓存降级等问题

一、缓存雪崩

我们可以简单的理解为：由于原有缓存失效，新缓存未到期间

(例如：我们设置缓存时采用了相同的过期时间，在同一时刻出现大面积的缓存过期)，所有原本应该访问缓存的请求都去查询数据库了，而对数据库CPU和内存造成巨大压力，严重的会造成数据库宕机。从而形成一系列连锁反应，造成整个系统崩溃。

解决办法：

大多数系统设计者考虑用加锁（最多的解决方案）或者队列的方式保证来保证不会有大量的线程对数据库一次性进行读写，从而避免失效时大量的并发请求落到底层存储系统上。还有一个简单方案就时讲缓存失效时间分散开。

二、缓存穿透

缓存穿透是指用户查询数据，在数据库没有，自然在缓存中也不会有。这样就导致用户查询的时候，在缓存中找不到，每次都要去数据库再查询一遍，然后返回空（相当于进行了两次无用的查询）。这样请求就绕过缓存直接查数据库，这也是经常提的缓存命中率问题。

解决办法：

最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。

另外也有一个更为简单粗暴的方法，如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。通过这个直接设置的默认值存放到缓存，这样第二次到缓冲中获取就有值了，而不会继续访问数据库，这种办法最简单粗暴。

5TB的硬盘上放满了数据，请写一个算法将这些数据进行排重。如果这些数据是一些32bit大小的数据该如何解决？如果是64bit的呢？

对于空间的利用到达了一种极致，那就是Bitmap和布隆过滤器(Bloom Filter)。

Bitmap：典型的就是哈希表

缺点是，Bitmap对于每个元素只能记录1bit信息，如果还想完成额外的功能，恐怕只能靠牺牲更多的空间、时间来完成了。

布隆过滤器（推荐）

就是引入了 $k(k>1)$ 个相互独立的哈希函数，保证在给定的空间、误判率下，完成元素判断的过程。

它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。

Bloom-Filter算法的核心思想就是利用多个不同的Hash函数来解决“冲突”。

Hash存在一个冲突（碰撞）的问题，用同一个Hash得到的两个URL的值有可能相同。为了减少冲突，我们可以多引入几个Hash，如果通过其中的一个Hash值我们得出某元素不在集合中，那么该元素肯定不在集合中。只有在所有的Hash函数告诉我们该元素在集合中时，才能确定该元素存在于集合中。这便是Bloom-Filter的基本思想。

Bloom-Filter一般用于在大数据量的集合中判定某元素是否存在。

三、缓存预热

缓存预热这个应该是一个比较常见的概念，相信很多小伙伴都应该可以很容易的理解，缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

解决思路：

- 1、直接写个缓存刷新页面，上线时手工操作；
- 2、数据量不大，可以在项目启动的时候自动进行加载；
- 3、定时刷新缓存；

四、缓存更新

除了缓存服务器自带的缓存失效策略之外（Redis默认的有6中策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

- (1) 定时去清理过期的缓存；
- (2) 当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的key是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，大家可以根据自己的应用场景来权衡。

五、缓存降级

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

以参考日志级别设置预案：

- (1) 一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；
- (2) 警告：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警；

(3) 错误：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；

(4) 严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

服务降级的目的，是为了防止Redis服务故障，导致数据库跟着一起发生雪崩问题。因此，对于不重要的缓存数据，可以采取服务降级策略，例如一个比较常见的做法就是，Redis出现问题，不去数据库查询，而是直接返回默认值给用户。

3、热点数据和冷数据是什么

热点数据，缓存才有价值

对于冷数据而言，大部分数据可能还没有再次访问到就已经被挤出内存，不仅占用内存，而且价值不大。频繁修改的数据，看情况考虑使用缓存

对于上面两个例子，寿星列表、导航信息都存在一个特点，就是信息修改频率不高，读取通常非常高的场景。

对于热点数据，比如我们的某IM产品，生日祝福模块，当天的寿星列表，缓存以后可能读取数十万次。再举个例子，某导航产品，我们将导航信息，缓存以后可能读取数百万次。

数据更新前至少读取两次，缓存才有意义。这个是最基本的策略，如果缓存还没有起作用就失效了，那就没有太大价值了。

那存不存在，修改频率很高，但是又不得不考虑缓存的场景呢？有！比如，这个读取接口对数据库的压力很大，但是又是热点数据，这个时候就需要考虑通过缓存手段，减少数据库的压力，比如我们的某助手产品的，点赞数，收藏数，分享数等是非常典型的热点数据，但是又不断变化，此时就需要将数据同步保存到Redis缓存，减少数据库压力。

4、Memcache与Redis的区别都有哪些？

- 1)、存储方式 Memecache把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小。 Redis有部份存在硬盘上，redis可以持久化其数据
- 2)、数据支持类型 memcached所有的值均是简单的字符串，redis作为其替代者，支持更为丰富的数据类型，提供list, set, zset, hash等数据结构的存储
- 3)、使用底层模型不同 它们之间底层实现方式以及与客户端之间通信的应用协议不一样。 Redis直接自己构建了VM机制，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。
- 4). value 值大小不同：Redis 最大可以达到 1gb; memcache 只有 1mb. 5) redis的速度比memcached快很多
- 6) Redis支持数据的备份，即master-slave模式的数据备份。

5、单线程的redis为什么这么快

(一)纯内存操作

(二)单线程操作，避免了频繁的上下文切换

(三)采用了非阻塞I/O多路复用机制

6、redis的数据类型，以及每种数据类型的使用场景

回答：一共五种

(一)String

这个其实没啥好说的，最常规的set/get操作，value可以是String也可以是数字。一般做一些复杂的计数功能的缓存。

(二)hash

这里value存放的是结构化的对象，比较方便的就是操作其中的某个字段。博主在做单点登录的时候，就是用这种数据结构存储用户信息，以cookield作为key，设置30分钟为缓存过期时间，能很好的模拟出类似session的效果。

(三)list

使用List的数据结构，可以做简单的消息队列的功能。另外还有一个就是，可以利用lrange命令，做基于redis的分页功能，性能极佳，用户体验好。本人还用一个场景，很合适—取行情信息。就也是个生产者和消费者的场景。LIST可以很好的完成排队，先进先出的原则。

(四)set

因为set堆放的是一堆不重复值的集合。所以可以做全局去重的功能。为什么不用JVM自带的Set进行去重？因为我们的系统一般都是集群部署，使用JVM自带的Set，比较麻烦，难道为了一个做一个全局去重，再起一个公共服务，太麻烦了。

另外，就是利用交集、并集、差集等操作，可以计算共同喜好，全部的喜好，自己独有的喜好等功能。

(五)sorted set

sorted set多了一个权重参数score，集合中的元素能够按score进行排列。可以做排行榜应用，取TOP N操作。

7、redis的过期策略以及内存淘汰机制

redis采用的是定期删除+惰性删除策略。

为什么不用定时删除策略？

定时删除，用一个定时器来负责监视key，过期则自动删除。虽然内存及时释放，但是十分消耗CPU资源。在大并发请求下，CPU要将时间应用在处理请求，而不是删除key，因此没有采用这一策略。

定期删除+惰性删除是如何工作的呢？

定期删除，redis默认每个100ms检查，是否有过期的key，有过期key则删除。需要说明的是，redis不是每个100ms将所有的key检查一次，而是随机抽取进行检查（如果每隔100ms，全部key进行检查，redis岂不是卡死）。因此，如果只采用定期删除策略，会导致很多key到时间没有删除。

于是，惰性删除派上用场。也就是说在你获取某个key的时候，redis会检查一下，这个key如果设置了过期时间那么是否过期了？如果过期了此时就会删除。

采用定期删除+惰性删除就没其他问题了么？

不是的，如果定期删除没删除key。然后你也没即时去请求key，也就是说惰性删除也没生效。这样，redis的内存会越来越高。那么就应该采用内存淘汰机制。

在redis.conf中有一行配置

```
maxmemory-policy volatile-lru
```

该配置就是配内存淘汰策略的（什么，你没配过？好好反省一下自己）

volatile-lru: 从已设置过期时间的数据集 (server.db[i].expires) 中挑选最近最少使用的数据淘汰

volatile-ttl: 从已设置过期时间的数据集 (server.db[i].expires) 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 (server.db[i].expires) 中任意选择数据淘汰

allkeys-lru: 从数据集 (server.db[i].dict) 中挑选最近最少使用的数据淘汰

allkeys-random: 从数据集 (server.db[i].dict) 中任意选择数据淘汰

no-eviction (驱逐) : 禁止驱逐数据，新写入操作会报错

ps: 如果没有设置 expire 的key, 不满足先决条件(prerequisites); 那么 volatile-lru, volatile-random 和 volatile-ttl 策略的行为, 和 noeviction(不删除) 基本上一致。

8、Redis 为什么是单线程的

官方FAQ表示，因为Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了（毕竟采用多线程会有很多麻烦！）Redis利用队列技术将并发访问变为串行访问

1) 绝大部分请求是纯粹的内存操作（非常快速）2) 采用单线程，避免了不必要的上下文切换和竞争条

件

3) 非阻塞IO优点：

- 速度快，因为数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)
- 支持丰富数据类型，支持string, list, set, sorted set, hash
- 支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行
- 丰富的特性：可用于缓存，消息，按key设置过期时间，过期后将会自动删除如何解决redis的并发竞争key问题

同时有多个子系统去set一个key。这个时候要注意什么呢？ 不推荐使用redis的事务机制。因为我们的生产环境，基本都是redis集群环境，做了数据分片操作。你一个事务中有涉及到多个key操作的时候， 这多个key不一定都存储在同一个redis-server上。因此，redis的事务机制，十分鸡肋。

- (1)如果对这个key操作，不要求顺序： 准备一个分布式锁，大家去抢锁，抢到锁就做set操作即可
(2)如果对这个key操作，要求顺序： 分布式锁+时间戳。 假设这会系统B先抢到锁，将key1设置为{valueB 3:05}。接下来系统A抢到锁，发现自己的valueA的时间戳早于缓存中的时间戳，那就不做set操作了。以此类推。
(3)利用队列，将set方法变成串行访问也可以redis遇到高并发，如果保证读写key的一致性 对redis的操作都是具有原子性的，是线程安全的操作，你不用考虑并发问题，redis内部已经帮你处理好并发的问题了。

9、Redis 常见性能问题和解决方案？

- (1) Master 最好不要做任何持久化工作，如 RDB 内存快照和 AOF 日志文件
- (2) 如果数据比较重要，某个 Slave 开启 AOF 备份数据，策略设置为每秒同步一次
- (3) 为了主从复制的速度和连接的稳定性，Master 和 Slave 最好在同一个局域网内
- (4) 尽量避免在压力很大的主库上增加从库
- (5) 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3...

10、为什么Redis的操作是原子性的，怎么保证原子性的？

对于Redis而言，命令的原子性指的是：一个操作的不可以再分，操作要么执行，要么不执行。

Redis的操作之所以是原子性的，是因为Redis是单线程的。

Redis本身提供的所有API都是原子操作，Redis中的事务其实是要保证批量操作的原子性。

多个命令在并发中也是原子性的吗？

不一定，将get和set改成单命令操作，incr。使用Redis的事务，或者使用Redis+Lua==的方式实现。

11、Redis事务

Redis事务功能是通过MULTI、EXEC、DISCARD和WATCH 四个原语实现的

Redis会将一个事务中的所有命令序列化，然后按顺序执行。

1.redis 不支持回滚 “Redis 在事务失败时不进行回滚，而是继续执行余下的命令”， 所以 Redis 的内部可以保持简单且快速。

2.如果在一个事务中的命令出现错误，那么所有的命令都不会执行；

3.如果在一个事务中出现运行错误，那么正确的命令会被执行。

1) MULTI命令用于开启一个事务，它总是返回OK。 MULTI执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个队列中，当EXEC命令被调用时，所有队列中的命令才会被执行。

2) EXEC: 执行所有事务块内的命令。返回事务块内所有命令的返回值，按命令执行的先后顺序排列。

当操作被打断时，返回空值

nil 。3) 通过调用DISCARD，客户端可以清空事务队列，并放弃执行事务，并且客户端会从事务状态中退出。

4) WATCH 命令可以为 Redis 事务提供 check-and-set (CAS) 行为。可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行，监控一直持续到EXEC命令。

SpringCloud篇

1、什么是SpringCloud

Spring cloud 流应用程序启动器是基于 Spring Boot 的 Spring 集成应用程序，提供与外部系统的集成。Spring cloud Task，一个生命周期短暂的微服务框架，用于快速构建执行有限数据处理的应用程序。

2、什么是微服务

微服务架构是一种架构模式或者说是一种架构风格，它提倡将单一应用程序划分为一组小的服务，每个服务运行在其独立的自己的进程中，服务之间相互协调、互相配合，为用户提供最终价值。服务之间采用轻量级的通信机制互相沟通（通常是基于HTTP的RESTful API），每个服务都围绕着具体的业务进行构建，并且能够被独立的构建在生产环境、类生产环境等。另外，应避免统一的、集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言、工具对其进行构建，可以有一个非常轻量级的集中式管理来协调这些服务，可以使用不同的语言来编写服务，也可以使用不同的数据存储。

3、SpringCloud有什么优势

使用 Spring Boot 开发分布式微服务时，我们面临以下问题

- (1) 与分布式系统相关的复杂性-这种开销包括网络问题，延迟开销，带宽问题，安全问题。
- (2) 服务发现-服务发现工具管理群集中的流程和服务如何查找和互相交谈。它涉及一个服务目录，在该目录中注册服务，然后能够查找并连接到该目录中的服务。
- (3) 冗余-分布式系统中的冗余问题。
- (4) 负载平衡 --负载平衡改善跨多个计算资源的工作负荷，诸如计算机，计算机集群，网络链路，中央处理单元，或磁盘驱动器的分布。
- (5) 性能-问题 由于各种运营开销导致的性能问题。
- (6) 部署复杂性-Devops 技能的要求。

4、什么是服务熔断？什么是服务降级？

熔断机制是应对雪崩效应的一种微服务链路保护机制。当某个微服务不可用或者响应时间太长时，会进行服务降级，进而熔断该节点微服务的调用，快速返回“错误”的响应信息。当检测到该节点微服务调用响应正常后恢复调用链路。在SpringCloud框架里熔断机制通过Hystrix实现，Hystrix会监控微服务间调用的状况，当失败的调用到一定阈值，缺省是5秒内调用20次，如果失败，就会启动熔断机制。

服务降级，一般是从整体负荷考虑。就是当某个服务熔断之后，服务器将不再被调用，此时客户端可以自己准备一个本地的fallback回调，返回一个缺省值。这样做，虽然水平下降，但好歹可用，比直接挂掉强。

Hystrix相关注解

@EnableHystrix: 开启熔断
@HystrixCommand(fallbackMethod="XXX"): 声明一个失败回滚处理函数XXX, 当被注解的方法执行超时(默认是1000毫秒), 就会执行fallback函数, 返回错误提示。

微服务条目	落地的技术
服务开发	SpringBoot、Spring、SpringMVC
服务配置管理	Netflix公司的Archaius、阿里的Diamond等
服务注册与发现	Eureka、Consul、Zookeeper
服务调用	RPC、Rest、gRPC
服务熔断器	Hystrix、Envoy等
负载均衡	Nginx、Ribbon
服务接口调用(客户端调用服务的简化工具)	Feign
消息队列	Kafka、RabbitMQ、ActiveMQ等
服务配置中心配置管理	SpringCloudConfig、Chef等
服务路由(API网关)	Zuul
服务监控	Zabbix、Nagios、Metrics、Spectator等
全链路追踪	Zipkin、Brave、Dapper等
服务部署	Docker、OpenStack、Kubernetes等
数据流操作开发包	SpringCloud Stream
事件消息总线	Spring Cloud Bus

5、Eureka和zookeeper都可以提供服务注册与发现的功能，请说说两个的区别？

Zookeeper保证了CP(C:一致性, P:分区容错性), Eureka保证了AP(A:高可用)

1.当向注册中心查询服务列表时, 我们可以容忍注册中心返回的是几分钟以前的信息, 但不能容忍直接down掉不可用。也就是说, 服务注册功能对高可用性要求比较高, 但zk会出现这样一种情况, 当master节点因为网络故障与其他节点失去联系时, 剩余节点会重新选leader。问题在于, 选取leader时间过长, 30 ~ 120s, 且选取期间zk集群都不可用, 这样就会导致选取期间注册服务瘫痪。在云部署的环境下, 因网络问题使得zk集群失去master节点是较大概率会发生的事, 虽然服务能够恢复, 但是漫长的选取时间导致的注册长期不可用是不能容忍的。

2.Eureka保证了可用性, Eureka各个节点是平等的, 几个节点挂掉不会影响正常节点的工作, 剩余的节点仍然可以提供注册和查询服务。而Eureka的客户端向某个Eureka注册或发现时发生连接失败, 则会自动切换到其他节点, 只要有一台Eureka还在, 就能保证注册服务可用, 只是查到的信息可能不是最新的。除此之外, Eureka还有自我保护机制, 如果在15分钟内超过85%的节点没有正常的心跳, 那么Eureka就认为客户端与注册中心发生了网络故障, 此时会出现以下几种情况:

- ①、Eureka不在从注册列表中移除因为长时间没有收到心跳而应该过期的服务。
- ②、Eureka仍然能够接受新服务的注册和查询请求, 但是不会被同步到其他节点上(即保证当前节点仍然可用)
- ③、当网络稳定时, 当前实例新的注册信息会被同步到其他节点。

因此, Eureka可以很好的应对因网络故障导致部分节点失去联系的情况, 而不会像Zookeeper那样使整个微服务瘫痪

6、SpringBoot和SpringCloud的区别？

SpringBoot专注于快速方便的开发单个个体微服务。

SpringCloud是关注全局的微服务协调整理治理框架，它将SpringBoot开发的一个个单体微服务整合并管理起来，

为各个微服务之间提供，配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等集成服务

SpringBoot可以离开SpringCloud独立使用开发项目，但是SpringCloud离不开SpringBoot，属于依赖的关系。

SpringBoot专注于快速、方便的开发单个微服务个体，SpringCloud关注全局的服务治理框架。

7、负载平衡的意义什么？

在计算中，负载平衡可以改善跨计算机，计算机集群，网络链接，中央处理单元或磁盘驱动器等多种计算资源的工作负载分布。负载平衡旨在优化资源使用，最大化吞吐量，最小化响应时间并避免任何单一资源

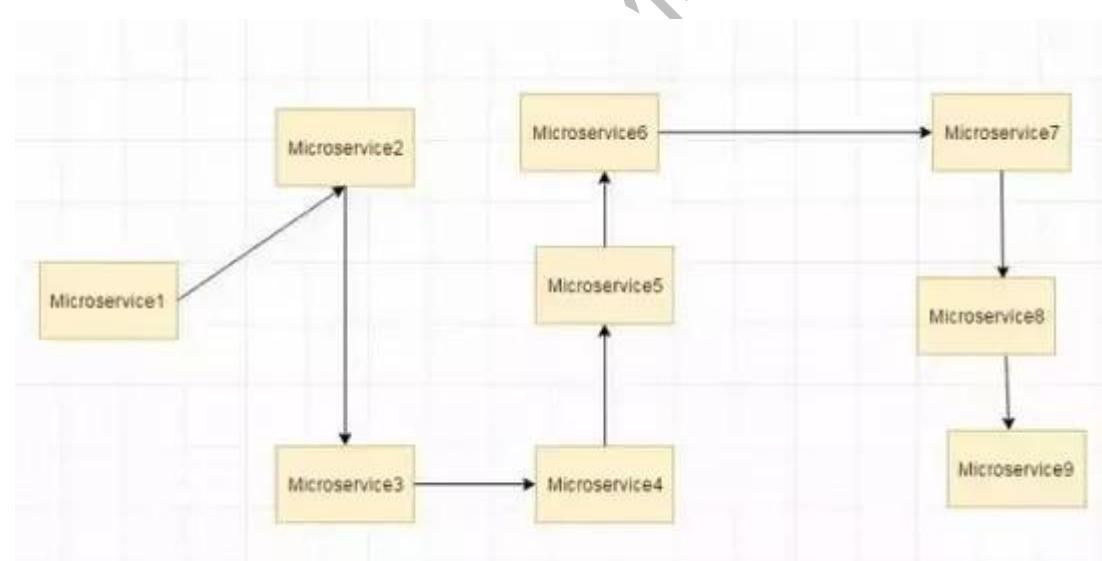
的过载。使用多个组件进行负载平衡而不是单个组件可能会通过冗余来提高可靠性和可用性。负载平衡通常涉及专用软件或硬件，例如多层交换机或域名系统服务器进程。

8、什么是Hystrix？它如何实现容错？

Hystrix是一个延迟和容错库，旨在隔离远程系统，服务和第三方库的访问点，当出现故障是不可避免的故障时，停止级联故障并在复杂的分布式系统中实现弹性。

通常对于使用微服务架构开发的系统，涉及到许多微服务。这些微服务彼此协作。

思考以下微服务



假设如果上图中的微服务9失败了，那么使用传统方法我们将传播一个异常。但这仍然会导致整个系统崩溃。

随着微服务数量的增加，这个问题变得更加复杂。微服务的数量可以高达1000.这是hystrix出现的地方我们将使用Hystrix在这种情况下的Fallback方法功能。我们有两个服务employee-consumer使用由employee-consumer公开的服务。

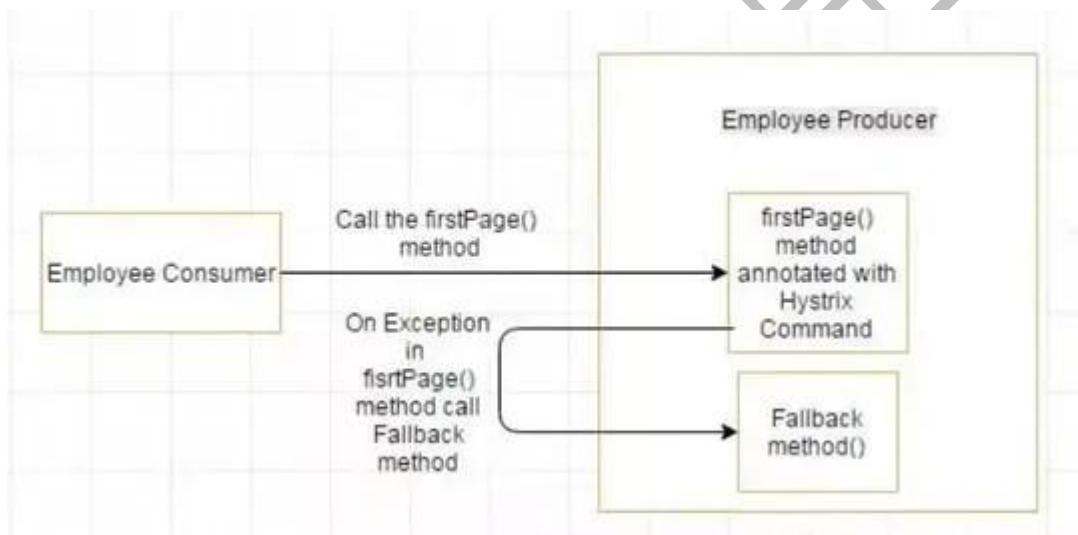
简化图如下所示



现在假设由于某种原因，employee-producer公开的服务会抛出异常。我们在这种情况下使用Hystrix定义了一个回退方法。这种后备方法应该具有与公开服务相同的返回类型。如果暴露服务中出现异常，则回退方法将返回一些值。

9、什么是Hystrix断路器？我们需要它吗？

由于某些原因，employee-consumer公开服务会引发异常。在这种情况下使用Hystrix我们定义了一个回退方法。如果在公开服务中发生异常，则回退方法返回一些默认值。



如果firstPage method() 中的异常继续发生，则Hystrix电路将中断，并且员工使用者将一起跳过firtsPage方法，并直接调用回退方法。断路器的目的是给第一页方法或第一页方法可能调用的其他方法留出时间，并导致异常恢复。可能发生的情况是，在负载较小的情况下，导致异常的问题有更好的恢复机会。



10、说说 RPC 的实现原理

首先需要有处理网络连接通讯的模块，负责连接建立、管理和消息的传输。其次需要有编解码的模块，因为网络通讯都是传输的字节码，需要将我们使用的对象序列化和反序列化。剩下的就是客户端和服务器端的部分，服务器端暴露要开放的服务接口，客户调用服务接口的一个代理实现，这个代理实现负责收集数据、编码并传输给服务器然后等待结果返回。

Nginx篇

1、简述一下什么是Nginx，它有什么优势和功能？

Nginx是一个web服务器和方向代理服务器，用于HTTP、HTTPS、SMTP、POP3和IMAP协议。因它的稳定性、丰富的功能集、示例配置文件和低系统资源的消耗而闻名。

Nginx---Ngine X，是一款免费的、自由的、开源的、高性能HTTP服务器和反向代理服务器，也是一个IMAP、POP3、SMTP代理服务器；Nginx以其高性能、稳定性、丰富的功能、简单的配置和低资源消耗而闻名。

也就是说Nginx本身就可以托管网站（类似于Tomcat一样），进行Http服务处理，也可以作为反向代理服务器、负载均衡器和HTTP缓存。

Nginx 解决了服务器的C10K（就是在一秒之内连接客户端的数目为10k即1万）问题。它的设计不像传统的服务器那样使用线程处理请求，而是一个更加高级的机制—事件驱动机制，是一种异步事件驱动结构。

优点：

（1）更快

这表现在两个方面：一方面，在正常情况下，单次请求会得到更快的响应；另一方面，在高峰期（如有数以万计的并发请求），Nginx可以比其他Web服务器更快地响应请求。

（2）高扩展性，跨平台

Nginx的设计极具扩展性，它完全是由多个不同功能、不同层次、不同类型且耦合度极低的模块组成。因此，当对某一个模块修复Bug或进行升级时，可以专注于模块自身，无须在意其他。而且在HTTP模块中，还设计了HTTP过滤器模块：一个正常的HTTP模块在处理完请求后，会有一串HTTP过滤器模块对请求的结果进行再处理。这样，当我们开发一个新的HTTP模块时，不但可以使用诸如HTTP核心模块、events模块、log模块等不同层次或者不同类型的模块，还可以原封不动地复用大量已有的HTTP过滤器模块。这种低耦合度的优秀设计，造就了Nginx庞大的第三方模块，当然，公开的第三方模块也如官方发布的模块一样容易使用。

Nginx的模块都是嵌入到二进制文件中执行的，无论官方发布的模块还是第三方模块都是如此。这使得第三方模块一样具备极其优秀的性能，充分利用Nginx的高并发特性，因此，许多高流量的网站都倾向于开发符合自己业务特性的定制模块。

（3）高可靠性：用于反向代理，宕机的概率微乎其微

高可靠性是我们选择Nginx的最基本条件，因为Nginx的可靠性是大家有目共睹的，很多家高流量网站都在核心服务器上大规模使用Nginx。Nginx的高可靠性来自于其核心框架代码的优秀设计、模块设计的简单性；另外，官方提供的常用模块都非常稳定，每个worker进程相对独立，master进程在1个worker进程出错时可以快速“拉起”新的worker子进程提供服务。

（4）低内存消耗

一般情况下，10 000个非活跃的HTTP Keep-Alive连接在Nginx中仅消耗2.5MB的内存，这是Nginx支持高并发连接的基础。

（5）单机支持10万以上的并发连接

这是一个非常重要的特性！随着互联网的迅猛发展和互联网用户数量的成倍增长，各大公司、网站都需要应付海量并发请求，一个能够在峰值期顶住10万以上并发请求的Server，无疑会得到大家的青睐。理

论上，Nginx支持的并发连接上限取决于内存，10万远未封顶。当然，能够及时地处理更多的并发请求，是与业务特点紧密相关的。

(6) 热部署

master管理进程与worker工作进程的分离设计，使得Nginx能够提供热部署功能，即可以在 7×24 小时不间断服务的前提下，升级Nginx的可执行文件。当然，它也支持不停止服务就更新配置项、更换日志文件等功能。

(7) 最自由的BSD许可协议

这是Nginx可以快速发展的强大动力。BSD许可协议不只是允许用户免费使用Nginx，它还允许用户在自己的项目中直接使用或修改Nginx源码，然后发布。这吸引了无数开发者继续为Nginx贡献自己的智慧。

以上7个特点当然不是Nginx的全部，拥有无数个官方功能模块、第三方功能模块使得Nginx能够满足绝大部分应用场景，这些功能模块间可以叠加以实现更加强大、复杂的功能，有些模块还支持Nginx与Perl、Lua等脚本语言集成工作，大大提高了开发效率。这些特点促使用户在寻找一个Web服务器时更多考虑Nginx。

选择Nginx的核心理由还是它能在支持高并发请求的同时保持高效的服务

2、Nginx是如何处理一个HTTP请求的呢？

Nginx 是一个高性能的 Web 服务器，能够同时处理大量的并发请求。它结合多进程机制和异步机制，异步机制使用的是异步非阻塞方式，接下来就给大家介绍一下 Nginx 的多线程机制和异步非阻塞机制。

1、多进程机制

服务器每当收到一个客户端时，就有服务器主进程（master process）生成一个子进程（worker process）出来和客户端建立连接进行交互，直到连接断开，该子进程就结束了。

使用进程的好处是各个进程之间相互独立，不需要加锁，减少了使用锁对性能造成影响，同时降低编程的复杂度，降低开发成本。其次，采用独立的进程，可以让进程互相之间不会影响，如果一个进程发生异常退出时，其它进程正常工作，master进程则很快启动新的worker进程，确保服务不会中断，从而将风险降到最低。

缺点是操作系统生成一个子进程需要进行内存复制等操作，在资源和时间上会产生一定的开销。当有大量请求时，会导致系统性能下降。

2、异步非阻塞机制

每个工作进程 使用 异步非阻塞方式，可以处理 多个客户端请求。

当某个工作进程 接收到客户端的请求以后，调用 IO 进行处理，如果不能立即得到结果，就去 处理其他请求（即为 非阻塞）；而 客户端 在此期间也 无需等待响应，可以去处理其他事情（即为 异步）。

当 IO 返回时，就会通知此 工作进程；该进程得到通知，暂时 挂起 当前处理的事务去 响应客户端请求。

3、列举一些Nginx的特性

1. Nginx服务器的特性包括：
2. 反向代理/L7负载均衡器
3. 嵌入式Perl解释器
4. 动态二进制升级
5. 可用于重新编写URL，具有非常好的PCRE支持



关注【程序员生活志】
免费获取更多海量资源

4、请列举Nginx和Apache之间的不同点

nginx	apache
1.nginx 是一个基于web服务器	1.Apache 是一个基于流程的服务器
2.所有请求都由一个线程来处理	2.单线程处理单个请求
3.nginx避免子进程的概念	3.apache是基于子进程的
4.nginx类似于速度	4.apache类似于功率
5.nginx在内存消耗和连接方面比较好	5.apache在内存消耗和连接方面并没有提高
6.nginx在负载均衡方面表现较好	6.apache当流量达到进程的极限时，apache将拒绝新的连接
7.对于PHP来说，nginx更可取，因为他支持 PHP	7.apache支持的php python Perl和其他语言，使用插件，当应用程序基于python和ruby时，它非常有用
8.nginx 不支持像ibmi 和 openvms 一样的os	8.apache支持更多的os
9.nginx 只具有核心功能	9.apache 提供了比Nginx更多的功能
10.nginx 性能和可伸缩性不依赖于硬件	10.apache 依赖于CPU和内存等硬件组件

5、在Nginx中，如何使用未定义的服务器名称来阻止处理请求？

只需将请求删除的服务器就可以定义为：

```
Server{
    listen 80;
    server_name "";
    return 444;
}
```

这里，服务器名被保留为一个空字符串，它将在没有“主机”头字段的情况下匹配请求，而一个特殊的 Nginx 的非标准代码 444 被返回，从而终止连接。

一般推荐 worker 进程数与 CPU 内核数一致，这样一来不存在大量的子进程生成和管理任务，避免了进程之间竞争 CPU 资源和进程切换的开销。而且 Nginx 为了更好的利用多核特性，提供了 CPU 亲缘性的绑定选项，我们可以将某一个进程绑定在某一个核上，这样就不会因为进程的切换带来 Cache 的失效。

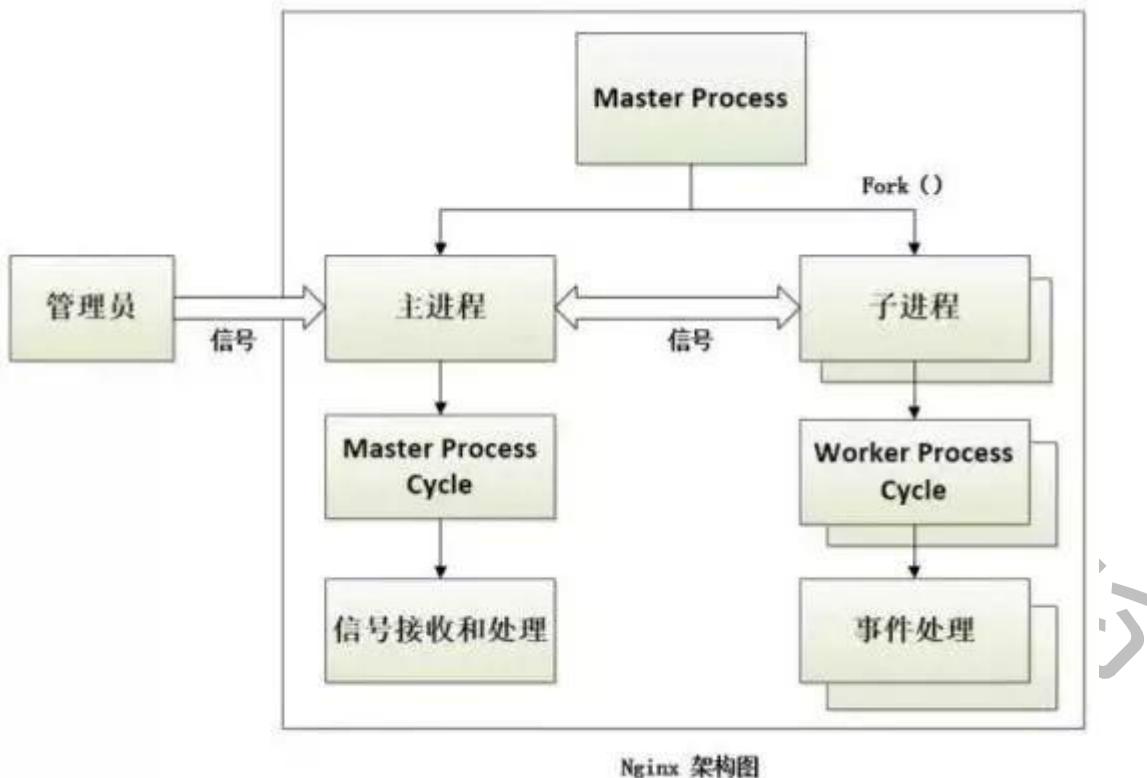
对于每个请求，有且只有一个工作进程对其处理。首先，每个 worker 进程都是从 master 进程 fork 过来。在 master 进程里面，先建立好需要 listen 的 socket (listenfd) 之后，然后再 fork 出多个 worker 进程。

所有 worker 进程的 listenfd 会在新连接到来时变得可读，为保证只有一个进程处理该连接，所有 worker 进程在注册 listenfd 读事件前抢占 accept_mutex，抢到互斥锁的那个进程注册 listenfd 读事件，在读事件里调用 accept 接受该连接。

当一个 worker 进程在 accept 这个连接之后，就开始读取请求、解析请求、处理请求，产生数据后，再返回给客户端，最后才断开连接。这样一个完整的请求就是这样的了。我们可以看到，一个请求，完全由 worker 进程来处理，而且只在一个 worker 进程中处理。



关注【程序员生活志】
免费获取更多海量资源

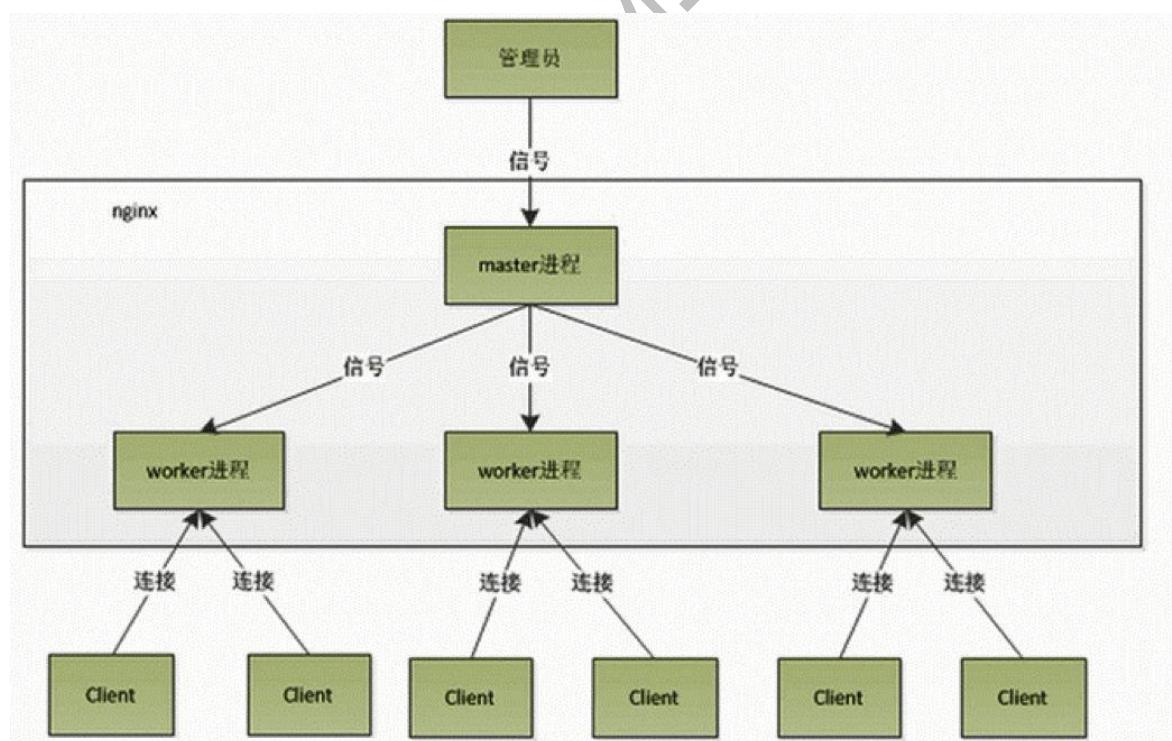


Nginx 架构图

在 Nginx 服务器的运行过程中，主进程和工作进程需要进程交互。交互依赖于 Socket 实现的管道来实现。

6、请解释Nginx服务器上的Master和Worker进程分别是什么？

- 主程序 Master process 启动后，通过一个 for 循环来接收 和 处理外部信号；
- 主进程通过 fork() 函数产生 worker 子进程，每个子进程执行一个 for 循环来实现Nginx服务器对事件的接收和处理。

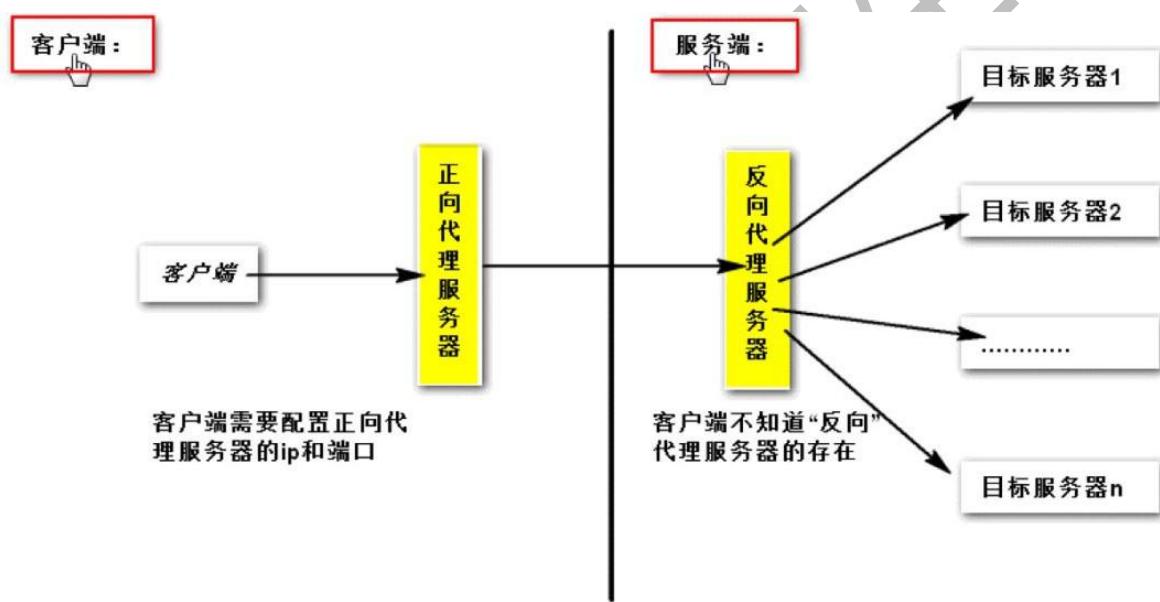


7、请解释代理中的正向代理和反向代理

首先，代理服务器一般指局域网内部的机器通过代理服务器发送请求到互联网上的服务器，代理服务器一般作用在客户端。例如：GoAgent翻墙软件。我们的客户端在进行翻墙操作的时候，我们使用的正是正向代理，通过正向代理的方式，在我们的客户端运行一个软件，将我们的HTTP请求转发到其他不同的服务器端，实现请求的分发。

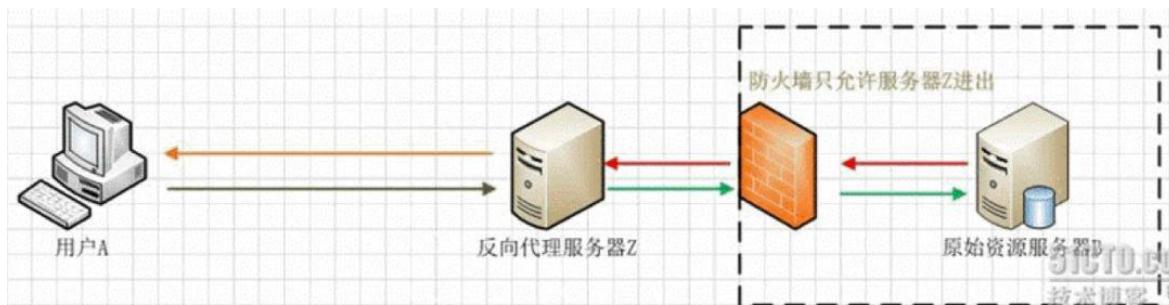


反向代理服务器作用在服务器端，它在服务器端接收客户端的请求，然后将请求分发给具体的服务器进行处理，然后再将服务器的相应结果反馈给客户端。Nginx就是一个反向代理服务器软件。



从上图可以看出：客户端必须设置正向代理服务器，当然前提是要知道正向代理服务器的IP地址，还有代理程序的端口。

反向代理正好与正向代理相反，对于客户端而言代理服务器就像是原始服务器，并且客户端不需要进行任何特别的设置。客户端向反向代理的命名空间（name-space）中的内容发送普通请求，接着反向代理将判断向何处（原始服务器）转交请求，并将获得的内容返回给客户端。



8、解释Nginx用途

Nginx服务器的最佳用法是在网络上部署动态HTTP内容，使用SCGI、WSGI应用程序服务器、用于脚本的FastCGI处理程序。它还可以作为负载均衡器。

MQ篇

1、为什么使用MQ

核心：解耦,异步,削峰

(1) 解耦：A 系统发送数据到 BCD 三个系统，通过接口调用发送。如果 E 系统也要这个数据呢？那如果 C 系统现在不需要了呢？A 系统负责人几乎崩溃……A 系统跟其它各种乱七八糟的系统严重耦合，A 系统产生一条比较关键的数据，很多系统都需要 A 系统将这个数据发送过来。如果使用 MQ，A 系统产生一条数据，发送到 MQ 里面去，哪个系统需要数据自己去 MQ 里面消费。如果新系统需要数据，直接从 MQ 里消费即可；如果某个系统不需要这条数据了，就取消对 MQ 消息的消费即可。这样下来，A 系统压根儿不需要去考虑要给谁发送数据，不需要维护这个代码，也不需要考虑人家是否调用成功、失败超时等情况。

就是一个系统或者一个模块，调用了多个系统或者模块，互相之间的调用很复杂，维护起来很麻烦。但是其实这个调用是不需要直接同步调用接口的，如果用 MQ 给它异步化解耦。

(2) 异步：A 系统接收一个请求，需要在自己本地写库，还需要在 BCD 三个系统写库，自己本地写库要 3ms，BCD 三个系统分别写库要 300ms、450ms、200ms。最终请求总延时是 $3 + 300 + 450 + 200 = 953\text{ms}$ ，接近 1s，用户感觉搞个什么东西，慢死了慢死了。用户通过浏览器发起请求。如果使用 MQ，那么 A 系统连续发送 3 条消息到 MQ 队列中，假如耗时 5ms，A 系统从接受一个请求到返回响应给用户，总时长是 $3 + 5 = 8\text{ms}$ 。

(3) 削峰：减少高峰期对服务器压力。

2、MQ优缺点

优点上面已经说了，就是在特殊场景下有其对应的好处，解耦、异步、削峰。

缺点有以下几个：

系统可用性降低

系统引入的外部依赖越多，越容易挂掉。万一 MQ 挂了，MQ 一挂，整套系统崩溃，你不就完了？

系统复杂度提高

硬生生加个 MQ 进来，你怎么保证消息没有重复消费？怎么处理消息丢失的情况？怎么保证消息传递的顺序性？问题一大堆。

一致性问题

A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，咋整？你这数据就不一致了。

3、Kafka、ActiveMQ、RabbitMQ、RocketMQ 都有什么区别？

对于吞吐量来说kafka和RocketMQ支撑高吞吐，ActiveMQ和RabbitMQ比他们低一个数量级。对于延迟量来说RabbitMQ是最低的。

1. 从社区活跃度

按照目前网络上的资料，RabbitMQ、activeM、ZeroMQ 三者中，综合来看，RabbitMQ 是首选。

2. 持久化消息比较

ActiveMq 和 RabbitMq 都支持。持久化消息主要是指我们机器在不可抗力因素等情况下挂掉了，消息不会丢失的机制。

3. 综合技术实现

可靠性、灵活的路由、集群、事务、高可用的队列、消息排序、问题追踪、可视化管理工具、插件系统等等。

RabbitMq / Kafka 最好，ActiveMq 次之，ZeroMq 最差。当然 ZeroMq 也可以做到，不过自己必须手动写代码实现，代码量不小。尤其是可靠性中的：持久性、投递确认、发布者证实和高可用性。

4. 高并发

毋庸置疑，RabbitMQ 最高，原因是它的实现语言是天生具备高并发高可用的erlang 语言。

5. 比较关注的比较， RabbitMQ 和 Kafka

RabbitMq 比 Kafka 成熟，在可用性上，稳定性上，可靠性上， RabbitMq 胜于 Kafka （理论上）。

另外，Kafka 的定位主要在日志等方面，因为 Kafka 设计的初衷就是处理日志的，可以看做是一个日志（消息）系统一个重要组件，针对性很强，所以如果业务方面还是建议选择 RabbitMq。

还有就是，Kafka 的性能（吞吐量、TPS）比 RabbitMq 要高出来很多。

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
单机吞吐量	万级，比 RocketMQ、Kafka 低一个数量级	同 ActiveMQ	10 万级，支撑高吞吐	10 万级，高吞吐，一般配合大数据类的系统来进行实时数据计算、日志采集等场景
topic 数量对吞吐量的影响			topic 可以达到几百/几千的级别，吞吐量会有较小幅度的下降，这是 RocketMQ 的一大优势，在同等机器下，可以支撑大量的 topic	topic 从几十到几百个时候，吞吐量会大幅度下降，在同等机器下，Kafka 尽量保证 topic 数量不要过多，如果要支撑大规模的 topic，需要增加更多的机器资源
时效性	ms 级	微秒级，这是 RabbitMQ 的一大特点，延迟最低	ms 级	延迟在 ms 级以内
可用性	高，基于主从架构实现高可用	同 ActiveMQ	非常高，分布式架构	非常高，分布式，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用
消息可靠性	有较低的概率丢失数据		经过参数优化配置，可以做到 0 丢失	同 RocketMQ
功能支持	MQ 领域的功能极其完备	基于 erlang 开发，并发能力很强，性能极好，延时很低	MQ 功能较为完善，还是分布式的，扩展性好	功能较为简单，主要支持简单的 MQ 功能，在大数据领域的实时计算以及日志采集被大规模使用

4、如何保证高可用的？

RabbitMQ 是比较有代表性的，因为是基于主从（非分布式）做高可用性的，我们就以 RabbitMQ 为例子讲解第一种 MQ 的高可用性怎么实现。RabbitMQ 有三种模式：单机模式、普通集群模式、镜像集群模式。

单机模式，就是 Demo 级别的，一般就是你本地启动了玩玩儿的？，没人生产用单机模式

普通集群模式，意思就是在多台机器上启动多个 RabbitMQ 实例，每个机器启动一个。你创建的 queue，只会放在一个 RabbitMQ 实例上，但是每个实例都同步 queue 的元数据（元数据可以认为是queue的一些配置信息，通过元数据，可以找到 queue 所在实例）。你消费的时候，实际上如果连接到了另外一个实例，那么那个实例会从 queue 所在实例上拉取数据过来。这方案主要是提高吞吐量的，就是说让集群中多个节点来服务某个 queue 的读写操作。

镜像集群模式：这种模式，才是所谓的 RabbitMQ 的高可用模式。跟普通集群模式不一样的是，在镜像集群模式下，你创建的 queue，无论元数据还是 queue 里的消息都会存在于多个实例上，就是说，每个 RabbitMQ 节点都有这个 queue 的一个完整镜像，包含 queue 的全部数据的意思。然后每次你写消息到 queue 的时候，都会自动把消息同步到多个实例的 queue 上。RabbitMQ 有很好的管理控制台，就是在后台新增一个策略，这个策略是镜像集群模式的策略，指定的时候是可以要求数据同步到所有节点的，也可以要求同步到指定数量的节点，再次创建 queue 的时候，应用这个策略，就会自动将数据同步到其他的节点上去了。这样的话，好处在于，你任何一个机器宕机了，没事儿，其它机器（节点）还包含了这个 queue 的完整数据，别的 consumer 都可以到其它节点上去消费数据。坏处在于，第一，这个性能开销也太大了吧，消息需要同步到所有机器上，导致网络带宽压力和消耗很重！

RabbitMQ 一个 queue 的数据都是放在一个节点里的，镜像集群下，也是每个节点都放这个 queue 的完整数据。

Kafka 一个最基本的架构认识：由多个 broker 组成，每个 broker 是一个节点；你创建一个 topic，这个 topic 可以划分为多个 partition，每个 partition 可以存在于不同的 broker 上，每个 partition 就放一部分数据。这就是天然的分布式消息队列，就是说一个 topic 的数据，是分散放在多个机器上的，每个机器就放一部分数据。Kafka 0.8 以后，提供了 HA 机制，就是 replica（复制品）副本机制。每个 partition 的数据都会同步到其它机器上，形成自己的多个 replica 副本。所有 replica 会选举一个 leader 出来，那么生产和消费都跟这个 leader 打交道，然后其他 replica 就是 follower。写的时候，leader 会负责把数据同步到所有 follower 上去，读的时候就直接读 leader 上的数据即可。只能读写 leader？很简单，要是你可以随意读写每个 follower，那么就要 care 数据一致性的问题，系统复杂度太高，很容易出问题。

Kafka 会均匀地将一个 partition 的所有 replica 分布在不同的机器上，这样才可以提高容错性。因为如果某个 broker 宕机了，没事儿，那个 broker 上面的 partition 在其他机器上都有副本的，如果这上面有某个 partition 的 leader，那么此时会从 follower 中重新选举一个新的 leader 出来，大家继续读写那个新的 leader 即可。这就有所谓的高可用性了。写数据的时候，生产者就写 leader，然后 leader 将数据落地写本地磁盘，接着其他 follower 自己主动从 leader 来 pull 数据。一

旦所有 follower 同步好数据了，就会发送 ack 给 leader，leader 收到所有 follower 的 ack 之后，就会返回写成功的消息给生产者。（当然，这只是其中一种模式，还可以适当调整这个行为）消费的时候，只会从 leader 去读，但是只有当一个消息已经被所有 follower 都同步成功返回 ack 的时候，这个消息才会被消费者读到。

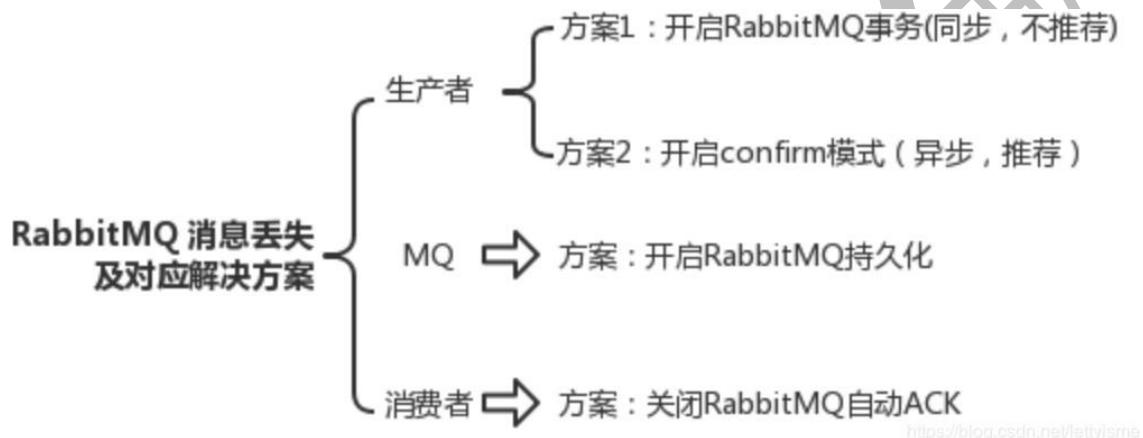
5、如何保证消息的可靠传输？如果消息丢了怎么办

数据的丢失问题，可能出现在生产者、MQ、消费者中

生产者丢失：生产者将数据发送到 RabbitMQ 的时候，可能数据就在半路给搞丢了，因为网络问题啥的，都有可能。此时可以选择用 RabbitMQ 提供的事务功能，就是生产者发送数据之前开启 RabbitMQ 事务 channel.txSelect，然后发送消息，如果消息没有成功被 RabbitMQ 接收到，那么生产者会收到异常报错，此时就可以回滚事务 channel.txRollback，然后重试发送消息；如果收到了消息，那么可以提交事务 channel.txCommit。吞吐量会下来，因为太耗性能。所以一般来说，如果你要确保说写 RabbitMQ 的消息别丢，可以开启 confirm 模式，在生产者那里设置开启 confirm 模式之后，你每次写的消息都会分配一个唯一的 id，然后如果写入了 RabbitMQ 中，RabbitMQ 会给你回传一个 ack 消息，告诉你说这个消息 ok 了。如果 RabbitMQ 没能处理这个消息，会回调你一个 nack 接口，告诉你这个消息接收失败，你可以重试。而且你可以结合这个机制自己在内存里维护每个消息 id 的状态，如果超过一定时间还没接收到这个消息的回调，那么你可以重发。事务机制和 confirm 机制最大的不同在于，事务机制是同步的，你提交一个事务之后会阻塞在那儿，但是 confirm 机制是异步的，你发送个消息之后就可以发送下一个消息，然后那个消息 RabbitMQ 接收了之后会异步回调你一个接口通知你这个消息接收到了。所以一般在生产者这块避免数据丢失，都是用 confirm 机制的。

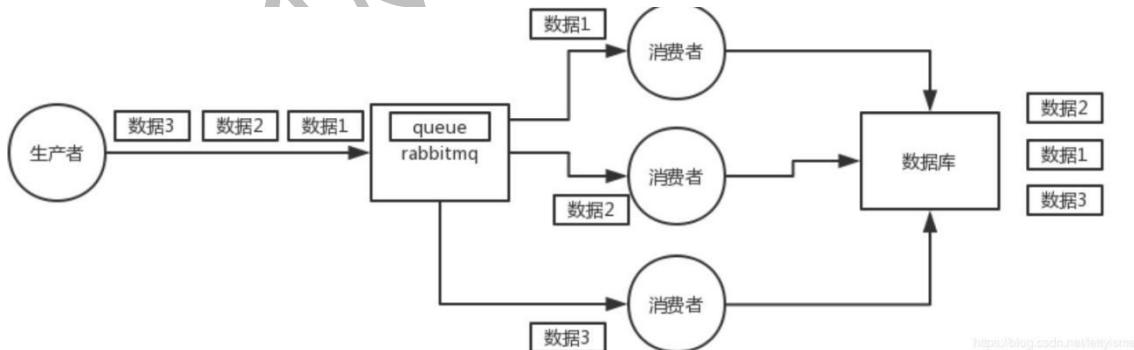
MQ中丢失：就是 RabbitMQ 自己弄丢了数据，这个你必须开启 RabbitMQ 的持久化，就是消息写入之后会持久化到磁盘，哪怕是 RabbitMQ 自己挂了，恢复之后会自动读取之前存储的数据，一般数据不会丢。设置持久化有两个步骤：创建 queue 的时候将其设置为持久化，这样就可以保证 RabbitMQ 持久化 queue 的元数据，但是不会持久化 queue 里的数据。第二个是发送消息的时候将消息的 deliveryMode 设置为 2，就是将消息设置为持久化的，此时 RabbitMQ 就会将消息持久化到磁盘上去。必须要同时设置这两个持久化才行，RabbitMQ 哪怕是挂了，再次重启，也会从磁盘上重启恢复 queue，恢复这个 queue 里的数据。持久化可以跟生产者那边的confirm机制配合起来，只有消息被持久化到磁盘之后，才会通知生产者ack了，所以哪怕是在持久化到磁盘之前，RabbitMQ 挂了，数据丢了，生产者收不到ack，你也是可以自己重发的。注意，哪怕是你给 RabbitMQ 开启了持久化机制，也有一种可能，就是这个消息写到了 RabbitMQ 中，但是还没来得及持久化到磁盘上，结果不巧，此时 RabbitMQ 挂了，就会导致内存里的一点点数据丢失。

消费端丢失：你消费的时候，刚消费到，还没处理，结果进程挂了，比如重启了，那么就尴尬了，RabbitMQ 认为你都消费了，这数据就丢了。这个时候得用 RabbitMQ 提供的ack机制，简单来说，就是你关闭 RabbitMQ 的自动ack，可以通过一个 api 来调用就行，然后每次你自己代码里确保处理完的时候，再在程序里ack一把。这样的话，如果你还没处理完，不就没有ack？那 RabbitMQ 就认为你还没处理完，这个时候 RabbitMQ 会把这个消费分配给别的 consumer 去处理，消息是不会丢的。



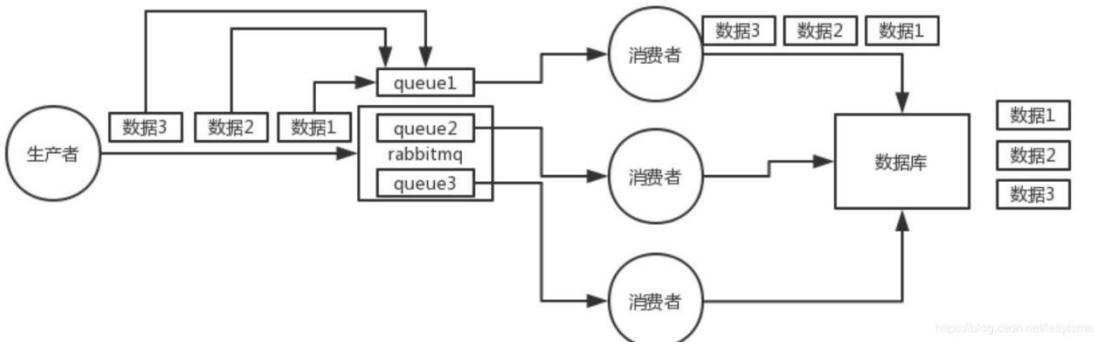
6、如何保证消息的顺序性

先看看顺序会错乱的场景：RabbitMQ：一个 queue，多个 consumer，这不明显乱了；



解决：

拆分多个 queue，每个 queue 一个 consumer，就是多一些 queue 而已，确实是麻烦点；或者就一个 queue 但是对应一个 consumer，然后这个 consumer 内部用内存队列做排队，然后分发给底层不同的 worker 来处理。



7、如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决？

消息积压处理办法：临时紧急扩容：

先修复 consumer 的问题，确保其恢复消费速度，然后将现有 consumer 都停掉。

新建一个 topic，partition 是原来的 10 倍，临时建立好原先 10 倍的 queue 数量。

然后写一个临时的分发数据的 consumer 程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的 10 倍数量的 queue。

接着临时征用 10 倍的机器来部署 consumer，每一批 consumer 消费一个临时 queue 的数据。这种做法相当于临时将 queue 资源和 consumer 资源扩大 10 倍，以正常的 10 倍速度来消费数据。

等快速消费完积压数据之后，得恢复原先部署的架构，重新用原先的 consumer 机器来消费消息。

MQ中消息失效：假设你用的是 RabbitMQ，RabbitMQ 是可以设置过期时间的，也就是 TTL。如果消息在 queue 中积压超过一定的时间就会被 RabbitMQ 给清理掉，这个数据就没了。那这就是第二个坑了。这就不是说数据会大量积压在 mq 里，而是大量的数据会直接搞丢。我们可以采取一个方案，就是批量重导，这个我们之前线上也有类似的场景干过。就是大量积压的时候，我们当时就直接丢弃数据了，然后等过了高峰期以后，比如大家一起喝咖啡熬夜到晚上12点以后，用户都睡觉了。这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入 mq 里面去，把白天丢的数据给他补回来。也只能是这样了。假设 1 万个订单积压在 mq 里面，没有处理，其中 1000 个订单都丢了，你只能手动写程序把那 1000 个订单给查出来，手动发到 mq 里去再补一次。

mq消息队列块满了：如果消息积压在 mq 里，你很长时间都没有处理掉，此时导致 mq 都快写满了，咋办？这个还有别的办法吗？没有，谁让你第一个方案执行的太慢了，你临时写程序，接入数据来消费，消费一个丢弃一个，都不要了，快速消费掉所有的消息。然后走第二个方案，到了晚上再补数据吧。

8、设计MQ的思路

比如说这个消息队列系统，我们从以下几个角度来考虑一下：

首先这个 mq 得支持可伸缩性吧，就是需要的时候快速扩容，就可以增加吞吐量和容量，那怎么搞？设计个分布式的系统呗，参照一下 kafka 的设计理念，broker -> topic -> partition，每个 partition 放一个机器，就存一部分数据。如果现在资源不够了，简单啊，给 topic 增加 partition，然后做数据迁移，增加机器，不就可以存放更多数据，提供更高的吞吐量了？

其次你得考虑一下这个 mq 的数据要不要落地磁盘吧？那肯定要了，落磁盘才能保证别进程挂了数据就丢了。那落磁盘的时候怎么落啊？顺序写，这样就没有磁盘随机读写的寻址开销，磁盘顺序读写的性能是很高的，这就是 kafka 的思路。

其次你考虑一下你的 mq 的可用性啊？这个事儿，具体参考之前可用性那个环节讲解的 kafka 的高可用保障机制。多副本 -> leader & follower -> broker 挂了重新选举 leader 即可对外服务。

能不能支持数据 0 丢失啊？可以的，参考我们之前说的那个 kafka 数据零丢失方案。

数据结构与算法篇

1、常用的数据结构

原文：[The top data structures you should know for your next coding interview](#)

译者：[Fundebug](#)

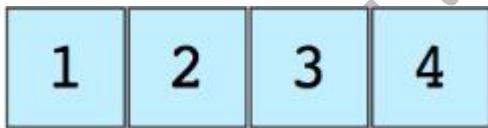
我们首先列出最常用的数据结构

- 数组
- 堆栈
- 队列
- 链表
- 树图
- 字典树
- 哈希表

1. 数组

数组(**Array**)大概是最简单，也是最常用的数据结构了。其他数据结构，比如栈和队列都是由数组衍生出来的。

下图展示了 1 个数组，它有 4 个元素：



每一个数组元素的位置由数字编号，称为下标或者索引(index)。大多数编程语言的数组第一个元素的下标是 0。

根据维度区分，有 2 种不同的数组：

- 一维数组(如上图所示)
- 多维数组(数组的元素为数组)

数组的基本操作

- Insert - 在某个索引处插入元素
- Get - 读取某个索引处的元素
- Delete - 删除某个索引处的元素
- Size - 获取数组的长度

常见数组代码面试题

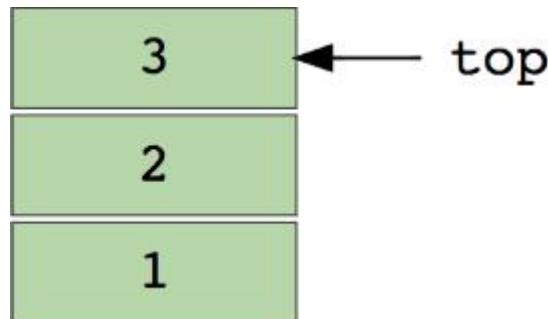
- 查找数组中第二小的元素: <https://www.geeksforgeeks.org/to-find-smallest-and-second-smallest-element-in-an-array/>
- 查找第一个没有重复的数组元素: <https://www.geeksforgeeks.org/non-repeating-element/>
- 合并 2 个排序好的数组: <https://www.geeksforgeeks.org/merge-two-sorted-arrays/>
- 重新排列数组中的正数和负数: <https://www.geeksforgeeks.org/rearrange-positive-and-negative-numbers-publish/>

2. 栈

撤回, 即 **Ctrl+Z**, 是我们最常见的操作之一, 大多数应用都会支持这个功能。你知道它是怎么实现的吗? 答案是这样的: 把之前的[应用状态\(限制个数\)](#)保存到内存中, 最近的状态放到第一个。这时, 我们需要栈(**stack**)来实现这个功能。

栈中的元素采用 LIFO (Last In First Out), 即后进先出。

下图的栈有 3 个元素, 3 在最上面, 因此它会被第一个移除:



栈的基本操作

- Push — 在栈的最上方插入元素
- Pop — 返回栈最上方的元素, 并将其删除
- isEmpty — 查询栈是否为空
- Top — 返回栈最上方的元素, 并不删除

常见的栈代码面试题

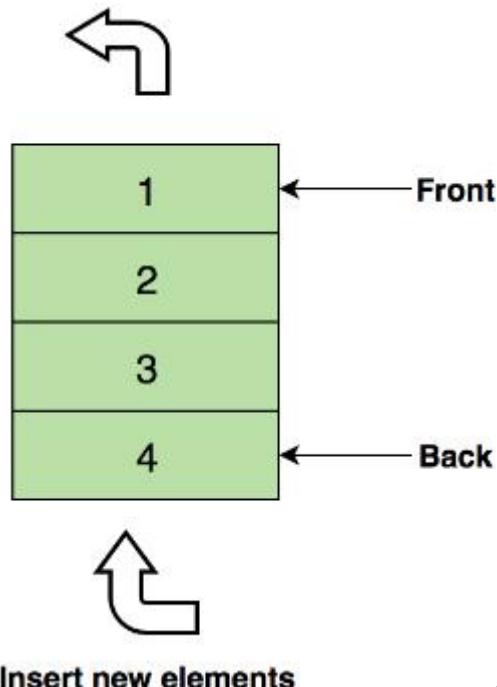
- 使用栈计算后缀表达式: <https://www.geeksforgeeks.org/stack-set-4-evaluation-postfix-expression/>
- 使用栈为栈中的元素排序: <https://www.geeksforgeeks.org/sort-stack-using-temporary-stack/>
- 符串中的括号是否匹配正确: <https://www.geeksforgeeks.org/check-for-balanced-parentheses-in-an-expression/>

3. 队列

队列(**Queue**)与栈类似, 都是采用线性结构存储数据。它们的区别在于, 栈采用 LIFO 方式, 而队列采用先进先出, 即**FIFO(First in First Out)**。

下图展示了一个队列, 1 是最上面的元素, 它会被第一个移除:

Remove previous elements



队列的基本操作

- Enqueue — 在队列末尾插入元素
- Dequeue — 将队列第一个元素删
- 除isEmpty — 查询队列是否为空
- Top — 返回队列的第一个元素

常见的队列代码面试题

- 使用队列实现栈: <https://www.geeksforgeeks.org/implement-stack-using-queue/>
- 倒转队列的前 K 个元素: <https://www.geeksforgeeks.org/reversing-first-k-elements-queue/>
- 将 1 到 n 转换为二进制: <https://www.geeksforgeeks.org/interesting-method-generating-binary-numbers-1-n/>

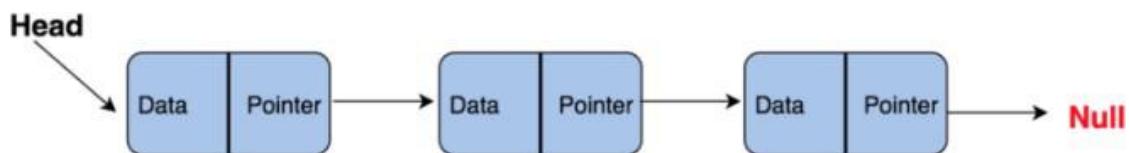
4. 链表

链表(Linked List)也是线性结构, 它与数组看起来非常像, 但是它们的内存分配方式、内部结构和插入删除操作方式都不一样。

链表是一系列节点组成的链, 每一个节点保存了数据以及指向下一个节点的指针。链表头指针指向第一个节点, 如果链表为空, 则头指针为空或者为 null。

链表可以用来实现文件系统、哈希表和邻接表。

下图展示了一个链表, 它有 3 个节点:



链表分为 2 种:

- 单向链表
- 双向链表

链表的基本操作

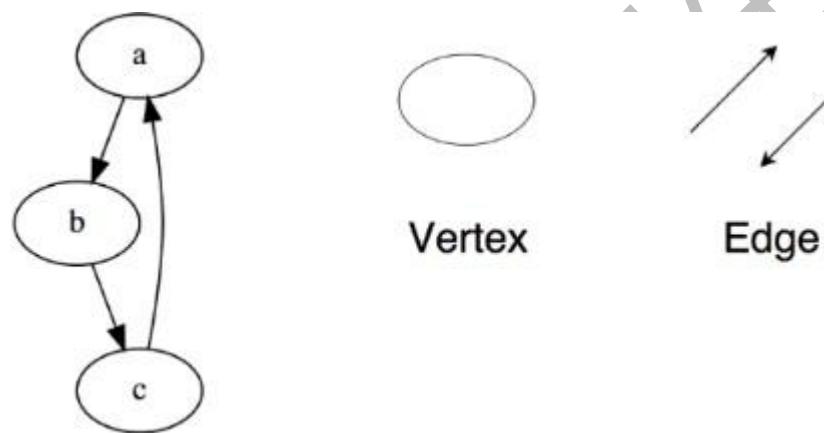
- InsertAtEnd — 在链表结尾插入元素
- InsertAtHead — 在链表开头插入元素
- Delete — 删除链表的指定元素
- DeleteAtHead — 删除链表第一个元素
- Search — 在链表中查询指定元素
- isEmpty — 查询链表是否为空

常见的队列代码面试题

- 倒转 1 个链表: <https://www.geeksforgeeks.org/reverse-a-linked-list/>
- 检查链表中是否存在循环: <https://www.geeksforgeeks.org/detect-loop-in-a-linked-list/>
- 返回链表倒数第 N 个元素: <https://www.geeksforgeeks.org/nth-node-from-the-end-of-a-linked-list/>
- 移除链表中的重复元素: <https://www.geeksforgeeks.org/remove-duplicates-from-an-unsorted-linked-list/>

5. 图

图(graph)由多个节点(vertex)构成，节点之间通过互相连接组成一个网络。 (x, y) 表示一条边(edge)，它表示节点 x 与 y 相连。边可能会有值(weight/cost)。



图分为两种:

- 无向图
- 有向图

在编程语言中，图有可能有以下两种形式表示：

- 邻接矩阵(Adjacency Matrix)
- 邻接表(Adjacency List)

遍历图有两周算法

- 广度优先搜索(Breadth First Search)
- 深度优先搜索(Depth First Search)

常见的图代码面试题

- 实现广度优先搜索: <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>
- 度优先搜索: <https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/>
- 检查图是否为树: <https://www.geeksforgeeks.org/check-given-graph-tree/>
- 统计图中边的个数: <https://www.geeksforgeeks.org/count-number-edges-undirected-graph/>

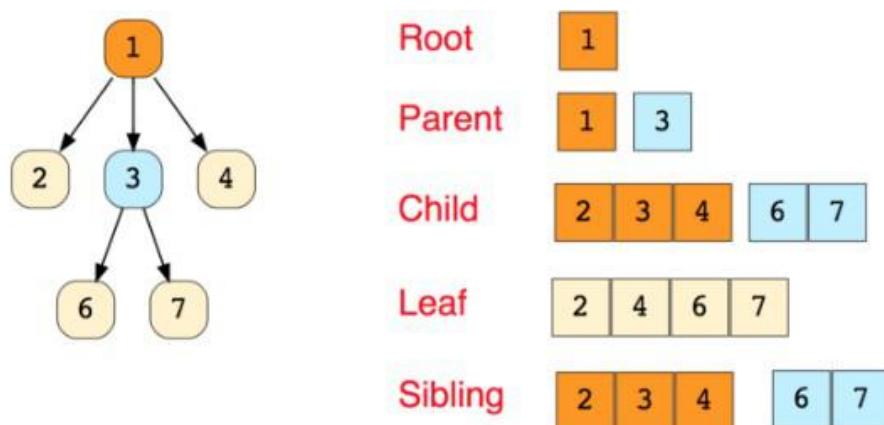
- 使用 Dijkstra 算法查找两个节点之间的最短距离: <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>

6. 树

树(**Tree**)是一个分层的数据结构，由节点和连接节点的边组成。树是一种特殊的图，它与图最大的区别是没有循环。

树被广泛应用在人工智能和一些复杂算法中，用来提供高效的存储结构。

下图是一个简单的树以及与树相关的术语：



树有很多分类：

- N 叉树(N-ary Tree)
- 平衡树(Balanced Tree)
- 二叉树(Binary Tree)
- 二叉查找树(Binary Search Tree)
- 平衡二叉树(AVL Tree)
- 红黑树(Red Black Tree)
- 2-3 树(2-3 Tree)

其中，二叉树和二叉查找树是最常用的树。

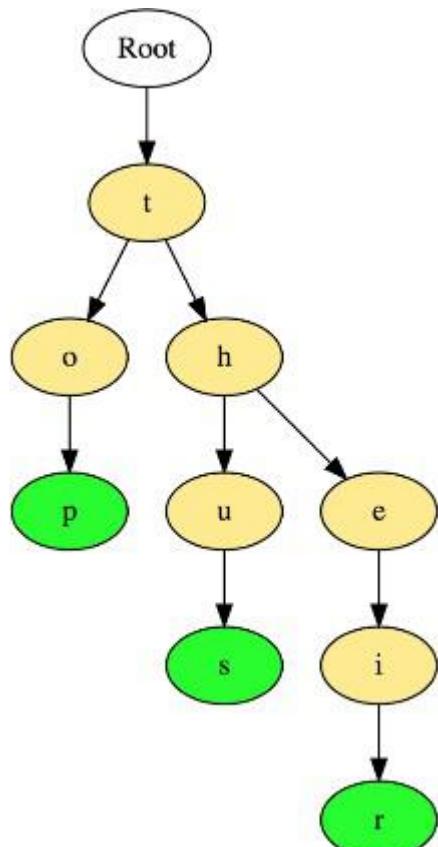
常见的树代码面试题

- 计算树的高度：<https://www.geeksforgeeks.org/write-a-c-program-to-find-the-maximum-depth-or-height-of-a-tree/>
- 查找二叉平衡树中第 K 大的元素：<https://www.geeksforgeeks.org/kth-largest-element-in-bst-when-modification-to-bst-is-not-allowed/>
- 查找树中与根节点距离为 k 的节点：<https://www.geeksforgeeks.org/print-nodes-at-k-distance-from-root/>
- 查找二叉树中某个节点所有祖先节点：<https://www.geeksforgeeks.org/print-ancestors-of-a-given-node-in-binary-tree/>

7. 前缀树

前缀树(**Prefix Trees** 或者 **Trie**)与树类似，用于处理字符串相关的问题时非常高效。它可以实现快速检索，常用于字典中的单词查询，搜索引擎的自动补全甚至 IP 路由。

下图展示了 “top” , “thus” 和 “their” 三个单词在前缀树中如何存储的：



单词是按照字母从上往下存储，“p”、“s”和“r”节点分别表示“top”、“thus”和“their”的单词结尾。

常见的树代码面试题

- 统计前缀树表示的单词个数：<https://www.geeksforgeeks.org/counting-number-words-trie/>
- 树为字符串数组排序：<https://www.geeksforgeeks.org/sorting-array-strings-words-using-trie/>

8. 哈希表

哈希(**Hash**)将某个对象变换为唯一标识符，该标识符通常用一个短的随机字母和数字组成的字符串来代表。哈希可以用来实现各种数据结构，其中最常用的就是哈希表(**hash table**)。

哈希表通常由数组实现。

哈希表的性能取决于 3 个指标：

- 哈希函数
- 哈希表的大小
- 哈希冲突处理方式

下图展示了有数组实现的哈希表，数组的下标即为哈希值，由哈希函数计算，作为哈希表的键(**key**)，而数组中保存的数据即为值(**value**)：



关注【程序员生活志】
免费获取更多海量资源

3	<key>
	<data>
16	<key>
17	<key>
	<data>

常见的哈希表代码面试题

- 查找数组中对称的组合: <https://www.geeksforgeeks.org/given-an-array-of-pairs-find-all-symmetric-pairs-in-it/>
- 确认某个数组的元素是否为另一个数组元素的子集: <https://www.geeksforgeeks.org/find-whether-an-array-is-subset-of-another-array-set-1/>
- 确认给定的数组是否互斥: <https://www.geeksforgeeks.org/check-two-given-sets-disjoint/>

2、数据里有{1,2,3,4,5,6,7,8,9}，请随机打乱顺序，生成一个新的数组（请以代码实现）

```

import java.util.Arrays;

//打乱数组
public class Demo1 {

    //随机打乱
    public static int[] srand(int[] a)
    { int[] b = new int[a.length];

        for(int i = 0; i < a.length;i++) {
            //随机获取下标
            int tmp = (int)(Math.random()*(a.length - i)); //随机数:[ 0 ,
a.length - i )
            b[i] = a[tmp];

            //将此时a[tmp]的下标移动到靠后的位置
            int change = a[a.length - i - 1];
            a[a.length - i - 1] = a[tmp];
            a[tmp] = change;
        }

        return b;
    }
}

```

```

public static void main(String[] args) {
    int[] a = {1,2,3,4,5,6,7,8,9};
    System.out.println(Arrays.toString(shard(a)));
}

```

3、写出代码判断一个整数是不是2的阶次方（请代码实现，谢绝调用API方法）

```

import java.util.Scanner;

//判断整数是不是2的阶次方
public class Demo2 {

    public static boolean check(int sum)
    { boolean flag = true; //判断标志
        while(sum > 1) {
            if (sum % 2 == 0)
                { sum = sum/2;
            } else {
                flag = false;
                break;
            }
        }
        return flag;
    }

    public static void main(String[] args)
    { Scanner scanner = new Scanner(System.in);
        System.out.println("请输入一个整数:");
        int sum = scanner.nextInt();
        System.out.println(sum + " 是不是2的阶次方: " + check(sum));
    }
}

```

4、假设今日是2015年3月1日，星期日，请算出13个月零6天后是星期几，距离现在多少天（请用代码实现，谢绝调用API方法）

```

import java.util.Scanner;

//算出星期几
public class Demo4 {
    public static String[] week = {"星期日", "星期一", "星期二", "星期三", "星期四", "星期五", "星期六"};
    public static int i = 0;
    public static int[] monthday1 = {0,31,28,31,30,31,30,31,31,30,31,30,31};
    public static int[] monthday2 = {0,31,29,31,30,31,31,30,31,30,31,30,31};

    //查看距离当前天数的差值

```

```

public static String distance(int year,int month,int day,int
newMonth,int newDay) {
    int sum = 0; //设定初始距离天数
    if (month + newMonth >= 12) {
        if (((year + 1) % 4 == 0 && (year + 1) % 100 != 0) || (year + 1) % 400
== 0) {
            sum += 366 + newDay;
            for(int i = 0;i < newMonth -
12;i++) { sum += monthday1[month + i];
        }
    } else {
        sum += 365 + newDay;
        for(int i = 0;i < newMonth -
12;i++) { sum += monthday1[month + i];
    }
}
} else {
    for(int i = 0;i <
newMonth;i++) { sum += monthday1[month + i];
}
sum += newDay;
}
return week[sum%7];
}

public static void main(String[] args)
{ Scanner scanner = new
Scanner(System.in);
System.out.println("请输入当前年份");
int year = scanner.nextInt();
System.out.println("请输入当前月份
"); int month =
scanner.nextInt();
System.out.println("请输入当前天数
"); int day = scanner.nextInt();
System.out.println("请输入当前是星期几：以数字表示，如：星期天 为
0"); int index = scanner.nextInt();
System.out.println("今天是：" + year + "—" + month + "—" + day +
"
" + week[index]);
}

```

```
System.err.println("请输入相隔月份  
"); int newMonth =  
scanner.nextInt();  
System.out.println("请输入剩余天数  
"); int newDay =  
scanner.nextInt();  
  
System.out.println("经过" + newMonth + "月" + newDay + "天后, 是" +  
distance(year,month,day,newMonth,newDay));  
}  
}
```

5、有两个篮子，分别为**A** 和 **B**，篮子**A**里装有鸡蛋，篮子**B**里装有苹果，请用面向对象的思想实现两个篮子里的物品交换（请用代码实现）

```
//面向对象思想实现篮子物品交换
```



关注【程序员生活志】
免费获取更多海量资源

```
public class Demo5 {
    public static void main(String[] args) {
        //创建篮子
        Basket A = new Basket("A");
        Basket B = new Basket("B");

        //装载物品
        A.load("鸡蛋");
        B.load("苹果");

        //交换物品
        A.change(B);

        A.show();
        B.show();
    }
}

class Basket{
    public String name; //篮子名称
    private Goods goods; //篮子中所装
    物品

    public Basket(String name) {
        // TODO Auto-generated constructor
        stub this.name = name;
        System.out.println(name + "篮子被创建
");
    }

    //装物品函数
    public void load(String name)
    {
        goods = new Goods(name);
        System.out.println(this.name + "装载了" + name + "物品");
    }

    public void change(Basket B) {
        System.out.println(this.name + " 和 " + B.name + "中的物品发生了交换
");
        String tmp = this.goods.getName();
        this.goods.setName(B.goods.getName());
        B.goods.setName(tmp);
    }

    public void show() {
        System.out.println(this.name + "中有" + goods.getName() + "物品");
    }
}
```

```
class Goods{  
    private String name; //物品名称  
  
    public String getName()  
    { return name;  
    }  
  
    public void setName(String name)  
    { this.name = name;  
    }  
  
    public Goods(String name) {
```



关注【程序员生活志】
免费获取更多海量资源

```
// TODO Auto-generated constructor stub
this.name = name;
}
```

6、更多算法练习

更多算法练习题，请访问 <https://leetcode-cn.com/problemset/algorithms/>

Linux篇

1、绝对路径用什么符号表示？当前目录、上层目录用什么表示？主目录用什么表示？切换目录用什么命令？

绝对路径：如/etc/init.d

当前目录和上层目录： ./ ../

主目录： ~/

切换目录： cd

2、怎么查看当前进程？怎么执行退出？怎么查看当前路径？

查看当前进程： ps

```
ps -l 列出与本次登录有关的进程信息；  
ps -aux 查询内存中进程信息；  
ps -aux | grep * 查询*进程的详细信息；  
top 查看内存中进程的动态信息；  
kill -9 pid 杀死进程。
```

执行退出： exit

查看当前路径： pwd

3、查看文件有哪些命令

```
vi 文件名 #编辑方式查看，可修改  
cat 文件名 #显示全部文件内容  
more 文件名 #分页显示文件内容  
less 文件名 #与 more 相似，更好的是可以往前翻页
```

tail 文件名 #仅查看尾部，还可以指定行数

head 文件名 #仅查看头部，还可以指定行数

4、列举几个常用的Linux命令

- 列出文件列表: ls 【参数 -a -l】
- 创建目录和移除目录: mkdir rmdir
- 用于显示文件后几行内容: tail, 例如: tail -n 1000: 显示最后1000行
- 打包: tar -xvf
- 打包并压缩: tar -zcvf
- 查找字符串: grep
- 显示当前所在目录: pwd
- 创建空文件: touch
- 编辑器: vim vi

5、你平时是怎么查看日志的?

Linux查看日志的命令有多种: tail、cat、tac、head、echo等, 本文只介绍几种常用的方法。

1、tail

最常用的一种查看方式

命令格式: **tail**[必要参数][选择参数][文件]

-f 循环读取
-q 不显示处理信息
-v 显示详细的处理信息
-c<数目> 显示的字节数
-n<行数> 显示行数
-q, --quiet, --silent 从不输出给出文件名的首部
-s, --sleep-interval=S 与-f合用, 表示在每次反复的间隔休眠S秒

例如:

```
tail -n 10 test.log 查询日志尾部最后10行的日志;  
tail -n +10 test.log 查询10行之后的所有日志;  
tail -fn 10 test.log 循环实时查看最后1000行记录(最常用的)
```

一般还会配合着grep搜索用, 例如:

```
tail -fn 1000 test.log | grep '关键字'
```

如果一次性查询的数据量太大, 可以进行翻页查看, 例如:

```
tail -n 4700 aa.log |more -1000 可以进行多屏显示(ctrl + f 或者 空格键可以快捷键)
```

2、head

跟tail是相反的head是看前多少行日志

```
head -n 10 test.log 查询日志文件中的头10行日志;  
head -n -10 test.log 查询日志文件除了最后10行的其他所有日志;
```

head其他参数参考tail

3、cat

cat 是由第一行到最后一行连续显示在屏幕上

一次显示整个文件：

```
$ cat filename
```

从键盘创建一个文件：

```
$cat > filename
```

将几个文件合并为一个文件：

```
$cat file1 file2 > file 只能创建新文件,不能编辑已有文件
```

将一个日志文件的内容追加到另外一个：

```
$cat -n textfile1 > textfile2
```

清空一个日志文件：

```
$cat : >textfile2
```

注意：> 意思是创建，>>是追加。千万不要弄混了。

cat其他参数参考tail

4、more

more命令是一个基于vi编辑器文本过滤器，它以全屏幕的方式按页显示文本文件的内容，支持vi中的关键字定位操作。more名单中内置了若干快捷键，常用的有H（获得帮助信息），Enter（向下翻滚一行），空格（向下滚动一屏），Q（退出命令）。more命令从前向后读取文件，因此在启动时就加载整个文件。

该命令一次显示一屏文本，满屏后停下来，并且在屏幕的底部出现一个提示信息，给出至今已显示的该文件的百分比：-More- (XX%)

- more的语法：more 文件名
- Enter 向下n行，需要定义，默认为1行
- Ctrl f 向下滚动一屏
- 空格键 向下滚动一屏
- Ctrl b 返回上一屏
- = 输出当前行的行号
- :f 输出文件名和当前行的行号
- v 调用vi编辑器
- !命令 调用Shell，并执行命令
- q退出more

5、sed

这个命令可以查找日志文件特定的一段，根据时间的一个范围查询，可以按照行号和时间范围查询按照行号

```
sed -n "5,10p" filename 这样你就可以只查看文件的第5行到第10行。
```

按照时间段

```
sed -n "/2014-12-17 16:17:20/,/2014-12-17 16:17:36/p" test.log
```

6、less

less命令在查询日志时，一般流程是这样的

```
less log.log
```

shift + G 命令到文件尾部 然后输入 **?** 加上你要搜索的关键字例如 **?**

1213按 **n** 向上查找关键字

shift+n 反向查找关键字

less与**more**类似，使用**less**可以随意浏览文件，而**more**仅能向前移动，不能向后移动，而且 **less** 在查看之前不会加载整个文件。

less log2013.log 查看文件

ps -ef | less ps查看进程信息并通过**less**分页显示

history | less 查看命令历史使用记录并通过**less**分页显示

less log2013.log log2014.log 浏览多个文件

常用命令参数：

less与**more**类似，使用**less**可以随意浏览文件，而**more**仅能向前移动，不能向后移动，而且 **less** 在查看之前不会加载整个文件。

less log2013.log 查看文件

ps -ef | less ps查看进程信息并通过**less**分页显示

history | less 查看命令历史使用记录并通过**less**分页显示

less log2013.log log2014.log 浏览多个文件

常用命令参数：

-b <缓冲区大小> 设置缓冲区的大小

-g 只标志最后搜索的关键词

-i 忽略搜索时的大小写

-m 显示类似**more**命令的百分比

-N 显示每行的行号

-o <文件名> 将**less** 输出的内容在指定文件中保存起来

-Q 不使用警告音

-s 显示连续空行为一行

/字符串: 向下搜索"字符串"的功能

?字符串: 向上搜索"字符串"的功能

n: 重复前一个搜索（与 **/** 或 **?** 有关）

N: 反向重复前一个搜索（与 **/** 或 **?** 有关）

b 向后翻一页

h 显示帮助界面

q 退出**less** 命令

一般人查日志配合应用的其他命令

```
history // 所有的历史记录  
history | grep XXX // 历史记录中包含某些指令的记录  
history | more // 分页查看记录  
history -c // 清空所有的历史记录  
!! 重复执行上一个命令  
查询出来记录后选中 : !323
```

简历篇

原文: <https://www.cnblogs.com/QQ12538552/p/12332620.html>

本篇文章除了教大家用Markdown如何写一份程序员专属的简历，后面还会给大家推荐一些不错的用来写Markdown简历的软件或者网站，以及如何优雅的将Markdown格式转变为PDF格式或者其他格式。

推荐大家使用Markdown语法写简历，然后再将Markdown格式转换为PDF格式后进行简历投递。

如果你对Markdown语法不太了解的话，可以花半个小时简单看一下Markdown语法说明：
<http://www.markdown.cn>。

为什么说简历很重要？

一份好的简历可以在整个申请面试以及面试过程中起到非常好的作用。在不夸大自己能力的情况下，写出一份好的简历也是一项很棒的能力。为什么说简历很重要呢？

先从面试来说

假如你是网申，你的简历必然会经过HR的筛选，一张简历HR可能也就花费10秒钟看一下，然后HR就会决定你这一关是Fail还是Pass。

假如你是内推，如果你的简历没有什么优势的话，就算是内推你的人再用心，也无能为力。

另外，就算你通过了筛选，后面的面试中，面试官也会根据你的简历来判断你究竟是否值得他花费很多时间去面试。

所以，简历就像是我们的一个门面一样，它在很大程度上决定了你能否进入到下一轮的面试中。

再从面试说起

我发现大家比较喜欢看面经，这点无可厚非，但是大部分面经都没告诉你很多问题都是在特定条件下才问的。举个简单的例子：一般情况下你的简历上注明你会的东西才会被问到（Java、数据结构、网络、算法这些基础是每个人必问的），比如写了你会 redis，那面试官就很大概率会问你 redis 的一些问题。比如：redis的常见数据类型及应用场景、redis是单线程为什么还这么快、redis 和 memcached 的区别、redis 内存淘汰机制等等。

所以，首先，你要明确的一点是：你不会的东西就不要写在简历上。另外，你要考虑你该如何才能让你的亮点在简历中凸显出来，比如：你在某某项目做了什么事情解决了什么问题（只要有项目就一定有要解决的问题）、你的某一个项目里使用了什么技术后整体性能和并发量提升了很多等等。

面试和工作是两回事，聪明的人会把面试官往自己擅长的领域领，其他人则被面试官牵着鼻子走。虽说面试和工作是两回事，但是你要想要获得自己满意的 offer，你自身的实力必须要强。

必知必会的几点

大部分公司的HR都说我们不看重学历（骗你的！），但是如果你的学校不出众的话，很难在一堆简历中脱颖而出，除非你的简历上有特别的亮点，比如：某某大厂的实习经历、获得了某某大赛的奖等等。

大部分应届生找工作的硬伤是没有工作经验或实习经历，所以如果你是应届生就不要错过秋招和春招。一旦错过，你后面就极大可能会面临社招，这个时候没有工作经验的你可能就会面临各种碰壁，导致找不到一个好的工作

写在简历上的东西一定要慎重，这是面试官大量提问的地方；

将自己的项目经历完美的展示出来非常重要。

必须了解的两大法则

STAR法则（Situation Task Action Result）

- **Situation**: 事情是在什么情况下发生；
- **Task**: 你是如何明确你的任务的；
- **Action**: 针对这样的情况分析，你采用了什么行动方式；
- **Result**: 结果怎样，在这样的情况下你学习到了什么。

简而言之，STAR法则，就是一种讲述自己故事的方式，或者说，是一个清晰、条理的作文模板。不管是什 么，合理熟练运用此法则，可以轻松的对面试官描述事物的逻辑方式，表现出自己分析阐述问题的清晰性、条理性和逻辑性。

FAB法则（Feature Advantage Benefit）

- **t) Feature**: 是什么；
- **Advantage**: 比别人好在哪些地方；
- **Benefit**: 如果雇佣你，招聘方会得到什么好处。

简单来说，这个法则主要是让你的面试官知道你的优势、招了你之后对公司有什么帮助。

项目经历怎么写

简历上有一两个项目经历很正常，但是真正能把项目经历很好的展示给面试官的非常少。对于项目经历大家可以考虑从如下几点来写：

1. 对项目整体设计的一个感受
2. 在这个项目中你负责了什么、做了什么、担任了什么角色
3. 从这个项目中你学会了那些东西，使用到了那些技术，学会了那些新技术的使用
4. 另外项目描述中，最好可以体现自己的综合素质，比如你是如何协调项目组成员协同开发的或者在遇到某一个棘手的问题的时候你是如何解决的又或者说你在这个项目用了什么技术实现了什么功能比如用redis做缓存提高访问速度和并发量、使用消息队列削峰和降流等等。

专业技能怎么写

先问一下你自己会什么，然后看看你意向的公司需要什么。一般HR可能并不太懂技术，所以他在筛选简历的时候可能就盯着你专业技能的关键词来看。对于公司有要求而你不会的技能，你可以花几天时间学习一下，然后在简历上可以写上自己了解这个技能。比如你可以这样写(下面这部分内容摘自我的简历，大家可以根据自己的情况做一些修改和完善)：

- 计算机网络、数据结构、算法、操作系统等课内基础知识：掌握
- Java 基础知识：掌握
- JVM 虚拟机 (Java内存区域、虚拟机垃圾算法、虚拟垃圾收集器、JVM内存管理)：掌握
- 高并发、高可用、高性能系统开发：掌握
- Struts2、Spring、Hibernate、Ajax、Mybatis、JQuery：掌握
- SSH 整合、SSM 整合、SOA 架构：掌握
- Dubbo：掌握
- Zookeeper：掌握
- 常见消息队列：掌握
- Linux：掌握
- MySQL常见优化手段：掌握
- Spring Boot +Spring Cloud +Docker:了解
- Hadoop 生态相关技术中的 HDFS、Storm、MapReduce、Hive、Hbase：了解
- Python 基础、一些常见第三方库比如OpenCV、wxpy、wordcloud、matplotlib：熟悉

排版注意事项

1. 尽量简洁，不要太花里胡哨；
2. 一些技术名词不要弄错了大小写比如MySQL不要写成mysql，Java不要写成java。这个在我看来还是比较忌讳的，所以一定要注意这个细节；
3. 中文和数字英文之间加上空格的话看起来会舒服一点；

其他一些小tips

1. 尽量避免主观表述，少一点语义模糊的形容词，尽量要简洁明了，逻辑结构清晰。
2. 如果自己有博客或者个人技术栈点的话，写上去会为你加分很多。
3. 如果自己的Github比较活跃的话，写上去也会为你加分很多。
4. 注意简历真实性，一定不要写自己不会的东西，或者带有欺骗性的内容
5. 项目经历建议以时间倒序排序，另外项目经历不在于多，而在于有亮点。
6. 如果内容过多的话，不需要非把内容压缩到一页，保持排版干净整洁就可以了。
7. 简历最后最好能加上：“感谢您花时间阅读我的简历，期待能有机会和您共事。”这句话，显的你会很有礼貌。



关注【程序员生活志】
免费获取更多海量资源