Java基础题

1.Java语言的三大特性

1. 封装：

   首先，属性可用来描述同一类事物的特征，方法可描述一类事物可做的操作。封装就是把属于同一类事物的共性（包括属性与方法）归到一个类中，以方便使用。

   1)概念：封装也称为信息隐藏，是指利用抽象数据类型将数据和基于数据的操作封装在一起，使其构成一个不可分割的独立实体，数据被保护在抽象数据类型的内部，尽可能地隐藏内部的细节，只保留一些对外接口使之与外部发生联系。系统的其他部分只有通过包裹在数据外面的被授权的操作来与这个抽象数据类型交流与交互。也就是说，用户无需知道对象内部方法的实现细节，但可以根据对象提供的外部接口(对象名和参数)访问该对象。
   2)好处：(1)实现了专业的分工。将能实现某一特定功能的代码封装成一个独立的实体后，各程序员可以在需要的时候调用，从而实现了专业的分工。(2)隐藏信息，实现细节。通过控制访问权限可以将可以将不想让客户端程序员看到的信息隐藏起来，如某客户的银行的密码需要保密，只能对该客户开发权限。

2. 继承：
   就是个性对共性的属性与方法的接受，并加入个性特有的属性与方法
   1.概念：一个类继承另一个类，则称继承的类为子类，被继承的类为父类。
   2.目的：实现代码的复用。
   3.理解：子类与父类的关系并不是日常生活中的父子关系，子类与父类而是一种特殊化与一般化的关系，是is-a的关系，子类是父类更加详细的分类。如class dog extends animal,就可以理解为dog is a animal.注意设计继承的时候，若要让某个类能继承，父类需适当开放访问权限，遵循里氏代换原则，即向修改关闭对扩展开放，也就是开-闭原则。
   4.结果：继承后子类自动拥有了父类的属性和方法，但特别注意的是，父类的私有属性和构造方法并不能被继承。
   另外子类可以写自己特有的属性和方法，目的是实现功能的扩展，子类也可以复写父类的方法即方法的重写。

3. 多态：
   多态的概念发展出来，是以封装和继承为基础的。
   多态就是在抽象的层面上实施一个统一的行为，到个体（具体）的层面上时，这个统一的行为会因为个体（具体）的形态特征而实施自己的特征行为。（针对一个抽象的事，对于内部个体又能找到其自身的行为去执行。）
   1.概念：相同的事物，调用其相同的方法，参数也相同时，但表现的行为却不同。
   2.理解：子类以父类的身份出现，但做事情时还是以自己的方法实现。子类以父类的身份出现需要向上转型(upcast)，其中向上转型是由JVM自动实现的，是安全的，但向下转型(downcast)是不安全的，需要强制转换。子类以父类的身份出现时自己特有的属性和方法将不能使用。

2.Java语言主要特性

1. **Java语言是易学的**。Java语言的语法与C语言和C++语言很接近，使得大多数程序员很容易学习和使用Java。
2. **Java语言是强制面向对象的**。Java语言提供类、接口和继承等原语，为了简单起见，只支持类之间的单继承，但支持接口之间的多继承，并支持类与接口之间的实现机制（关键字为implements）。
3. **Java语言是分布式的**。Java语言支持Internet应用的开发，在基本的Java应用编程接口中有一个网络应用编程接口（java net），它提供了用于网络应用编程的类库，包括URL、URLConnection、Socket、ServerSocket等。Java的RMI（远程方法激活）机制也是开发分布式应用的重要手段。
4. **Java语言是健壮的**。Java的强类型机制、异常处理、垃圾的自动收集等是Java程序健壮性的重要保证。对指针的丢弃是Java的明智选择。
5. **Java语言是安全的**。Java通常被用在网络环境中，为此，Java提供了一个安全机制以防恶意代码的攻击。如：安全防范机制（类ClassLoader），如分配不同的名字空间以防替代本地的同名类、字节代码检查。
6. **Java语言是体系结构中立的**。Java程序（后缀为java的文件）在Java平台上被编译为体系结构中立的字节码格式（后缀为class的文件），然后可以在实现这个Java平台的任何系统中运行。
7. **Java语言是解释型的**。如前所述，Java程序在Java平台上被编译为字节码格式，然后可以在实现这个Java平台的任何系统的解释器中运行。（一次编译，到处运行）
8. **Java是性能略高的**。与那些解释型的高级脚本语言相比，Java的性能还是较优的。
9. **Java语言是原生支持多线程的**。在Java语言中，线程是一种特殊的对象，它必须由Thread类或其子（孙）类来创建。

链接:https://blog.csdn.net/shenzixincaiji/article/details/82766104

3. JDK 和 JRE 有什么区别

- JDK：Java Development Kit 的简称，Java 开发工具包，提供了 Java 的开发环境和运行环境。
- JRE：Java Runtime Environment 的简称，Java 运行环境，为 Java 的运行提供了所需环境。

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

4.Java基本数据类型及其封装类

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

Tips:boolean类型占了单独使用是4个字节，在数组中又是1个字节
基本类型所占的存储空间是不变的。这种不变性也是Java具有可移植性的原因之一。
基本类型放在栈中，直接存储值。
所有数值类型都有正负号，没有无符号的数值类型。

为什么需要封装类?
因为泛型类包括预定义的集合，使用的参数都是对象类型，无法直接使用基本数据类型，所以Java又提供了这些基本类型的封装类。

基本类型和对应的封装类由于本质的不同。具有一些区别:
1.基本类型只能按值传递，而封装类按引用传递。
2.基本类型会在栈中创建，而对于对象类型，对象在堆中创建，对象的引用在栈中创建，基本类型由于在栈中，效率会比较高，但是可能存在内存泄漏的问题。

5.如果main方法被声明为private会怎样?

能正常编译，但运行的时候会提示"main方法不是public的"。在idea中如果不用public修饰，则会自动去掉可运行的按钮。

6.说明一下public static void main(String args[])这段声明里每个关键字的作用

public: main方法是Java程序运行时调用的第一个方法，因此它必须对Java环境可见。所以可见性设置为pulic.

static: Java平台调用这个方法时不会创建这个类的一个实例，因此这个方法必须声明为static。

void: main方法没有返回值。

String是命令行传进参数的类型，args是指命令行传进的字符串数组。

7.==与equals的区别

==比较两个对象在内存里是不是同一个对象，就是说在内存里的存储位置一致。两个String对象存储的值是一样的，但有可能在内存里存储在不同的地方 .

==比较的是引用而equals方法比较的是内容。public boolean equals(Object obj) 这个方法是由Object对象提供的，可以由子类进行重写。默认的实现只有当对象和自身进行比较时才会返回true,这个时候和==是等价的。String, BitSet, Date, 和File都对equals方法进行了重写，对两个String对象 而言，值相等意味着它们包含同样的字符序列。对于基本类型的包装类来说，值相等意味着对应的基本类型的值一样。

```
public class EqualsTest {
    public static void main(String[] args) {
      String s1 = “abc”;
      String s2 = s1;
      String s5 = “abc”;
      String s3 = new String(”abc”);
      String s4 = new String(”abc”);
      System.out.println(”== comparison : ” + (s1 == s5));
      System.out.println(”== comparison : ” + (s1 == s2));
      System.out.println(”Using equals method : ” + s1.equals(s2));
      System.out.println(”== comparison : ” + s3 == s4);
      System.out.println(”Using equals method : ” + s3.equals(s4));
     }
}
```

结果：

```
== comparison : true
== comparison : true
Using equals method : true
false
Using equals method :true
```

8.Object有哪些公用方法

Object是所有类的父类，任何类都默认继承Object

clone 保护方法，实现对象的浅复制，只有实现了Cloneable接口才可以调用该方法，否则抛出CloneNotSupportedException异常。

equals 在Object中与==是一样的，子类一般需要重写该方法。

hashCode 该方法用于哈希查找，重写了equals方法一般都要重写hashCode方法。这个方法在一些具有哈希功能的Collection中用到。

getClass final方法，获得运行时类型

wait 使当前线程等待该对象的锁，当前线程必须是该对象的拥有者，也就是具有该对象的锁。 **wait()** 方法一直等待，直到获得锁或者被中断。 **wait(long timeout)** 设定一个超时间隔，如果在规定时间内没有获得锁就返回。

调用该方法后当前线程进入睡眠状态，直到以下事件发生

1、其他线程调用了该对象的notify方法。

2、其他线程调用了该对象的notifyAll方法。

3、其他线程调用了interrupt中断该线程。

4、时间间隔到了。

5、此时该线程就可以被调度了，如果是被中断的话就抛出一个InterruptedException异常。

notify 唤醒在该对象上等待的某个线程。

notifyAll 唤醒在该对象上等待的所有线程。

toString 转换成字符串，一般子类都有重写，否则打印句柄。

链接：https://www.cnblogs.com/remember-forget/p/5971962.html

9.为什么Java里没有全局变量?

全局变量是全局可见的，Java不支持全局可见的变量，因为：全局变量破坏了引用透明性原则。全局变量导致了命名空间的冲突。

10.while循环和do循环有什么不同?

while结构在循环的开始判断下一个迭代是否应该继续。do/while结构在循环的结尾来判断是否将继续下一轮迭代。do结构至少会执行一次循环体。

11.char型变量中能不能存储一个中文汉字?为什么?

可以。Java默认Unicode编码。Unicode码占16位。 char两个字节刚好16位。

作用域	当前类	同一package	子孙类	其他package
public	√	√	√	√
protected	√	√	√	×
default	√	√	×	×
private	√	×	×	×

Tips:不写默认default

13.float f=3.4;是否正确?

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

14.short s1 = 1; s1 = s1 + 1;有错吗？ short s1 = 1; s1 += 1;有错吗?

对于short s1 = 1; s1 = s1 + 1；由于1是int类型，因此s1+1运算结果也是int 型，需要强制转换类型才能赋值给short型。而short s1 = 1; s1 += 1；+=操作符会进行隐式自动类型转换，是 **Java 语言规定的运算符**；Java编译器会对它进行特殊处理，因此可以正确编译。因为s1+= 1;相当于s1 = (short)(s1 + 1)。

15.&和&&的区别?

1. &：(1)按位与；(2)逻辑与。

按位与: 0 & 1 = 0 ; 0 & 0 = 0; 1 & 1 = 1

逻辑与: a == b & b ==c （即使a==b已经是 false了，程序还会继续判断b是否等于c)

2.&&: 短路与

a== b && b== c （当a==b 为false则不会继续判断b是否等与c)

比如判断某对象中的属性是否等于某值，则必须用&&,否则会出现空指针问题。

16.IntegerCache

```
public class IntegerTest {

    public static void main(String[] args) {

        Integer a = 100, b = 100 ,c = 129,d = 129;

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

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

结果:

```
true
false
```

小朋友，你是否有很多问号?

来解释一下:

```
   /**
     * Cache to support the object identity semantics of autoboxing for
values between
     * -128 and 127 (inclusive) as required by JLS.
     *
     * The cache is initialized on first usage.  The size of the cache
     * may be controlled by the {@code -XX:AutoBoxCacheMax=<size>} option.
```

```
     * During VM initialization, java.lang.Integer.IntegerCache.high property
     * may be set and saved in the private system properties in the
     * sun.misc.VM class.
     */

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

        static {
            // high value may be configured by property
            int h = 127;
            String integerCacheHighPropValue =

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

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

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

        private IntegerCache() {}
    }
```

```
 public static Integer valueOf(int i) {
    assert IntegerCache.high >= 127;
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
 }
```

通过源码，我们可以看出, -128~127之间做了缓存。考虑到高频数值的复用场景，这样做还是很合理的，合理优化。最大边界可以通过-XX:AutoBoxCacheMax进行配置。

17.Locale类是什么?

Locale类用来根据语言环境来动态调整程序的输出。

18.Java中final、finally、finalize的区别与用法

1. final
   final是一个修饰符也是一个关键字。

- 被final修饰的类无法被继承
- 对于一个final变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；
  如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。但是它*指向的对象的内容是可变的*。
- 被final修饰的方法将无法被重写，但*允许重载*
  注意：类的private方法会隐式地被指定为final方法。

2. finally
   finally是一个关键字。

- finally在异常处理时提供finally块来执行任何清除操作。不管有没有异常被抛出或者捕获，finally块都会执行，通常用于释放资源。
- finally块正常情况下一定会被执行。但是有至少两个极端情况：
  如果对应的try块没有执行，则这个try块的finally块并不会被执行
  如果在try块中jvm关机，例如system.exit(n)，则finally块也不会执行（都拔电源了，怎么执行）
- finally块中如果有return语句，则会覆盖try或者catch中的return语句，导致二者无法return，所以强烈建议finally块中不要存在return关键字

3. finalize
   finalize()是Object类的protected方法，子类可以覆盖该方法以实现资源清理工作。
   GC在回收对象之前都会调用该方法
   finalize()方法是存在很多问题的：

- java语言规范并不保证finalize方法会被及时地执行，更根本不会保证它们一定会被执行
- finalize()方法可能带来性能问题，因为JVM通常在单独的低优先级线程中完成finalize的执行
- finalize()方法中，可将待回收对象赋值给GC Roots可达的对象引用，从而达到对象再生的目的
- finalize方法最多由GC执行一次（但可以手动调用对象的finalize方法）

19.hashCode()和equals()的区别

下边从两个角度介绍了他们的区别：一个是性能，一个是可靠性。他们之间的主要区别也基本体现在这里。

1.equals()既然已经能实现对比的功能了，为什么还要hashCode()呢?
因为重写的equals（）里一般比较的比较全面比较复杂，这样效率就比较低，而利用hashCode()进行对比，则只要生成一个hash值进行比较就可以了，效率很高。

2.hashCode()既然效率这么高为什么还要equals()呢?
因为hashCode()并不是完全可靠，有时候不同的对象他们生成的hashcode也会一样（生成hash值得公式可能存在的问题），所以hashCode()只能说是大部分时候可靠，并不是绝对可靠，所以我们可以得出（PS：以下两条结论是重点，很多人面试的时候都说不出来）：

equals()相等的两个对象他们的hashCode()肯定相等，也就是用equals()对比是绝对可靠的。

hashCode()相等的两个对象他们的equals()不一定相等，也就是hashCode()不是绝对可靠的。
扩展

1.阿里巴巴开发规范明确规定：

只要重写 equals，就必须重写 hashCode；

因为 Set 存储的是不重复的对象，依据 hashCode 和 equals 进行判断，所以 Set 存储的对象必须重写这两个方法；

如果自定义对象做为 Map 的键，那么必须重写 hashCode 和 equals；

String 重写了 hashCode 和 equals 方法，所以我们可以非常愉快地使用 String 对象作为 key 来使用；

2、什么时候需要重写?
一般的地方不需要重载hashCode，只有当类需要放在HashTable、HashMap、HashSet等等hash结构的集合时才会重载hashCode。

3、那么为什么要重载hashCode呢?
如果你重写了equals，比如说是基于对象的内容实现的，而保留hashCode的实现不变，那么很可能某两个对象明明是“相等”，而hashCode却不一样。

这样，当你用其中的一个作为键保存到hashMap、hasoTable或hashSet中，再以“相等的”找另一个作为键值去查找他们的时候，则根本找不到。

4、为什么equals()相等，hashCode就一定要相等，而hashCode相等，却不要求equals相等?

因为是按照hashCode来访问小内存块，所以hashCode必须相等。
HashMap获取一个对象是比较key的hashCode相等和equals为true。
之所以hashCode相等，却可以equal不等，就比如ObjectA和ObjectB他们都有属性name，那么hashCode都以name计算，所以hashCode一样，但是两个对象属于不同类型，所以equals为false。

5、为什么需要hashCode?

通过hashCode可以很快的查到小内存块。
通过hashCode比较比equals方法快，当get时先比较hashCode，如果hashCode不同，直接返回false。

链接：https://blog.csdn.net/xlgen157387/article/details/88087963

20.深拷贝和浅拷贝的区别是什么?

浅拷贝

（1）、定义

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。即对象的浅拷贝会对“主”对象进行拷贝，但不会复制主对象里面的对象。”里面的对象“会在原来的对象和它的副本之间共享。

简而言之，浅拷贝仅仅复制所考虑的对象，而不复制它所引用的对象

（2）、浅拷贝实例

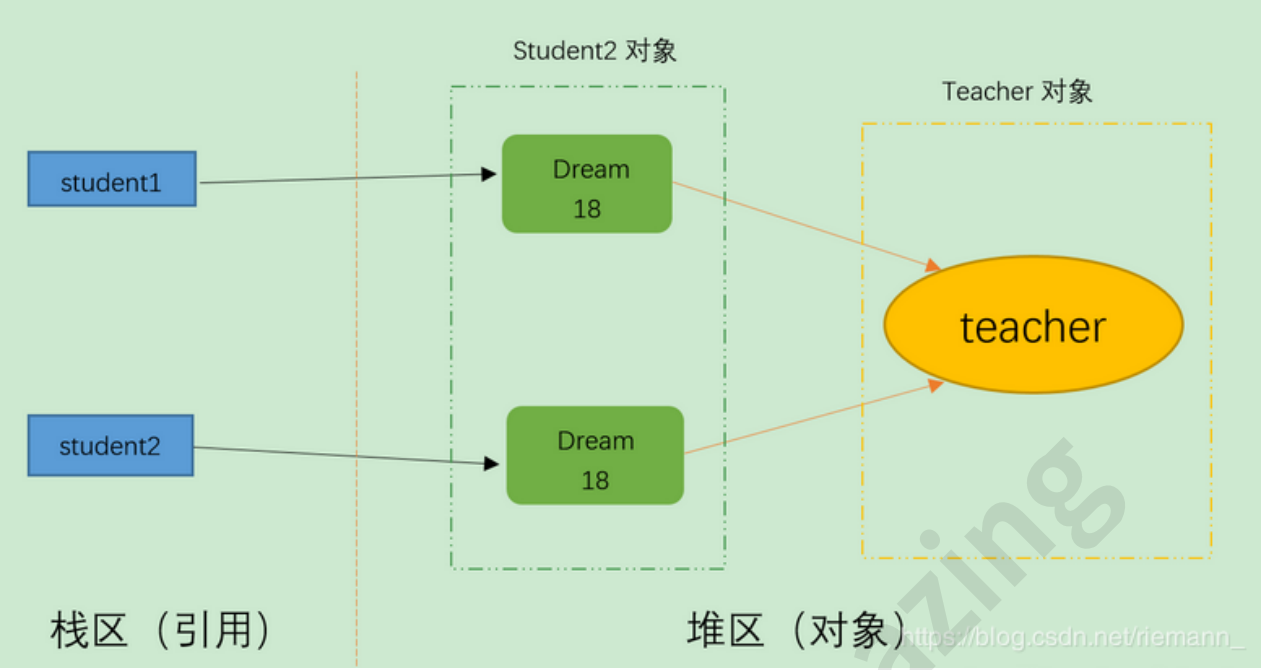
```
package com.test;

public class ShallowCopy {
    public static void main(String[] args) throws CloneNotSupportedException
{
        Teacher teacher = new Teacher();
        teacher.setName("riemann");
        teacher.setAge(27);

        Student2 student1 = new Student2();
        student1.setName("edgar");
        student1.setAge(18);
        student1.setTeacher(teacher);

        Student2 student2 = (Student2) student1.clone();
        System.out.println("拷贝后");
        System.out.println(student2.getName());
        System.out.println(student2.getAge());
        System.out.println(student2.getTeacher().getName());
        System.out.println(student2.getTeacher().getAge());

        System.out.println("修改老师的信息后————");
        // 修改老师的信息
        teacher.setName("Games");
        System.out.println(student1.getTeacher().getName());
        System.out.println(student2.getTeacher().getName());

    }
}

class Teacher implements Cloneable {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
```

```
}

class Student2 implements Cloneable {
    private String name;
    private int age;
    private Teacher teacher;

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public Teacher getTeacher() {
        return teacher;
    }

    public void setTeacher(Teacher teacher) {
        this.teacher = teacher;
    }

    public Object clone() throws CloneNotSupportedException {
        Object object = super.clone();
        return object;
    }
}
```

输出结果:

```
拷贝后
edgar
18
riemann
27
修改老师的信息后————
Games
Games
```

结果分析： 两个引用student1和student2指向不同的两个对象，但是两个引用student1和student2中的两个teacher引用指向的是同一个对象，所以说明是浅拷贝。



深拷贝

（1）、定义

深拷贝是一个整个独立的对象拷贝，深拷贝会拷贝所有的属性,并拷贝属性指向的动态分配的内存。当对象和它所引用的对象一起拷贝时即发生深拷贝。深拷贝相比于浅拷贝速度较慢并且花销较大。

简而言之，深拷贝把要复制的对象所引用的对象都复制了一遍。

（2）、深拷贝实例

```
package com.test;

public class DeepCopy {
    public static void main(String[] args) throws CloneNotSupportedException
{
        Teacher2 teacher = new Teacher2();
        teacher.setName("riemann");
        teacher.setAge(27);

        Student3 student1 = new Student3();
        student1.setName("edgar");
        student1.setAge(18);
        student1.setTeacher(teacher);

        Student3 student2 = (Student3) student1.clone();
        System.out.println("拷贝后");
        System.out.println(student2.getName());
        System.out.println(student2.getAge());
        System.out.println(student2.getTeacher().getName());
        System.out.println(student2.getTeacher().getAge());
```

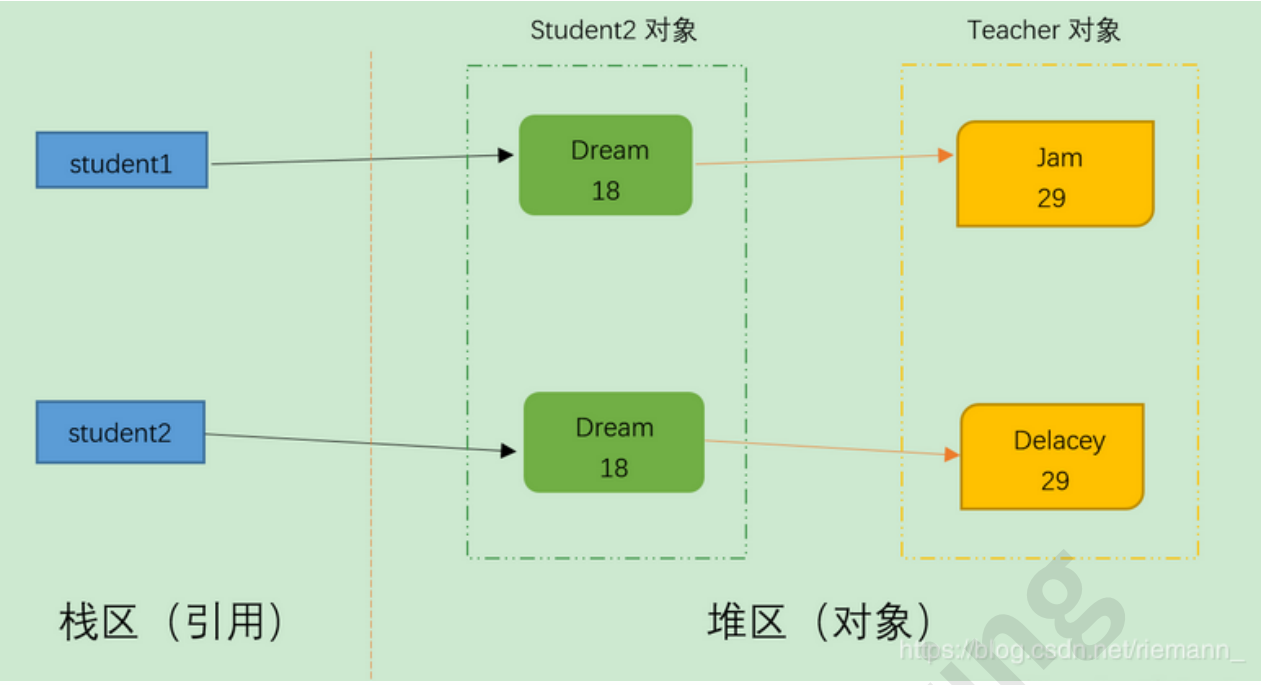
```
        System.out.println("修改老师的信息后————");
        // 修改老师的信息
        teacher.setName("Games");
        System.out.println(student1.getTeacher().getName());
        System.out.println(student2.getTeacher().getName());
    }
}

class Teacher2 implements Cloneable {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

class Student3 implements Cloneable {
    private String name;
    private int age;
    private Teacher2 teacher;

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
```

```
    }

    public Teacher2 getTeacher() {
        return teacher;
    }

    public void setTeacher(Teacher2 teacher) {
        this.teacher = teacher;
    }

    public Object clone() throws CloneNotSupportedException {
        // 浅复制时:
        // Object object = super.clone();
        // return object;

        // 改为深复制:
        Student3 student = (Student3) super.clone();
        // 本来是浅复制，现在将Teacher对象复制一份并重新set进来
        student.setTeacher((Teacher2) student.getTeacher().clone());
        return student;

    }
}
```

输出结果:

```
拷贝后
edgar
18
riemann
27
修改老师的信息后————
Games
riemann
```

结果分析:
两个引用student1和student2指向不同的两个对象，两个引用student1和student2中的两个teacher引用指向的是两个对象，但对teacher对象的修改只能影响student1对象,所以说是深拷贝。



21.Java 中操作字符串都有哪些类？它们之间有什么区别？

String、StringBuffer、StringBuilder。

String 和 StringBuffer、StringBuilder 的区别在于 String 声明的是不可变的对象，每次操作都会生成新的 String 对象，然后将指针指向新的 String 对象，而 StringBuffer、StringBuilder 可以在原有对象的基础上进行操作，所以在经常改变字符串内容的情况下最好不要使用 String。

StringBuffer 和 StringBuilder 最大的区别在于，StringBuffer 是线程安全的，而 StringBuilder 是非线程安全的，但 StringBuilder 的性能却高于 StringBuffer，所以在单线程环境下推荐使用 StringBuilder，多线程环境下推荐使用 StringBuffer。

22.String str="a"与 String str=new String("a")一样吗？

不一样，因为内存的分配方式不一样。

String str="a"; -> 常量池　　String str=new String("a") -> 堆内存

23.抽象类能使用 final 修饰吗？

不能。定义抽象类就是让其他类继承的，而 final修饰的类不能被继承。

24.static关键字5连问

（1）抽象的（abstract）方法是否可同时是静态的（static）？

抽象方法将来是要被重写的，而静态方法是不能重写的，所以这个是错误的。

(2)是否可以从一个静态（static）方法内部发出对非静态方法的调用？

不可以，静态方法只能访问静态成员，非静态方法的调用要先创建对象。

(3) static 可否用来修饰局部变量？

static 不允许用来修饰局部变量

（4）内部类与静态内部类的区别?

静态内部类相对与外部类是独立存在的，在静态内部类中无法直接访问外部类中变量、方法。如果　要访问的话，必须要new一个外部类的对象，使用new出来的对象来访问。但是可以直接访问静态的变量、调用静态的方法；

普通内部类作为外部类一个成员而存在，在普通内部类中可以直接访问外部类属性，调用外部类的方法。

如果外部类要访问内部类的属性或者调用内部类的方法，必须要创建一个内部类的对象，使用该对象访问属性或者调用方法。

如果其他的类要访问普通内部类的属性或者调用普通内部类的方法，必须要在外部类中创建一个普通内部类的对象作为一个属性，外同类可以通过该属性调用普通内部类的方法或者访问普通内部类的属性

如果其他的类要访问静态内部类的属性或者调用静态内部类的方法，直接创建一个静态内部类对象即可。

（5）Java中是否可以覆盖(override) 一个private或者是static的方法?

Java中static方法不能被覆盖，因为方法覆盖是基于运行时动态绑定的，而static方法是编译时静态绑定的。static方法跟类的任何实例都不相关，所以概念上不适用。

25. 重载（Overload）和重写（Override）的区别。重载的方法能否根据返回类型进行区分?

方法的重载和重写都是实现多态的方式，区别在于前者实现的是编译时的多态性，而后者实现的是运行时的多态性。重载发生在一个类中，同名的方法如果有不同的参数列表（参数类型不同、参数个数不同或者二者都不同）则视为重载；重写发生在子类与父类之间，重写要求子类被重写方法与父类被重写方法有相同的参数列表，有兼容的返回类型，比父类被重写方法更好访问，不能比父类被重写方法声明更多的异常（里氏代换原则）。重载对返回类型没有特殊的要求，不能根据返回类型进行区分。

26.Java的四种引用

1、强引用

最普遍的一种引用方式，如String s = "abc"，变量s就是字符串“abc”的强引用，只要强引用存在，则垃圾回收器就不会回收这个对象。

2、软引用（SoftReference）

用于描述还有用但非必须的对象，如果内存足够，不回收，如果内存不足，则回收。一般用于实现内存敏感的高速缓存，软引用可以和引用队列ReferenceQueue联合使用，如果软引用的对象被垃圾回收，JVM就会把这个软引用加入到与之关联的引用队列中。

3、弱引用（WeakReference）

弱引用和软引用大致相同，弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。

4、虚引用（PhantomReference）

就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。 虚引用主要用来跟踪对象被垃圾回收器回收的活动。

虚引用与软引用和弱引用的一个区别在于：

虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。

27.Java 中Comparator 与Comparable 有什么不同?

Comparable 接口用于定义对象的自然顺序，是排序接口，而 comparator 通常用于定义用户定制的顺序，是比较接口。我们如果需要控制某个类的次序，而该类本身不支持排序(即没有实现Comparable接口)，那么我们就可以建立一个“该类的比较器”来进行排序。Comparable 总是只有一个，但是可以有多个 comparator 来定义对象的顺序。

28. Java 序列化,反序列化?

Java 序列化就是指将对象转换为字节序列的过程，反序列化是指将字节序列转换成目标对象的过程。

29.什么情况需要Java序列化?

当 Java 对象需要在网络上传输 或者 持久化存储到文件中时。

30.序列化的实现?

让类实现Serializable接口,标注该类对象是可被序列。

31.如果某些数据不想序列化，如何处理?

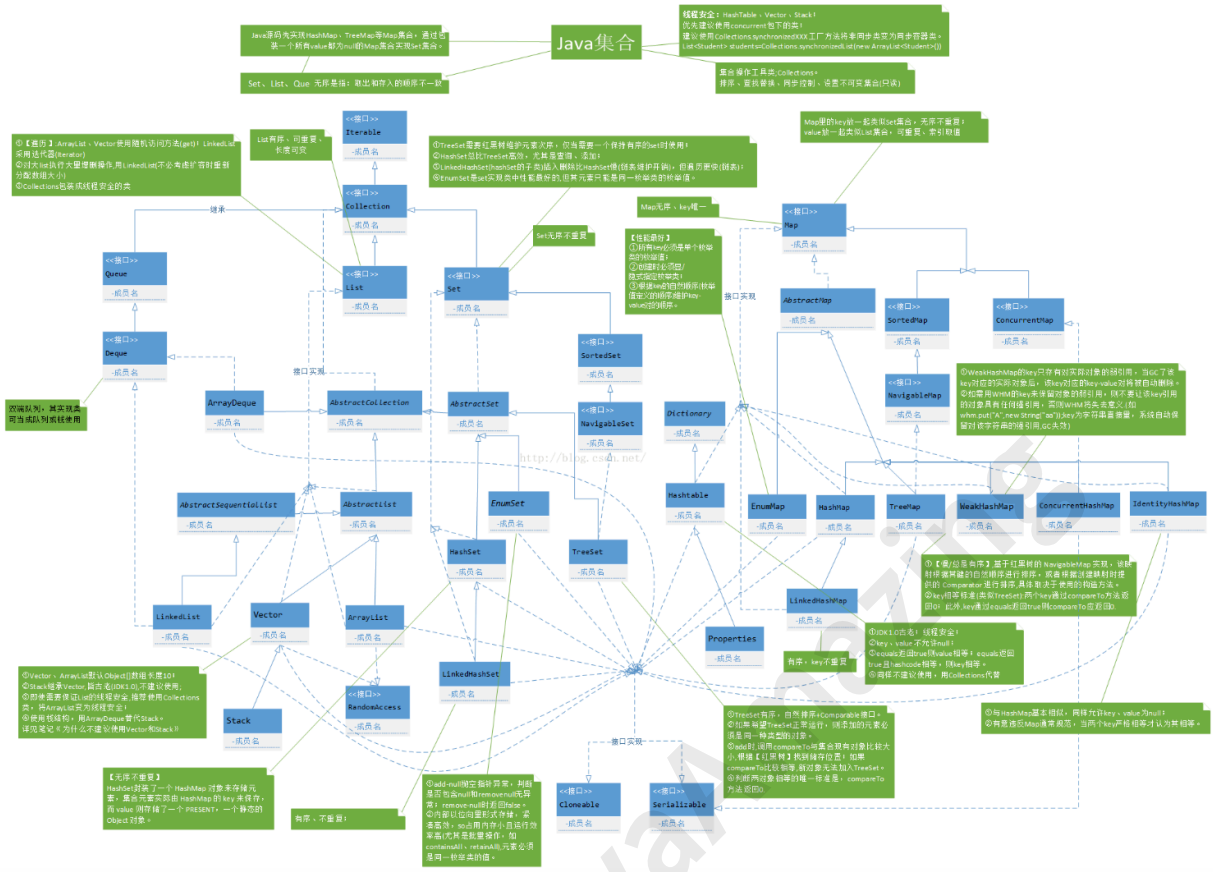
在字段面前加 transient 关键字，例如:

```
transient private String phone;//不参与序列化
```

32.Java泛型和类型擦除?

泛型即参数化类型，在创建集合时，指定集合元素的类型，此集合只能传入该类型的参数。类型擦除：java编译器生成的字节码不包含泛型信息，所以在编译时擦除：1.泛型用最顶级父类替换；2.移除。

java集合



1.Java集合框架的基础接口有哪些?

Collection为集合层级的根接口。一个集合代表一组对象，这些对象即为它的元素。Java平台不提供这个接口任何直接的实现。

Set是一个不能包含重复元素的集合。这个接口对数学集合抽象进行建模，被用来代表集合，就如一副牌。

List是一个有序集合，可以包含重复元素。你可以通过它的索引来访问任何元素。List更像长度动态变换的数组。

Map是一个将key映射到value的对象.一个Map不能包含重复的key：每个key最多只能映射一个value。

一些其它的接口有Queue、Dequeue、SortedSet、SortedMap和ListIterator。

2.Collection 和 Collections 有什么区别?

- Collection 是一个集合接口，它提供了对集合对象进行基本操作的通用接口方法，所有集合都是它的子类，比如 List、Set 等。
- Collections 是一个包装类，包含了很多静态方法，不能被实例化，就像一个工具类，比如提供的排序方法： Collections. sort(list)。

3.List、Set、Map是否继承自Collection接口?

List、Set 是，Map 不是。Map是键值对映射容器，与List和Set有明显的区别，而Set存储的零散的元素且不允许有重复元素（数学中的集合也是如此），List是线性结构的容器，适用于按数值索引访问元素的情形。

4.Collections.sort排序内部原理

在Java 6中Arrays.sort()和Collections.sort()使用的是MergeSort，而在Java 7中，内部实现换成了TimSort，其对对象间比较的实现要求更加严格。

5.List、Set、Map 之间的区别是什么?

List、Set、Map 的区别主要体现在两个方面：元素是否有序、是否允许元素重复。

三者之间的区别，如下表：

		元素有序	允许元素重复
List		是	是
Set	AbstractSet	否	否
	HashSet		
	TreeSet	是（用二叉树排序）	
Map	AbstractMap	否	Key 值必须惟一，value 可重复
	HashMap		
	TreeMap	是（用二叉树排序）	

6.HashMap 和 Hashtable 有什么区别?

（1）HashMap允许key和value为null，而HashTable不允许。

（2）HashTable是同步的，而HashMap不是。所以HashMap适合单线程环境，HashTable适合多线程环境。

（3）在Java1.4中引入了LinkedHashMap，HashMap的一个子类，假如你想要遍历顺序，你很容易从HashMap转向LinkedHashMap，但是HashTable不是这样的，它的顺序是不可预知的。

（4）HashMap提供对key的Set进行遍历，因此它是fail-fast的，但HashTable提供对key的Enumeration进行遍历，它不支持fail-fast。

（5）HashTable被认为是个遗留的类，如果你寻求在迭代的时候修改Map，你应该使用CocurrentHashMap。

7.如何决定使用 HashMap 还是 TreeMap?

对于在 Map 中插入、删除、定位一个元素这类操作，HashMap 是最好的选择，因为相对而言 HashMap 的插入会更快，但如果你要对一个 key 集合进行有序的遍历，那 TreeMap 是更好的选择。

8.说一下 HashMap 的实现原理?

HashMap 基于 Hash 算法实现的，我们通过 put(key,value)存储，get(key)来获取。当传入 key 时，HashMap 会根据 key. hashCode() 计算出 hash 值，根据 hash 值将 value 保存在 bucket 里。当计算出的 hash 值相同时，我们称之为 hash 冲突，HashMap 的做法是用链表和红黑树存储相同 hash 值的 value。当 hash 冲突的个数比较少时，使用链表否则使用红黑树。

9.说一下 HashSet 的实现原理?

HashSet 是基于 HashMap 实现的，HashSet 底层使用 HashMap 来保存所有元素，因此 HashSet 的实现比较简单，相关 HashSet 的操作，基本上都是直接调用底层 HashMap 的相关方法来完成，HashSet 不允许重复的值。

10.ArrayList 和 LinkedList 的区别是什么?

- 数据结构实现：ArrayList 是动态数组的数据结构实现，而 LinkedList 是双向链表的数据结构实现。
- 随机访问效率：ArrayList 比 LinkedList 在随机访问的时候效率要高，因为 LinkedList 是线性的数据存储方式，所以需要移动指针从前往后依次查找。
- 增加和删除效率：在非首尾的增加和删除操作，LinkedList 要比 ArrayList 效率要高，因为 ArrayList 增删操作要影响数组内的其他数据的下标。

  综合来说，在需要频繁读取集合中的元素时，更推荐使用 ArrayList，而在插入和删除操作较多时，更推荐使用 LinkedList。

11.为何Map接口不继承Collection接口?

尽管Map接口和它的实现也是集合框架的一部分，但Map不是集合，集合也不是Map。因此，Map继承Collection毫无意义，反之亦然。

如果Map继承Collection接口，那么元素去哪儿？Map包含key-value对，它提供抽取key或value列表集合的方法，但是它不适合“一组对象”规范。

12.ArrayList和Vector有何异同点?

ArrayList和Vector在很多时候都很类似。

（1）两者都是基于索引的，内部由一个数组支持。

（2）两者维护插入的顺序，我们可以根据插入顺序来获取元素。

（3）ArrayList和Vector的迭代器实现都是fail-fast的。

（4）ArrayList和Vector两者允许null值，也可以使用索引值对元素进行随机访问。

以下是ArrayList和Vector的不同点。

（1）Vector是同步的，而ArrayList不是。然而，如果你寻求在迭代的时候对列表进行改变，你应该使用CopyOnWriteArrayList。

（2）ArrayList比Vector快，它因为有同步，不会过载。

（3）ArrayList更加通用，因为我们可以使用Collections工具类轻易地获取同步列表和只读列表。

13.Array 和 ArrayList 有何区别?

- Array 可以存储基本数据类型和对象，ArrayList 只能存储对象。
- Array 是指定固定大小的，而 ArrayList 大小是自动扩展的。
- Array 内置方法没有 ArrayList 多，比如 addAll、removeAll、iteration 等方法只有 ArrayList 有。

14.在 Queue 中 poll()和 remove()有什么区别?

- 相同点：都是返回第一个元素，并在队列中删除返回的对象。
- 不同点：如果没有元素 poll()会返回 null，而 remove()会直接抛出 NoSuchElementException 异常。

代码示例：

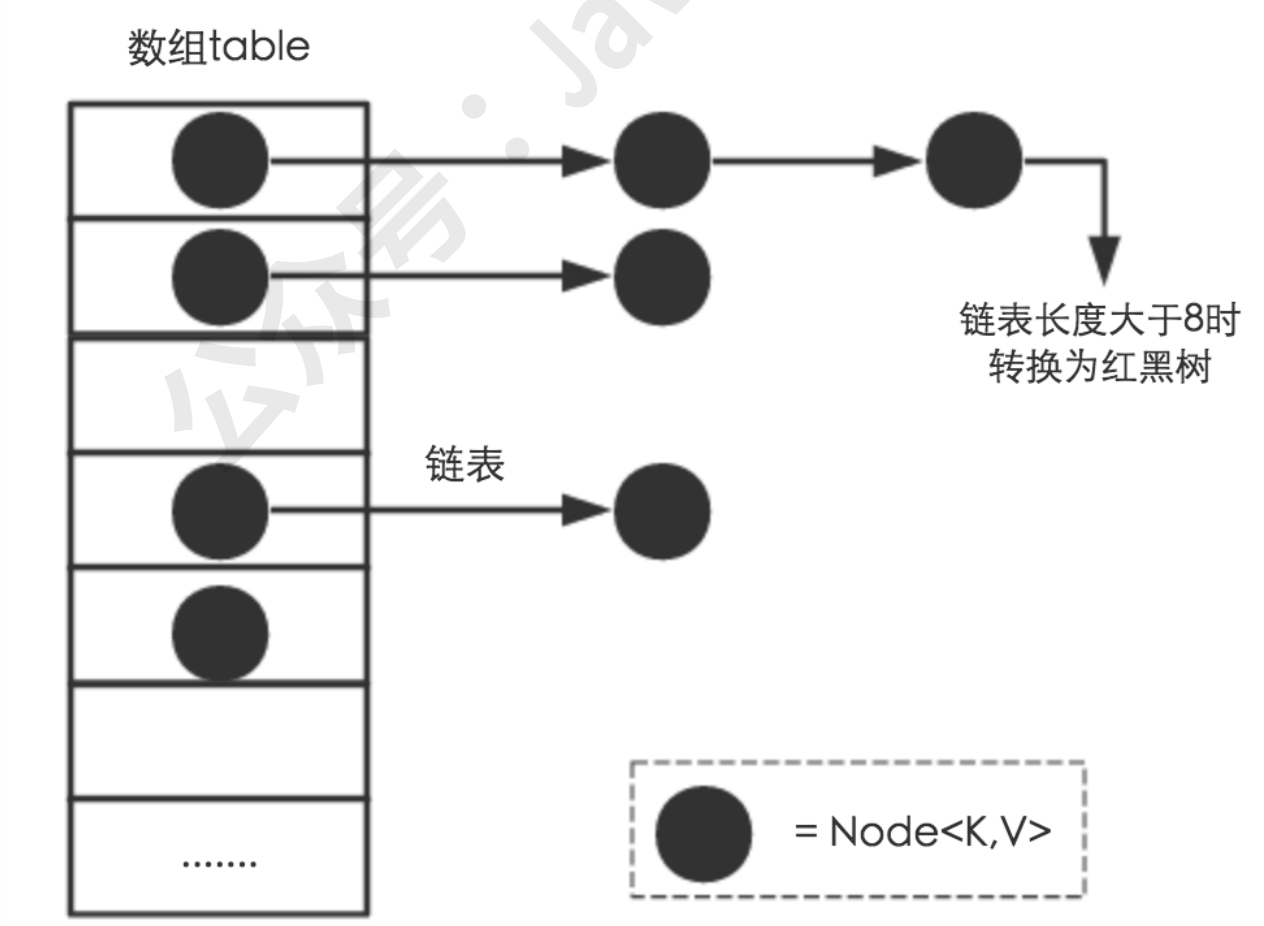
```
Queue<String> queue = new LinkedList<String>();
queue. offer("string"); // add
System. out. println(queue. poll());
System. out. println(queue. remove());
System. out. println(queue. size());
```

15.LinkedHashMap有什么特点?

LinkedHashMap是HashMap的一个子类，保存了记录的插入顺序，在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的，也可以在构造时带参数，按照访问次序排序。

16.HashMap的底层实现原理? （高频问题）

从结构实现来讲，HashMap是数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的，如下如所示。



这里需要讲明白两个问题：数据底层具体存储的是什么？这样的存储方式有什么优点呢？

(1) 从源码可知，HashMap类中有一个非常重要的字段，就是 Node[] table，即哈希桶数组，明显它是一个Node的数组。我们来看Node[JDK1.8]是何物。

```
static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;    //用来定位数组索引位置
        final K key;
        V value;
        Node<K,V> next;   //链表的下一个node

        Node(int hash, K key, V value, Node<K,V> next) { … }
        public final K getKey(){ … }
        public final V getValue() { … }
        public final String toString() { … }
        public final int hashCode() { … }
        public final V setValue(V newValue) { … }
        public final boolean equals(Object o) { … }
}
```

Node是HashMap的一个内部类，实现了Map.Entry接口，本质是就是一个映射(键值对)。上图中的每个黑色圆点就是一个Node对象。

(2) HashMap就是使用哈希表来存储的。哈希表为解决冲突，可以采用开放地址法和链地址法等来解决问题，Java中HashMap采用了链地址法。链地址法，简单来说，就是数组加链表的结合。在每个数组元素上都一个链表结构，当数据被Hash后，得到数组下标，把数据放在对应下标元素的链表上。例如程序执行下面代码：

```
    map.put("美团","小美");
```

系统将调用"美团"这个key的hashCode()方法得到其hashCode 值（该方法适用于每个Java对象），然后再通过Hash算法的后两步运算（高位运算和取模运算，下文有介绍）来定位该键值对的存储位置，有时两个key会定位到相同的位置，表示发生了Hash碰撞。当然Hash算法计算结果越分散均匀，Hash碰撞的概率就越小，map的存取效率就会越高。

如果哈希桶数组很大，即使较差的Hash算法也会比较分散，如果哈希桶数组数组很小，即使好的Hash算法也会出现较多碰撞，所以就需要在空间成本和时间成本之间权衡，其实就是在根据实际情况确定哈希桶数组的大小，并在此基础上设计好的hash算法减少Hash碰撞。那么通过什么方式来控制map使得Hash碰撞的概率又小，哈希桶数组（Node[] table）占用空间又少呢？答案就是好的Hash算法和扩容机制。

在理解Hash和扩容流程之前，我们得先了解下HashMap的几个字段。从HashMap的默认构造函数源码可知，构造函数就是对下面几个字段进行初始化，源码如下：

```
    int threshold;             // 所能容纳的key-value对极限
    final float loadFactor;    // 负载因子
    int modCount;
    int size;
```

首先，Node[] table的初始化长度length(默认值是16)，Load factor为负载因子(默认值是0.75)，threshold是HashMap所能容纳的最大数据量的Node(键值对)个数。threshold = length * Load factor。也就是说，在数组定义好长度之后，负载因子越大，所能容纳的键值对个数越多。

结合负载因子的定义公式可知，threshold就是在此Load factor和length(数组长度)对应下允许的最大元素数目，超过这个数目就重新resize(扩容)，扩容后的HashMap容量是之前容量的两倍。默认的负载因子0.75是对空间和时间效率的一个平衡选择，建议大家不要修改，除非在时间和空间比较特殊的情况下，如果内存空间很多而又对时间效率要求很高，可以降低负载因子Load factor的值；相反，如果内存空间紧张而对时间效率要求不高，可以增加负载因子loadFactor的值，这个值可以大于1。

size这个字段其实很好理解，就是HashMap中实际存在的键值对数量。注意和table的长度length、容纳最大键值对数量threshold的区别。而modCount字段主要用来记录HashMap内部结构发生变化的次数，主要用于迭代的快速失败。强调一点，内部结构发生变化指的是结构发生变化，例如put新键值对，但是某个key对应的value值被覆盖不属于结构变化。

在HashMap中，哈希桶数组table的长度length大小必须为2的n次方(一定是合数)，这是一种非常规的设计，常规的设计是把桶的大小设计为素数。相对来说素数导致冲突的概率要小于合数，具体证明可以参考http://blog.csdn.net/liuqiyao_01/article/details/14475159，Hashtable初始化桶大小为11，就是桶大小设计为素数的应用（Hashtable扩容后不能保证还是素数）。HashMap采用这种非常规设计，主要是为了在取模和扩容时做优化，同时为了减少冲突，HashMap定位哈希桶索引位置时，也加入了高位参与运算的过程。

这里存在一个问题，即使负载因子和Hash算法设计的再合理，也免不了会出现拉链过长的情况，一旦出现拉链过长，则会严重影响HashMap的性能。于是，在JDK1.8版本中，对数据结构做了进一步的优化，引入了红黑树。而当链表长度太长（默认超过8）时，链表就转换为红黑树，利用红黑树快速增删改查的特点提高HashMap的性能，其中会用到红黑树的插入、删除、查找等算法。

功能实现-方法

HashMap的内部功能实现很多，本文主要从根据key获取哈希桶数组索引位置、put方法的详细执行、扩容过程三个具有代表性的点深入展开讲解。

1.确定哈希桶数组索引位置

不管增加、删除、查找键值对，定位到哈希桶数组的位置都是很关键的第一步。前面说过HashMap的数据结构是数组和链表的结合，所以我们当然希望这个HashMap里面的元素位置尽量分布均匀些，尽量使得每个位置上的元素数量只有一个，那么当我们用hash算法求得这个位置的时候，马上就可以知道对应位置的元素就是我们要的，不用遍历链表，大大优化了查询的效率。HashMap定位数组索引位置，直接决定了hash方法的离散性能。先看看源码的实现(方法一+方法二):

```
方法一:
static final int hash(Object key) {   //jdk1.8 & jdk1.7
     int h;
     // h = key.hashCode() 为第一步 取hashCode值
     // h ^ (h >>> 16)  为第二步 高位参与运算
     return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
方法二:
static int indexFor(int h, int length) {  //jdk1.7的源码，jdk1.8没有这个方法，但是实现原理一样的
     return h & (length-1);  //第三步 取模运算
}
```

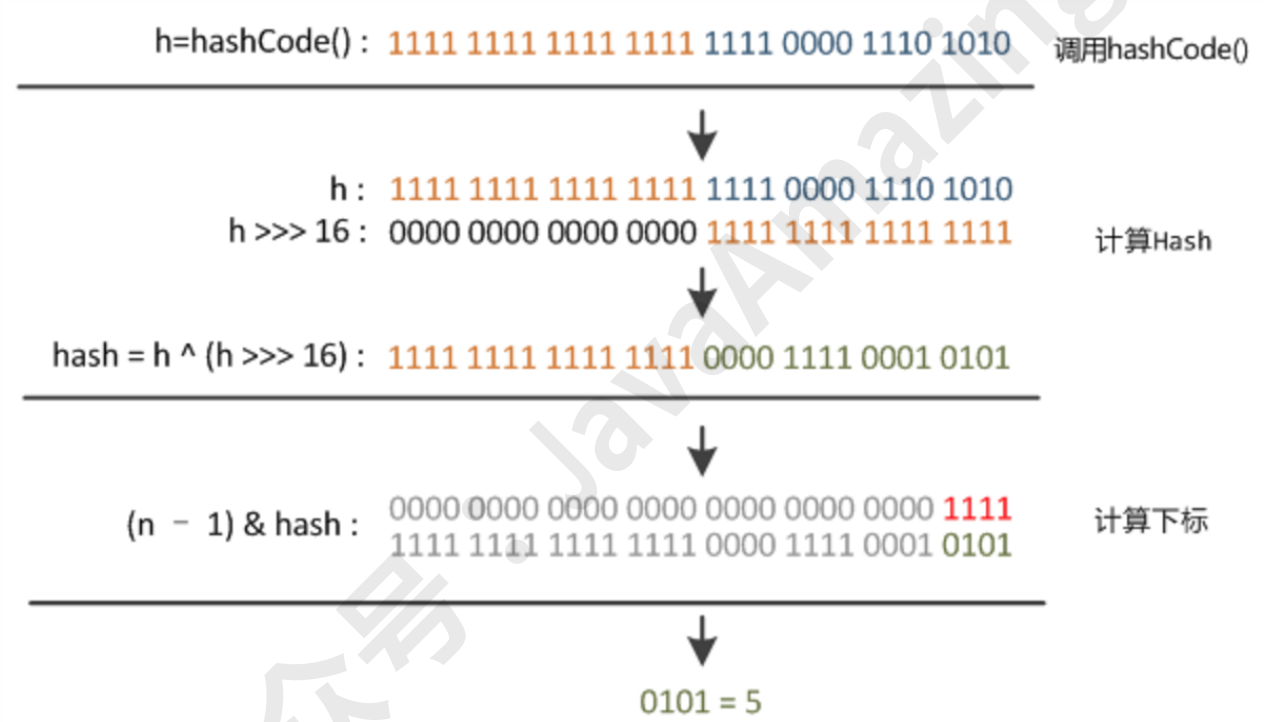
这里的Hash算法本质上就是三步：**取key的hashCode值、高位运算、取模运算。**

对于任意给定的对象，只要它的hashCode()返回值相同，那么程序调用方法一所计算得到的Hash码值总是相同的。我们首先想到的就是把hash值对数组长度取模运算，这样一来，元素的分布相对来说是比较均匀的。但是，模运算的消耗还是比较大的，在HashMap中是这样做的：调用方法二来计算该对象应该保存在table数组的哪个索引处。

这个方法非常巧妙，它通过h & (table.length -1)来得到该对象的保存位，而HashMap底层数组的长度总是2的n次方，这是HashMap在速度上的优化。当length总是2的n次方时，h& (length-1)运算等价于对length取模，也就是h%length，但是&比%具有更高的效率。

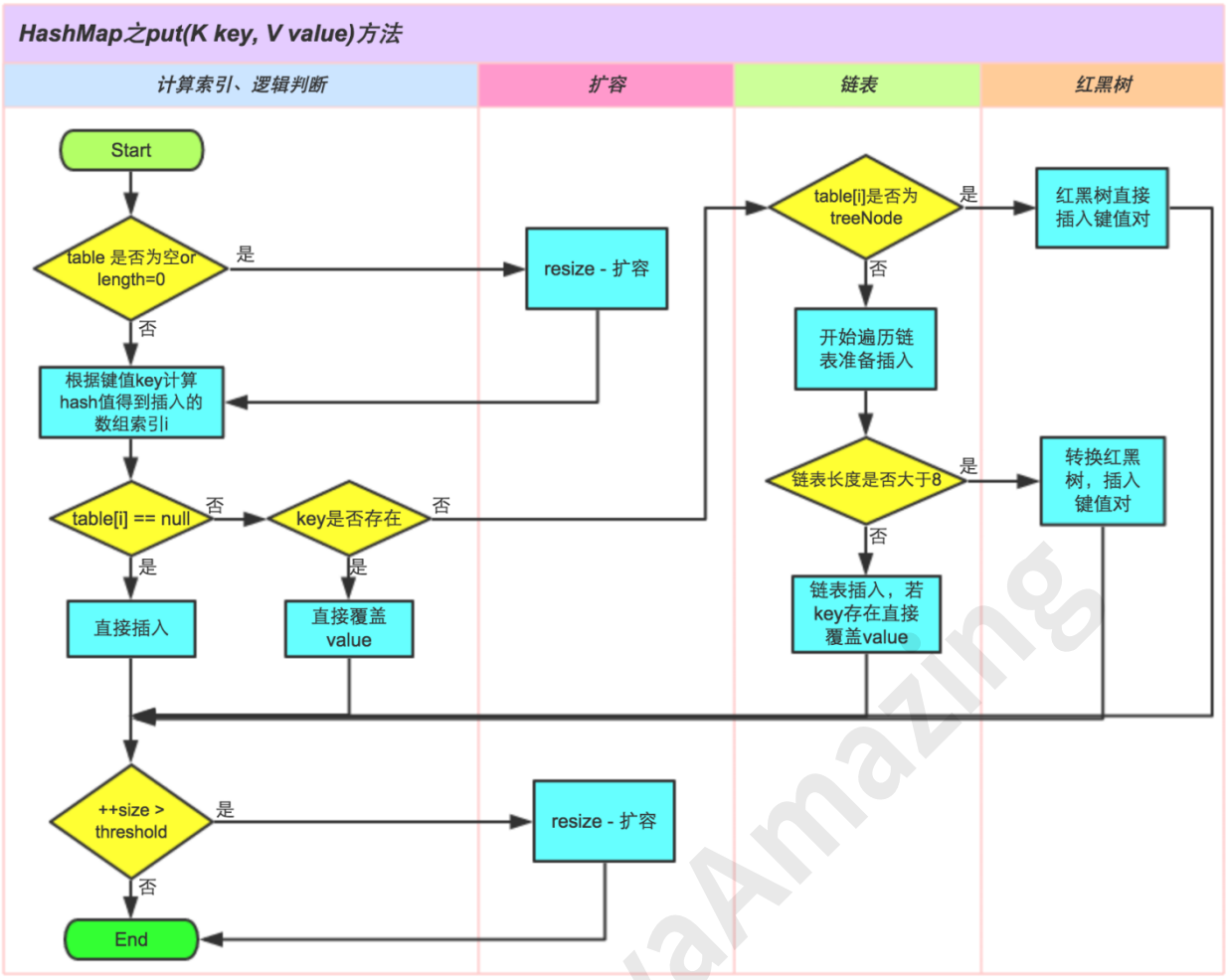
在JDK1.8的实现中，优化了高位运算的算法，通过hashCode()的高16位异或低16位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度、功效、质量来考虑的，这么做可以在数组table的length比较小的时候，也能保证考虑到高低Bit都参与到Hash的计算中，同时不会有太大的开销。

下面举例说明下，n为table的长度。



2.HashMap的put方法

HashMap的put方法执行过程可以通过下图来理解，自己有兴趣可以去对比源码更清楚地研究学习。



①.判断键值对数组table[i]是否为空或为null，否则执行resize()进行扩容；

②.根据键值key计算hash值得到插入的数组索引i，如果table[i]==null，直接新建节点添加，转向⑥，如果table[i]不为空，转向③；

③.判断table[i]的首个元素是否和key一样，如果相同直接覆盖value，否则转向④，这里的相同指的是hashCode以及equals；

④.判断table[i] 是否为treeNode，即table[i] 是否是红黑树，如果是红黑树，则直接在树中插入键值对，否则转向⑤；

⑤.遍历table[i]，判断链表长度是否大于8，大于8的话把链表转换为红黑树，在红黑树中执行插入操作，否则进行链表的插入操作；遍历过程中若发现key已经存在直接覆盖value即可；

⑥.插入成功后，判断实际存在的键值对数量size是否超多了最大容量threshold，如果超过，进行扩容。

JDK1.8HashMap的put方法源码如下:

```
public V put(K key, V value) {
    // 对key的hashCode()做hash
    return putVal(hash(key), key, value, false, true);
}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
```

```
 9     // 步骤①：tab为空则创建
10     if ((tab = table) == null || (n = tab.length) == 0)
11         n = (tab = resize()).length;
12     // 步骤②：计算index，并对null做处理
13     if ((p = tab[i = (n - 1) & hash]) == null)
14         tab[i] = newNode(hash, key, value, null);
15     else {
16         Node<K,V> e; K k;
17         // 步骤③：节点key存在，直接覆盖value
18         if (p.hash == hash &&
19             ((k = p.key) == key || (key != null && key.equals(k))))
20             e = p;
21         // 步骤④：判断该链为红黑树
22         else if (p instanceof TreeNode)
23             e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key,
value);
24         // 步骤⑤：该链为链表
25         else {
26             for (int binCount = 0; ; ++binCount) {
27                 if ((e = p.next) == null) {
28                     p.next = newNode(hash, key,value,null);
                        //链表长度大于8转换为红黑树进行处理
29                     if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
30                         treeifyBin(tab, hash);
31                     break;
32                 }
                    // key已经存在直接覆盖value
33                 if (e.hash == hash &&
34                     ((k = e.key) == key || (key != null &&
key.equals(k))))                                        break;
36                 p = e;
37             }
38         }
39
40         if (e != null) { // existing mapping for key
41             V oldValue = e.value;
42             if (!onlyIfAbsent || oldValue == null)
43                 e.value = value;
44             afterNodeAccess(e);
45             return oldValue;
46         }
47     }

48     ++modCount;
49     // 步骤⑥：超过最大容量 就扩容
50     if (++size > threshold)
51         resize();
52     afterNodeInsertion(evict);
53     return null;
54 }
```

3.扩容机制

扩容(resize)就是重新计算容量，向HashMap对象里不停的添加元素，而HashMap对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。当然Java里的数组是无法自动扩容的，方法是使用一个新的数组代替已有的容量小的数组，就像我们用一个小桶装水，如果想装更多的水，就得换大水桶。

我们分析下resize的源码，鉴于JDK1.8融入了红黑树，较复杂，为了便于理解我们仍然使用JDK1.7的代码，好理解一些，本质上区别不大，具体区别后文再说。

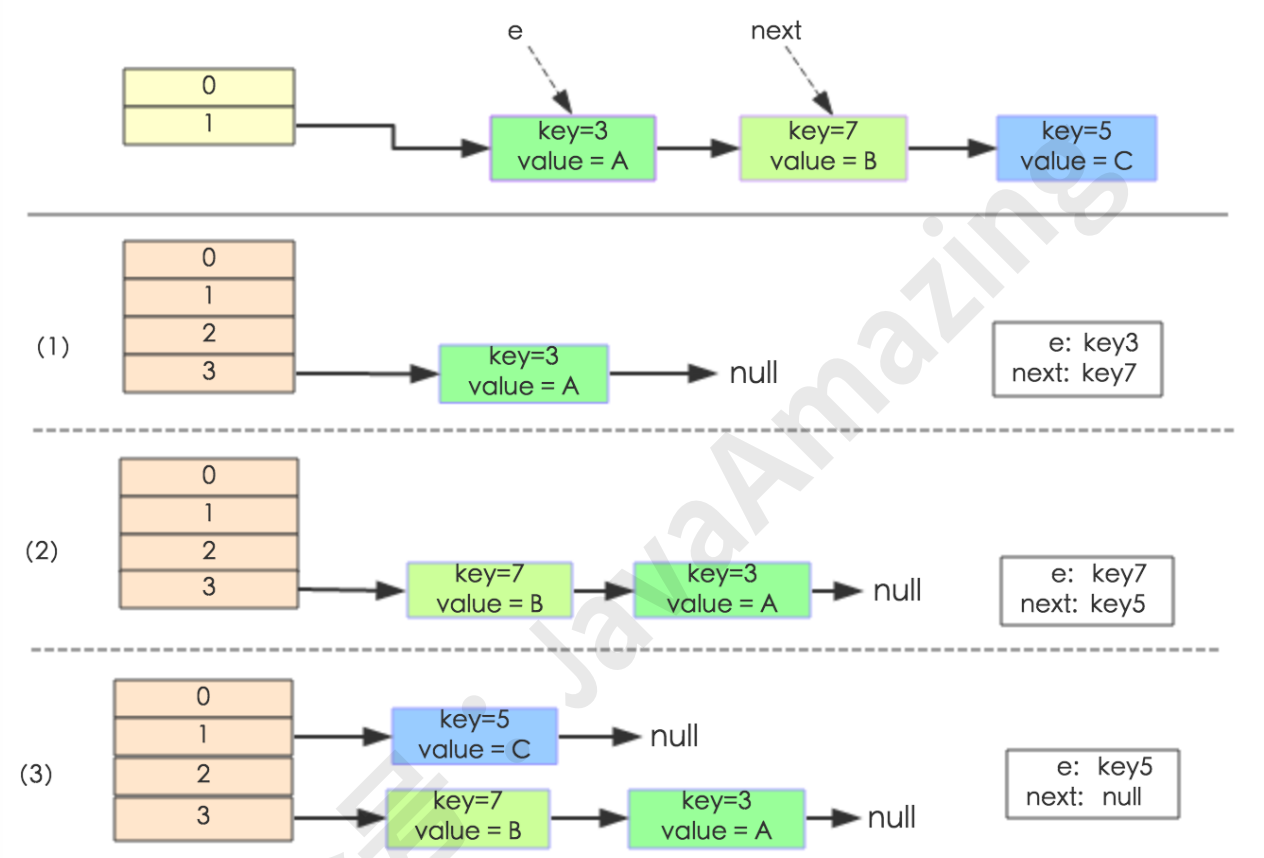
```
 1 void resize(int newCapacity) {   //传入新的容量
 2     Entry[] oldTable = table;    //引用扩容前的Entry数组
 3     int oldCapacity = oldTable.length;
 4     if (oldCapacity == MAXIMUM_CAPACITY) {  //扩容前的数组大小如果已经达到最大(2^30)了
 5         threshold = Integer.MAX_VALUE; //修改阈值为int的最大值(2^31-1)，这样以后就不会扩容了
 6         return;
 7     }
 8
 9     Entry[] newTable = new Entry[newCapacity];  //初始化一个新的Entry数组
10     transfer(newTable);                         //！！将数据转移到新的Entry数组里
11     table = newTable;                           //HashMap的table属性引用新的Entry数组
12     threshold = (int)(newCapacity * loadFactor);//修改阈值
13 }
```

这里就是使用一个容量更大的数组来代替已有的容量小的数组，transfer()方法将原有Entry数组的元素拷贝到新的Entry数组里。

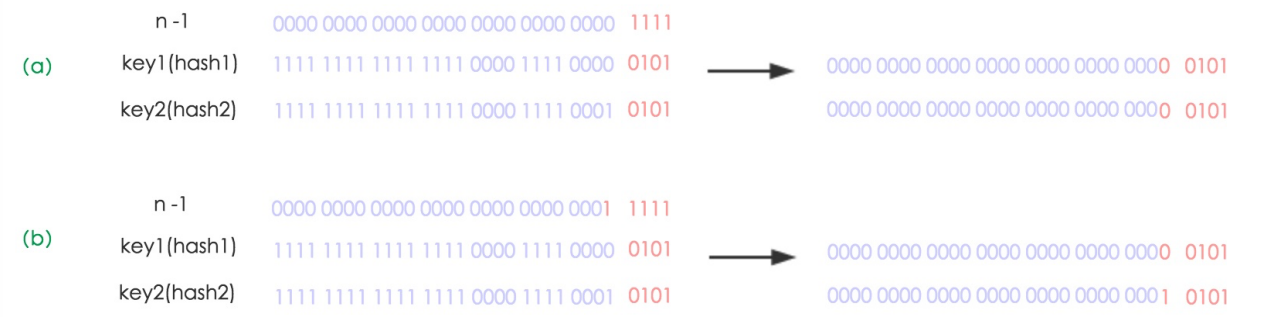
```
 1 void transfer(Entry[] newTable) {
 2     Entry[] src = table;                   //src引用了旧的Entry数组
 3     int newCapacity = newTable.length;
 4     for (int j = 0; j < src.length; j++) { //遍历旧的Entry数组
 5         Entry<K,V> e = src[j];             //取得旧Entry数组的每个元素
 6         if (e != null) {
 7             src[j] = null;//释放旧Entry数组的对象引用（for循环后，旧的Entry数组不再引用任何对象）
 8             do {
 9                 Entry<K,V> next = e.next;
10                 int i = indexFor(e.hash, newCapacity); //！！重新计算每个元素在数组中的位置
11                 e.next = newTable[i]; //标记[1]
12                 newTable[i] = e;      //将元素放在数组上
13                 e = next;             //访问下一个Entry链上的元素
14             } while (e != null);
15         }
16     }
17 }
```

newTable[i]的引用赋给了e.next，也就是使用了单链表的头插入方式，同一位置上新元素总会被放在链表的头部位置；这样先放在一个索引上的元素终会被放到Entry链的尾部(如果发生了hash冲突的话），这一点和Jdk1.8有区别，下文详解。在旧数组中同一条Entry链上的元素，通过重新计算索引位置后，有可能被放到了新数组的不同位置上。

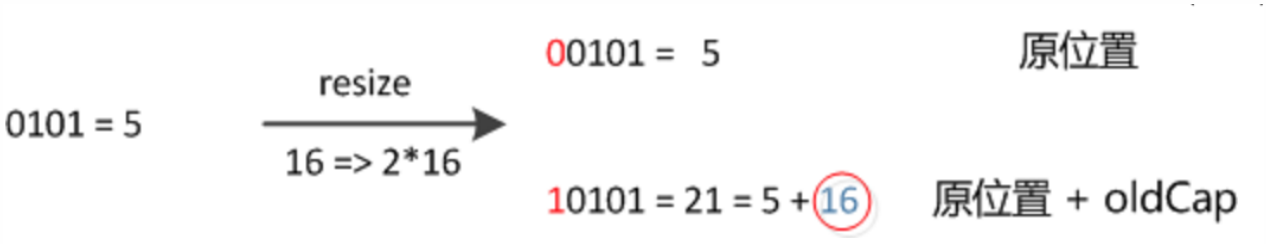
下面举个例子说明下扩容过程。假设了我们的hash算法就是简单的用key mod 一下表的大小（也就是数组的长度）。其中的哈希桶数组table的size=2， 所以key = 3、7、5，put顺序依次为 5、7、3。在mod 2以后都冲突在table[1]这里了。这里假设负载因子 loadFactor=1，即当键值对的实际大小size 大于 table 的实际大小时进行扩容。接下来的三个步骤是哈希桶数组 resize成4，然后所有的Node重新rehash的过程。



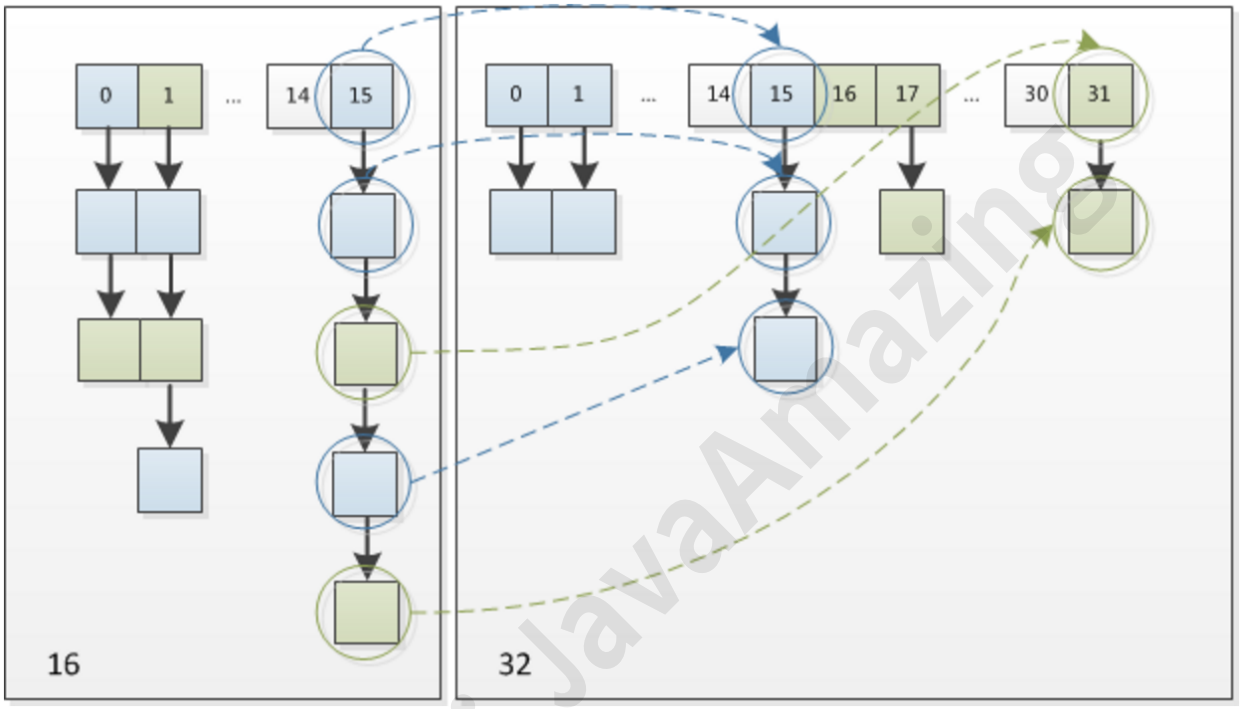
下面我们讲解下JDK1.8做了哪些优化。经过观测可以发现，我们使用的是2次幂的扩展(指长度扩为原来2倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。看下图可以明白这句话的意思，n为table的长度，图（a）表示扩容前的key1和key2两种key确定索引位置的示例，图（b）表示扩容后key1和key2两种key确定索引位置的示例，其中hash1是key1对应的哈希与高位运算结果。



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



因此，我们在扩充HashMap的时候，不需要像JDK1.7的实现那样重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”，可以看看下图为16扩充为32的resize示意图：



这个设计确实非常的巧妙，既省去了重新计算hash值的时间，而且同时，由于新增的1bit是0还是1可以认为是随机的，因此resize的过程，均匀的把之前的冲突的节点分散到新的bucket了。这一块就是JDK1.8新增的优化点。有一点注意区别，JDK1.7中rehash的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置，但是从上图可以看出，JDK1.8不会倒置。有兴趣的同学可以研究下JDK1.8的resize源码，写的很赞，如下：

```
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        // 超过最大值就不再扩充了，就只好随你碰撞去吧
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        // 没超过最大值，就扩充为原来的2倍
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
```

```
16      }
17      else if (oldThr > 0) // initial capacity was placed in threshold
18          newCap = oldThr;
19      else {               // zero initial threshold signifies using
defaults
20          newCap = DEFAULT_INITIAL_CAPACITY;
21          newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
22      }
23      // 计算新的resize上限
24      if (newThr == 0) {
25
26          float ft = (float)newCap * loadFactor;
27          newThr = (newCap < MAXIMUM_CAPACITY && ft <
(float)MAXIMUM_CAPACITY ?
28                    (int)ft : Integer.MAX_VALUE);
29      }
30      threshold = newThr;
31      @SuppressWarnings({"rawtypes", "unchecked"})
32          Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
33      table = newTab;
34      if (oldTab != null) {
35          // 把每个bucket都移动到新的buckets中
36          for (int j = 0; j < oldCap; ++j) {
37              Node<K,V> e;
38              if ((e = oldTab[j]) != null) {
39                  oldTab[j] = null;
40                  if (e.next == null)
41                      newTab[e.hash & (newCap - 1)] = e;
42                  else if (e instanceof TreeNode)
43                      ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
44                  else { // 链表优化重hash的代码块
45                      Node<K,V> loHead = null, loTail = null;
46                      Node<K,V> hiHead = null, hiTail = null;
47                      Node<K,V> next;
48                      do {
49                          next = e.next;
50                          // 原索引
51                          if ((e.hash & oldCap) == 0) {
52                              if (loTail == null)
53                                  loHead = e;
54                              else
55                                  loTail.next = e;
56                              loTail = e;
57                          }
58                          // 原索引+oldCap
59                          else {
60                              if (hiTail == null)
61                                  hiHead = e;
62                              else
63                                  hiTail.next = e;
64                              hiTail = e;
65                          }
```

```
66                      } while ((e = next) != null);
67                      // 原索引放到bucket里
68                      if (loTail != null) {
69                          loTail.next = null;
70                          newTab[j] = loHead;
71                      }
72                      // 原索引+oldCap放到bucket里
73                      if (hiTail != null) {
74                          hiTail.next = null;
75                          newTab[j + oldCap] = hiHead;
76                      }
77                  }
78              }
79          }
80      }
81      return newTab;
82  }
```

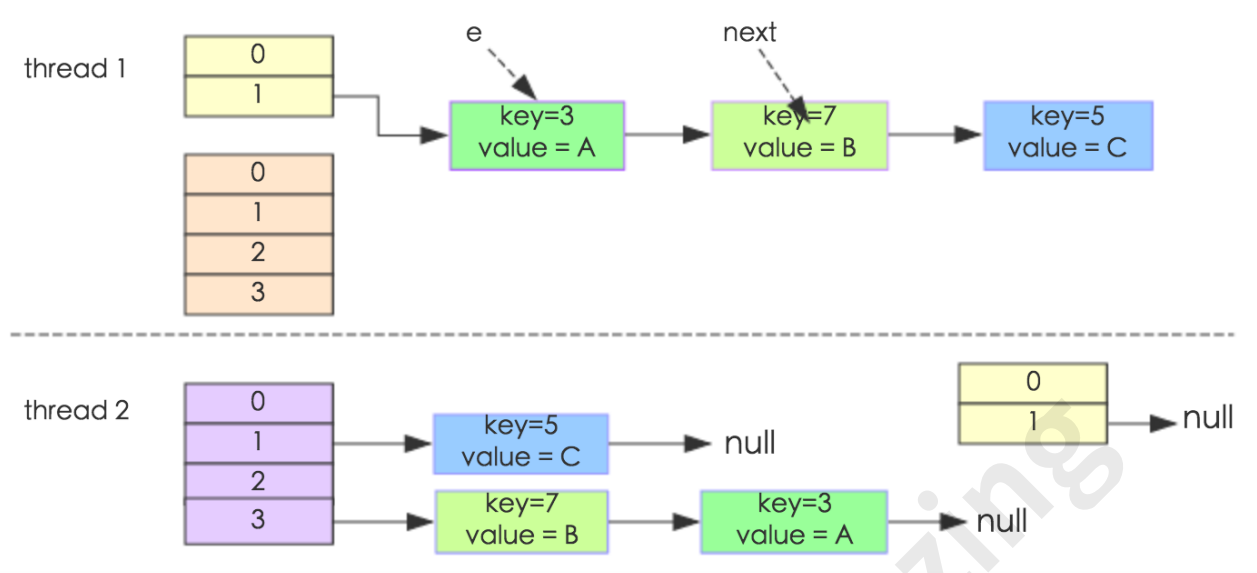
17.HashMap并发安全的问题

并发的多线程使用场景中使用HashMap可能造成死循环。代码例子如下(便于理解，仍然使用JDK1.7的环境):

```
public class HashMapInfiniteLoop {

    private static HashMap<Integer,String> map = new HashMap<Integer,String>
(2, 0.75f);
    public static void main(String[] args) {
        map.put(5,  "C");

        new Thread("Thread1") {
            public void run() {
                map.put(7, "B");
                System.out.println(map);
            };
        }.start();
        new Thread("Thread2") {
            public void run() {
                map.put(3, "A);
                System.out.println(map);
            };
        }.start();
    }
}
```

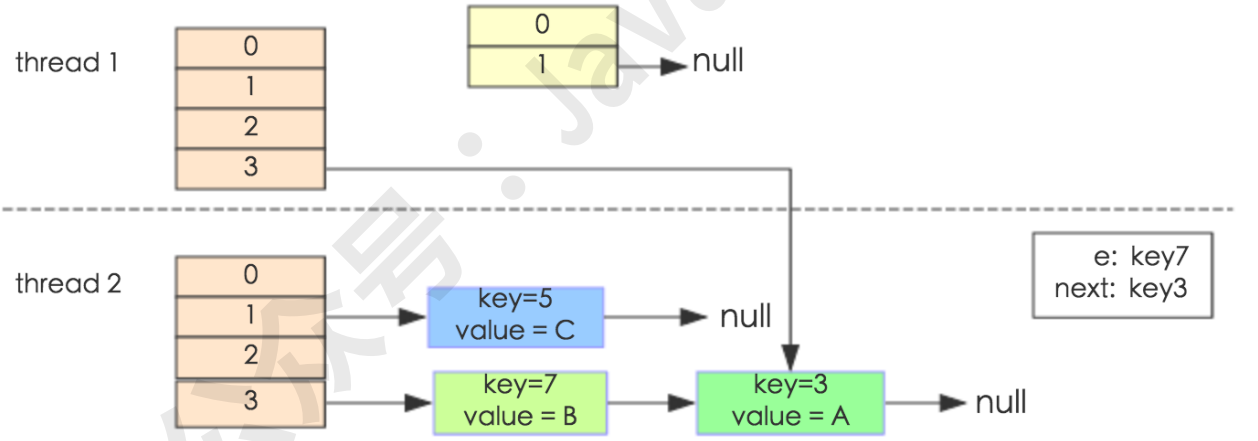
其中，map初始化为一个长度为2的数组，loadFactor=0.75，threshold=2*0.75=1，也就是说当put第二个key的时候，map就需要进行resize。

通过设置断点让线程1和线程2同时debug到transfer方法(3.3小节代码块)的首行。注意此时两个线程已经成功添加数据。放开thread1的断点至transfer方法的“Entry next = e.next;” 这一行；然后放开线程2的的断点，让线程2进行resize。结果如下图。

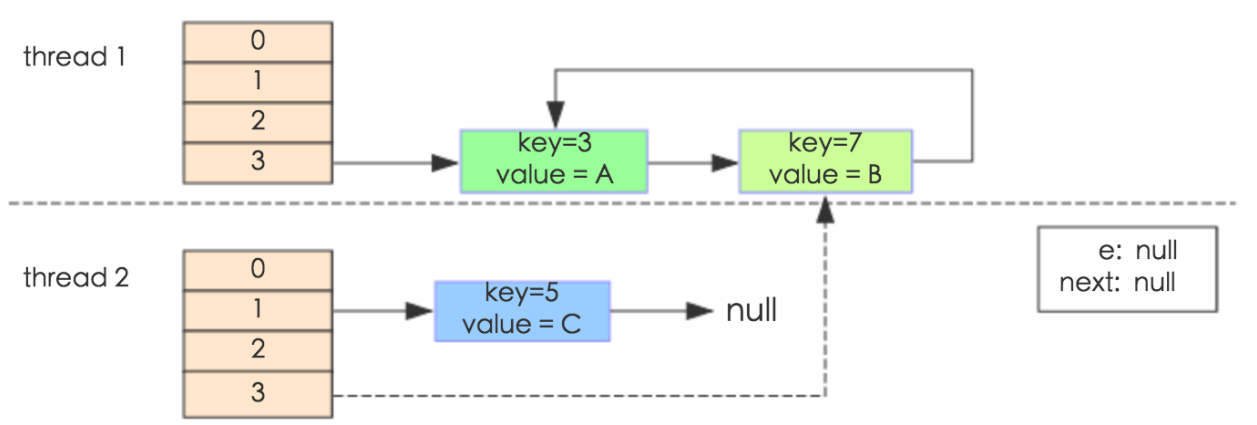


注意，Thread1的 e 指向了key(3)，而next指向了key(7)，其在线程二rehash后，指向了线程二重组后的链表。

线程一被调度回来执行，先是执行 newTalbe[i] = e， 然后是e = next，导致了e指向了key(7)，而下一次循环的next = e.next导致了next指向了key(3)。



e.next = newTable[i] 导致 key(3).next 指向了 key(7)。注意：此时的key(7).next 已经指向了key(3)， 环形链表就这样出现了。



于是，当我们用线程一调用map.get(11)时，悲剧就出现了——Infinite Loop。

18.JDK1.8与JDK1.7的性能对比

HashMap中，如果key经过hash算法得出的数组索引位置全部不相同，即Hash算法非常好，那样的话，getKey方法的时间复杂度就是O(1)，如果Hash算法技术的结果碰撞非常多，假如Hash算极其差，所有的Hash算法结果得出的索引位置一样，那样所有的键值对都集中到一个桶中，或者在一个链表中，或者在一个红黑树中，时间复杂度分别为O(n)和O(lgn)。 鉴于JDK1.8做了多方面的优化，总体性能优于JDK1.7，下面我们从两个方面用例子证明这一点。

Hash较均匀的情况

为了便于测试，我们先写一个类Key，如下：

```
class Key implements Comparable<Key> {

    private final int value;

    Key(int value) {
        this.value = value;
    }

    @Override
    public int compareTo(Key o) {
        return Integer.compare(this.value, o.value);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass())
            return false;
        Key key = (Key) o;
        return value == key.value;
    }

    @Override
    public int hashCode() {
        return value;
    }
}
```

这个类复写了equals方法，并且提供了相当好的hashCode函数，任何一个值的hashCode都不会相同，因为直接使用value当做hashcode。为了避免频繁的GC，我将不变的Key实例缓存了起来，而不是一遍一遍的创建它们。代码如下：

```
public class Keys {

    public static final int MAX_KEY = 10_000_000;
    private static final Key[] KEYS_CACHE = new Key[MAX_KEY];
```

```
    static {
        for (int i = 0; i < MAX_KEY; ++i) {
            KEYS_CACHE[i] = new Key(i);
        }
    }

    public static Key of(int value) {
        return KEYS_CACHE[value];
    }
}
```

现在开始我们的试验，测试需要做的仅仅是，创建不同size的HashMap（1、10、100、……10000000），屏蔽了扩容的情况，代码如下：

```
static void test(int mapSize) {

        HashMap<Key, Integer> map = new HashMap<Key,Integer>(mapSize);
        for (int i = 0; i < mapSize; ++i) {
            map.put(Keys.of(i), i);
        }

        long beginTime = System.nanoTime(); //获取纳秒
        for (int i = 0; i < mapSize; i++) {
            map.get(Keys.of(i));
        }
        long endTime = System.nanoTime();
        System.out.println(endTime - beginTime);
    }

    public static void main(String[] args) {
        for(int i=10;i<= 1000 0000;i*= 10){
            test(i);
        }
    }
```

在测试中会查找不同的值，然后度量花费的时间，为了计算getKey的平均时间，我们遍历所有的get方法，计算总的时间，除以key的数量，计算一个平均值，主要用来比较，绝对值可能会受很多环境因素的影响。结果如下：

map 的size大小	10	100	1000	10000	10 0000	100 0000	1000 0000
JDK1.7 get方法平均时间(ns)	900	540	570	285	55	6.9	8.1
JDK1.8 get方法平均时间(ns)	705	400	120	68	15	6.25	6.8

通过观测测试结果可知，JDK1.8的性能要高于JDK1.7 15%以上，在某些size的区域上，甚至高于100%。由于Hash算法较均匀，JDK1.8引入的红黑树效果不明显，下面我们看看Hash不均匀的的情况。

Hash极不均匀的情况

假设我们又一个非常差的Key，它们所有的实例都返回相同的hashCode值。这是使用HashMap最坏的情况。代码修改如下：

```
class Key implements Comparable<Key> {

    //…

    @Override
    public int hashCode() {
        return 1;
    }
}
```

仍然执行main方法，得出的结果如下表所示：

map 的size大小	10	100	1000	10000	10 0000	100 0000	1000 0
JDK1.7 get方法平均时间(ns)	2100	12960	3700	21000	17200	36000	---
JDK1.8 get方法平均时间(ns)	1960	3340	1470	720	190	230	220

从表中结果中可知，随着size的变大，JDK1.7的花费时间是增长的趋势，而JDK1.8是明显的降低趋势，并且呈现对数增长稳定。当一个链表太长的时候，HashMap会动态的将它替换成一个红黑树，这话的话会将时间复杂度从O(n)降为O(logn)。hash算法均匀和不均匀所花费的时间明显也不相同，这两种情况的相对比较，可以说明一个好的hash算法的重要性。

测试环境：处理器为2.2 GHz Intel Core i7，内存为16 GB 1600 MHz DDR3，SSD硬盘，使用默认的JVM参数，运行在64位的OS X 10.10.1上。

18.HashMap操作注意事项以及优化?

(1) 扩容是一个特别耗性能的操作，所以当程序员在使用HashMap的时候，估算map的大小，初始化的时候给一个大致的数值，避免map进行频繁的扩容。

(2) 负载因子是可以修改的，也可以大于1，但是建议不要轻易修改，除非情况非常特殊。

(3) HashMap是线程不安全的，不要在并发的环境中同时操作HashMap，建议使用ConcurrentHashMap。

(4) JDK1.8引入红黑树大程度优化了HashMap的性能。

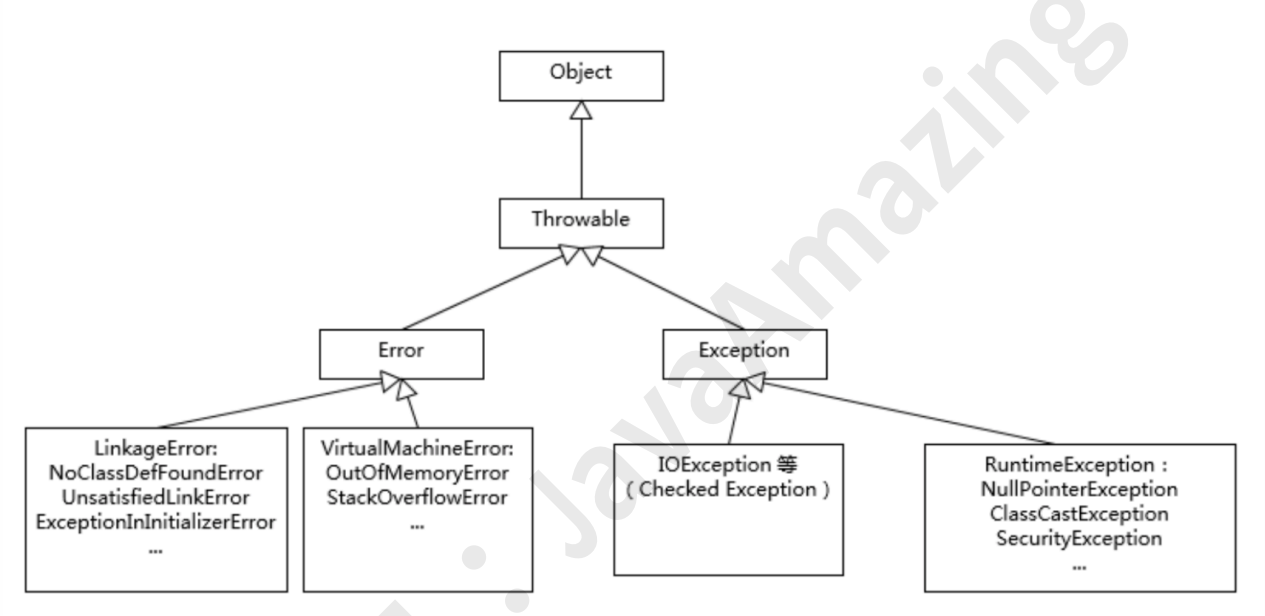
(5) 还没升级JDK1.8的，现在开始升级吧。HashMap的性能提升仅仅是JDK1.8的冰山一角。

参考：

- https://zhuanlan.zhihu.com/p/21673805
- https://www.jianshu.com/p/939b8a672070
- https://www.jianshu.com/p/c45b6d782e91
- https://blog.csdn.net/Mrs_chens/article/details/92761868
- https://blog.csdn.net/riemann_/article/details/87217229
- https://blog.csdn.net/qq_34626097/article/details/83053004

- https://blog.csdn.net/u010887744/article/details/50575735

异常&反射



1.error和exception有什么区别?

error表示系统级的错误，是java运行环境内部错误或者硬件问题，不能指望程序来处理这样的问题，除了退出运行外别无选择，它是Java虚拟机抛出的。

exception 表示程序需要捕捉、需要处理的异常，是由与程序设计的不完善而出现的问题，程序必须处理的问题。

2.说出5个常见的RuntimeException?

(1)Java.lang.NullPointerException 空指针异常;出现原因：调用了未经初始化的对象或者是不存在的对象。

(2)Java.lang.NumberFormatException 字符串转换为数字异常;出现原因：字符型数据中包含非数字型字符。

(3)Java.lang.IndexOutOfBoundsException 数组角标越界异常，常见于操作数组对象时发生。

(4)Java.lang.IllegalArgumentException 方法传递参数错误。

(5)Java.lang.ClassCastException 数据类型转换异常。

3.throw和throws的区别?

throw：

(1)throw 语句用在方法体内，表示抛出异常，由方法体内的语句处理。

(2)throw 是具体向外抛出异常的动作，所以它抛出的是一个异常实例，执行 throw 一定是抛出了某种异常。

throws：

(1)@throws 语句是用在方法声明后面，表示如果抛出异常，由该方法的调用者来进行异常的处理。

(2)throws 主要是声明这个方法会抛出某种类型的异常，让它的使用者要知道需要捕获的异常的类型。

(3)throws 表示出现异常的一种可能性，并不一定会发生这种异常。

4.Java中异常分类

按照异常处理时机:

编译时异常(受控异常(CheckedException))和运行时异常(非受控异常(UnCheckedException))

5.如何自定义异常

继承Exception是检查性异常，继承RuntimeException是非检查性异常，一般要复写两个构造方法，用throw抛出新异常

如果同时有很多异常抛出，那可能就是异常链，就是一个异常引发另一个异常，另一个异常引发更多异常，一般我们会找它的原始异常来解决问题，一般会在开头或结尾，异常可通过initCause串起来，可以通过自定义异常

6.Java中异常处理

首先处理异常主要有两种方式:一种try catch，一种是throws。

1. try catch:

- try{} 中放入可能发生异常的代码。catch{}中放入对捕获到异常之后的处理。

2.throw throws：

- throw是语句抛出异常，出现于函数内部，用来抛出一个具体异常实例，throw被执行后面的语句不起作用，直接转入异常处理阶段。
- throws是函数方法抛出异常，一般写在方法的头部，抛出异常，给方法的调用者进行解决。

7.什么是Java反射机制?

Java的反射（reflection）机制是指在程序的运行状态中，可以构造任意一个类的对象，可以了解任意一个对象所属的类，可以了解任意一个类的成员变量和方法，可以调用任意一个对象的属性和方法。这种动态获取程序信息以及动态调用对象的功能称为Java语言的反射机制。反射被视为动态语言的关键。

8.举例什么地方用到反射机制?

1. JDBC中，利用反射动态加载了数据库驱动程序。
2. Web服务器中利用反射调用了Sevlet的服务方法。
3. Eclispe等开发工具利用反射动态刨析对象的类型与结构，动态提示对象的属性和方法。
4. 很多框架都用到反射机制，注入属性，调用方法，如Spring。

9.java反射机制的作用

- 在运行时判定任意一个对象所属的类
- 在运行时构造任意一个类的对象；
- 在运行时判定任意一个类所具有的成员变量和方法；
- 在运行时调用任意一个对象的方法；
- 生成动态代理；

10.Java反射机制类

```
java.lang.Class; //类
java.lang.reflect.Constructor;//构造方法
java.lang.reflect.Field; //类的成员变量
java.lang.reflect.Method;//类的方法
java.lang.reflect.Modifier;//访问权限
```

11.反射机制优缺点?

优点：运行期类型的判断，动态加载类，提高代码灵活度。

缺点：性能瓶颈：反射相当于一系列解释操作，通知 JVM 要做的事情，性能比直接的java代码要慢很多。

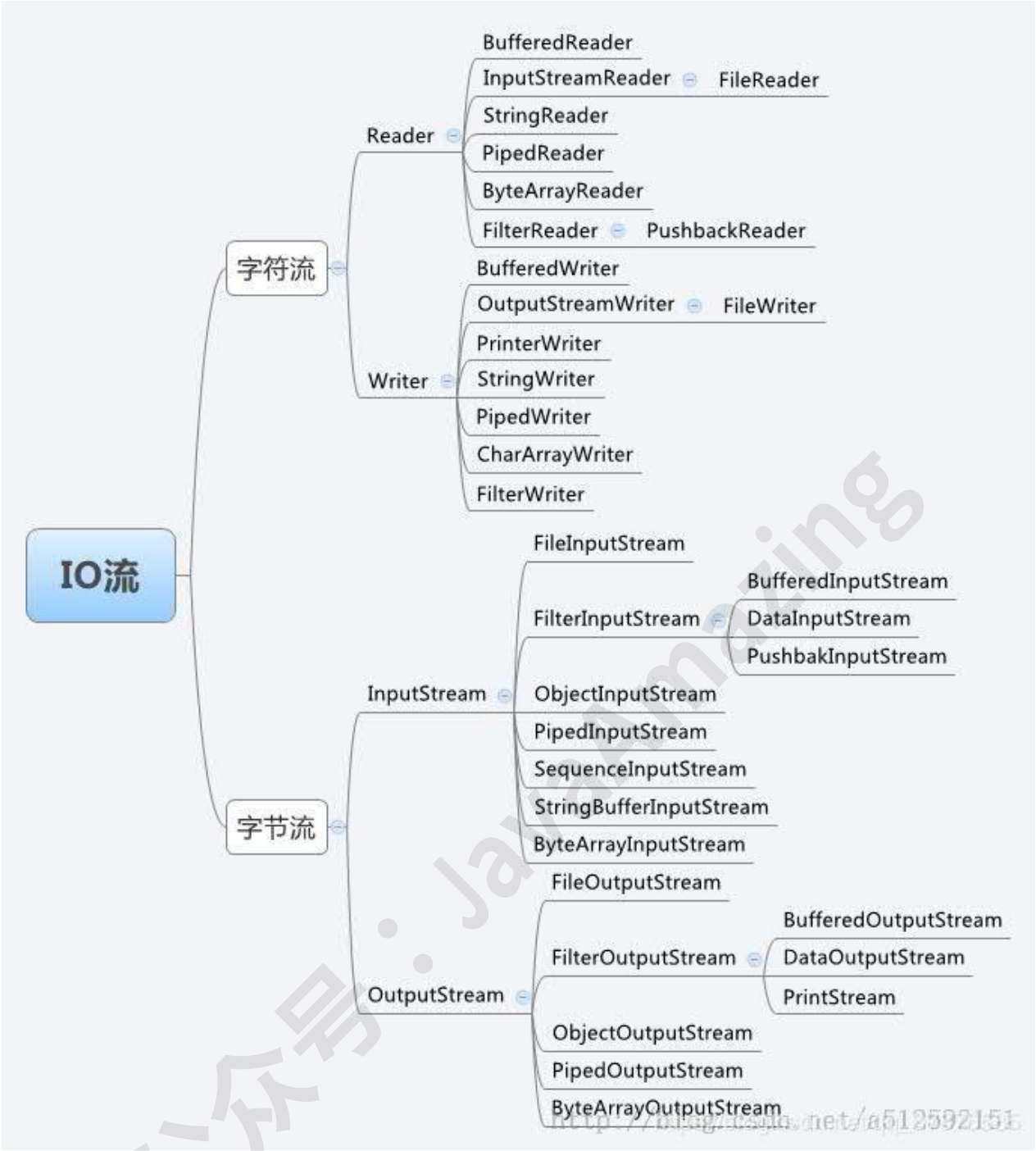
12.利用反射创建对象?

1.通过一个全限类名创建一个对象
Class.forName(“全限类名”); 例如：com.mysql.jdbc.Driver Driver类已经被加载到 jvm中，并且完成了类的初始化工作就行了
类名.class; 获取Class<？>clz 对象
对象.getClass();
2.获取构造器对象，通过构造器new出一个对象
Clazz.getConstructor([String.class]);
Con.newInstance([参数]);
3.通过class对象创建一个实例对象（就相当与new类名（）无参构造器）
Cls.newInstance();

参考：

- https://blog.csdn.net/qq_37875585/article/details/89340495
- https://www.cnblogs.com/whoislcj/p/6038511.html

IO&NIO



1.什么是IO流?

它是一种数据的流从源头流到目的地。比如文件拷贝，输入流和输出流都包括了。输入流从文件中读取数据存储到进程(process)中，输出流从进程中读取数据然后写入到目标文件。

2.java中有几种类型的流?

按照单位大小：字符流、字节流。按照流的方向：输出流、输入流。

3.字节流和字符流哪个好？怎么选择?

1. 缓大多数情况下使用字节流会更好，因为字节流是字符流的包装，而大多数时候 IO 操作都是直接操作磁盘文件，所以这些流在传输时都是以字节的方式进行的（图片等都是按字节存储的）
2. 如果对于操作需要通过 IO 在内存中频繁处理字符串的情况使用字符流会好些，因为字符流具备缓冲

区，提高了性能

4.读取数据量大的文件时，速度会很慢，如何选择流?

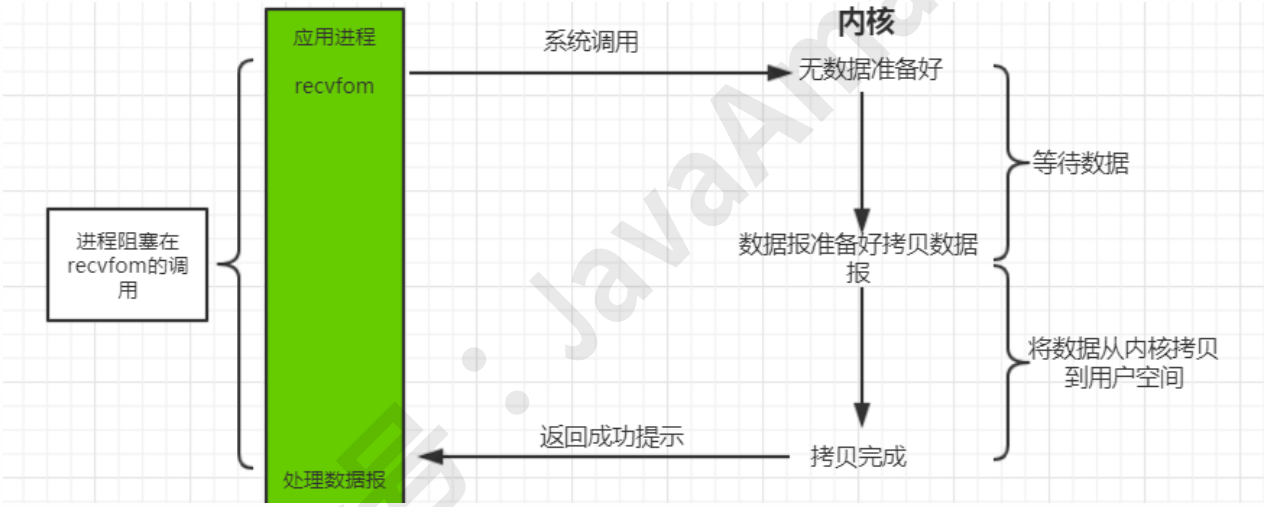
字节流时，选择BufferedInputStream和BufferedOutputStream。
字符流时，选择BufferedReader 和 BufferedWriter

5. IO模型有几种?

阻塞IO、非阻塞IO、多路复用IO、信号驱动IO以及异步IO。

6.阻塞IO（blocking IO）

应用程序调用一个IO函数，导致应用程序阻塞，如果数据已经准备好，从内核拷贝到用户空间，否则一直等待下去。一个典型的读操作流程大致如下图，当用户进程调用recvfrom这个系统调用时，kernel就开始了IO的第一个阶段：准备数据，就是数据被拷贝到内核缓冲区中的一个过程（很多网络IO数据不会那么快到达，如没收一个完整的UDP包），等数据到操作系统内核缓冲区了，就到了第二阶段：将数据从内核缓冲区拷贝到用户内存，然后kernel返回结果，用户进程才会解除block状态，重新运行起来。
blocking IO的特点就是在IO执行的两个阶段用户进程都会block住；

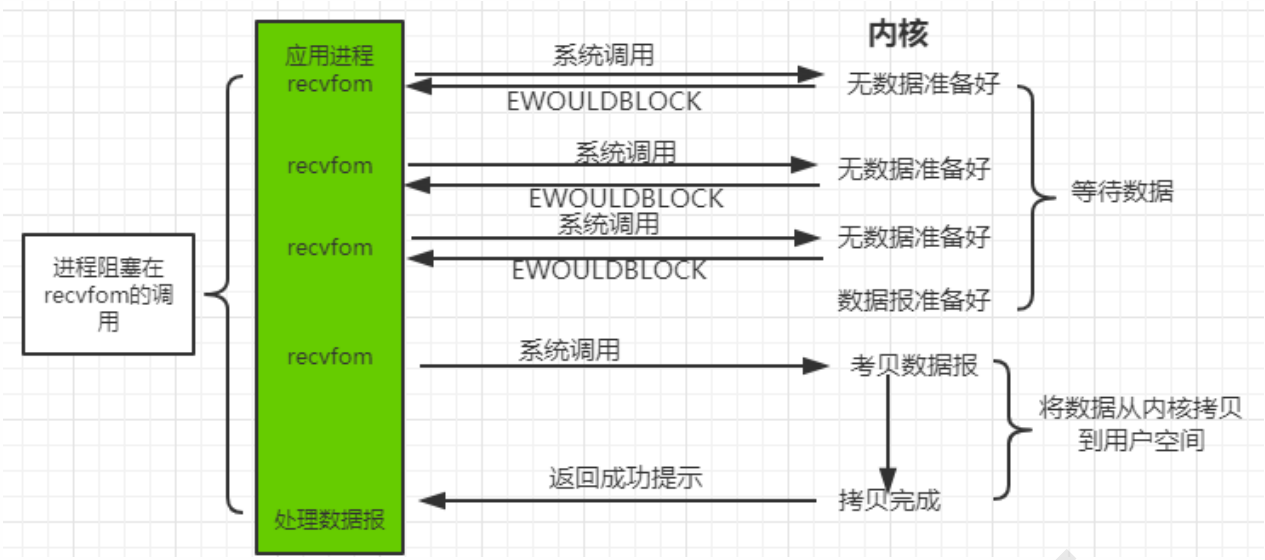


7.非阻塞I/O（nonblocking IO）

非阻塞I/O模型，我们把一个套接口设置为非阻塞就是告诉内核，当所请求的I/O操作无法完成时，不要将进程睡眠，而是返回一个错误。这样我们的I/O操作函数将不断的测试数据是否已经准备好，如果没有准备好，继续测试，直到数据准备好为止。在这个不断测试的过程中，**会大量的占用CPU的时间**。

当用户进程发出read操作时，如果kernel中数据还没准备好，那么并不会block用户进程，而是立即返回error，用户进程判断结果是error，就知道数据还没准备好，用户可以再次发read，直到kernel中数据准备好，并且用户再一次发read操作，产生system call，那么kernel 马上将数据拷贝到用户内存，然后返回；所以**nonblocking IO的特点是用户进程需要不断的主动询问kernel数据好了没有**。

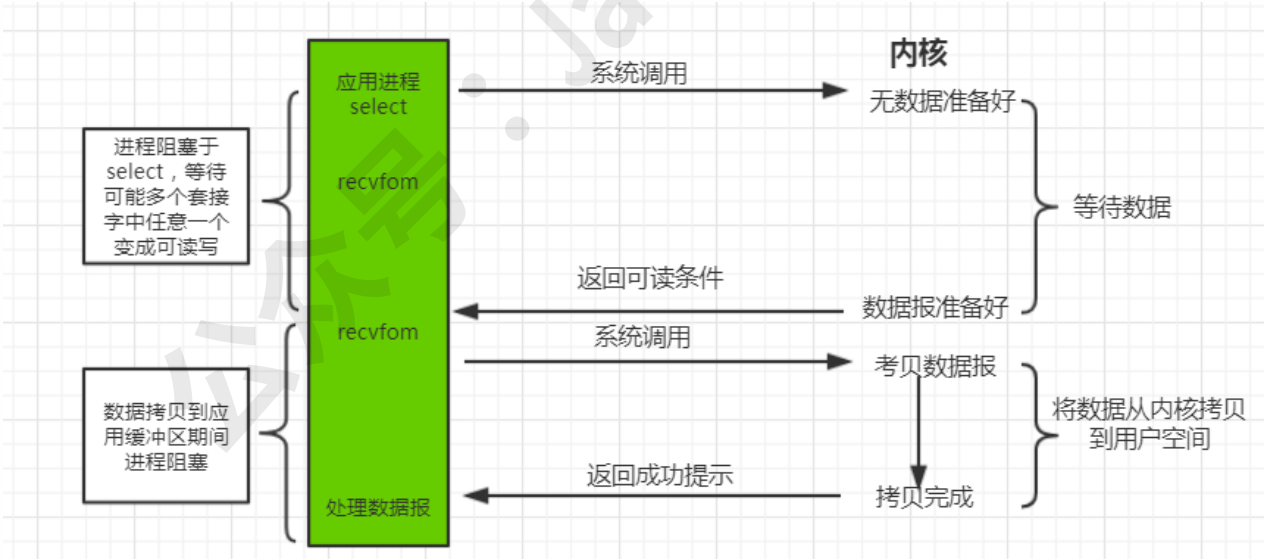
阻塞IO一个线程只能处理一个IO流事件，要想同时处理多个IO流事件要么多线程要么多进程，这样做效率显然不会高，而非阻塞IO可以一个线程处理多个流事件，只要不停地询所有流事件即可，当然这个方式也不好，当大多数流没数据时，也是会大量浪费CPU资源；为了避免CPU空转，引进代理(select和poll，两种方式相差不大)，代理可以观察多个流I/O事件，空闲时会把当前线程阻塞掉，当有一个或多个I/O事件时，就从阻塞态醒过来，把所有IO流都轮询一遍，于是没有IO事件我们的程序就阻塞在select方法处，即便这样依然存在问题，我们从select出只是知道有IO事件发生，却不知道是哪几个流，还是只能轮询所有流，**epoll**这样的代理就可以把哪个流发生怎样的IO事件通知我们；



8.I/O多路复用模型(IO multiplexing)

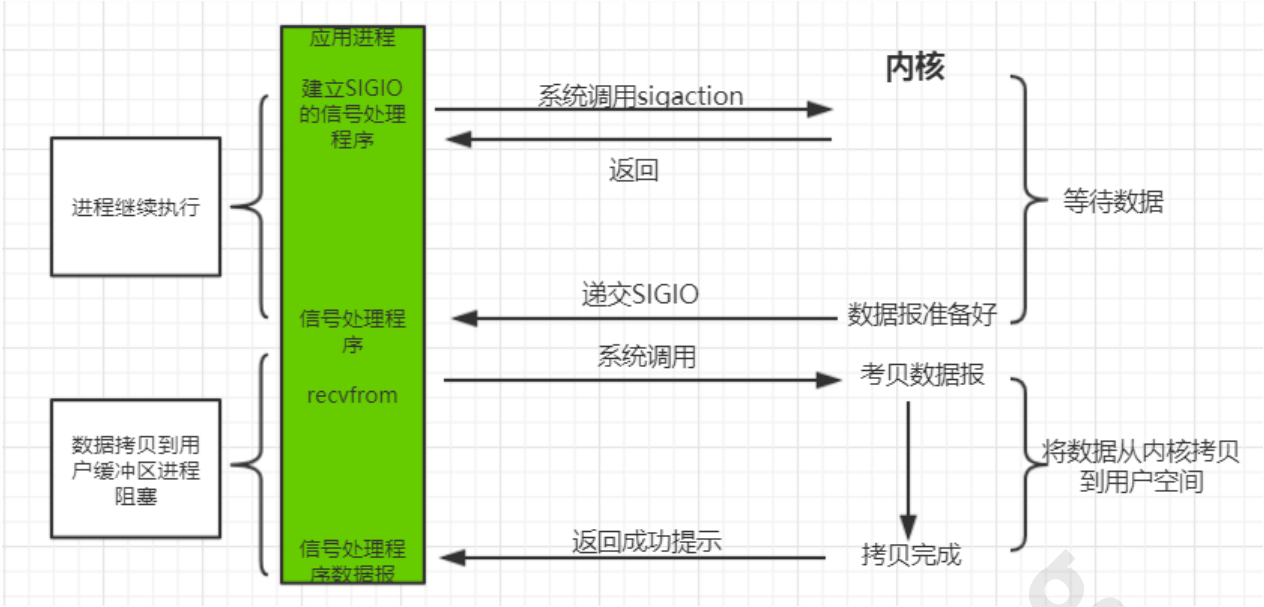
I/O多路复用就在于单个进程可以同时处理多个网络连接IO,基本原理就是select，poll，epoll这些个函数会不断轮询所负责的所有socket，当某个socket有数据到达了，就通知用户进程，这三个functon会阻塞进程，但和IO阻塞不同，这些函数可以同时阻塞多个IO操作，而且可以同时对多个读操作，写操作IO进行检验，直到有数据到达，才真正调用IO操作函数，调用过程如下图；**所以IO多路复用的特点是通过一种机制一个进程能同时等待多个文件描述符，而这些文件描述符(套接字描述符)其中任意一个进入就绪状态，select函数就可以返回。**

IO多路复用的优势在于并发数比较高的IO操作情况，可以同时处理多个连接，和bloking IO一样socket是被阻塞的，只不过在多路复用中socket是被select阻塞，而在阻塞IO中是被socket IO给阻塞。



9.信号驱动I/O模型

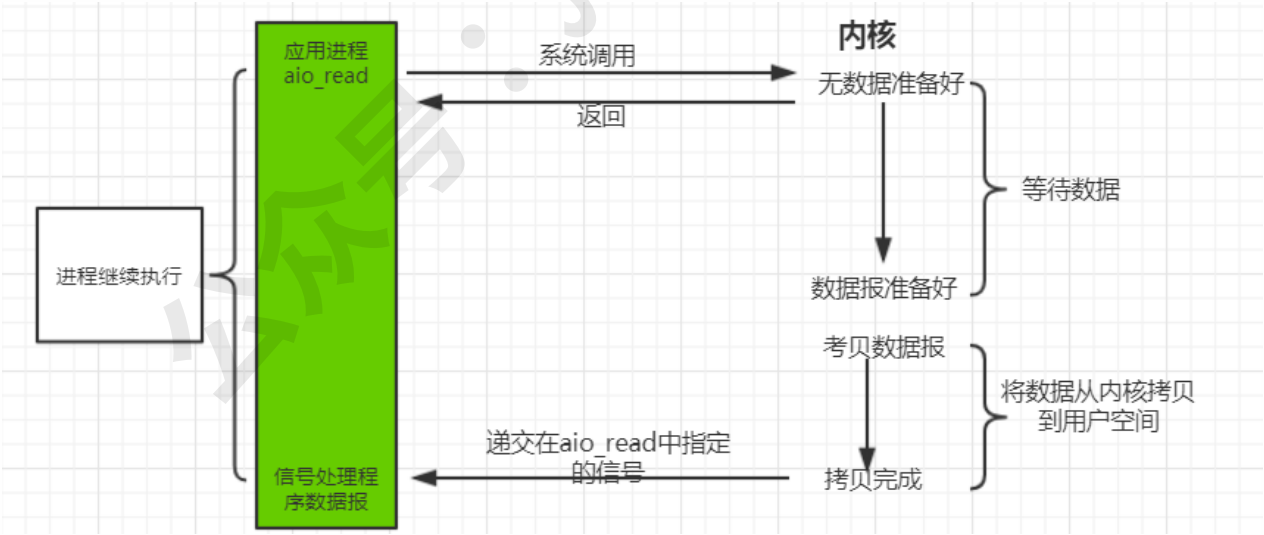
可以用信号，让内核在描述符就绪时发送SIGIO信号通知我们，通过sigaction系统调用安装一个信号处理函数。该系统调用将立即返回，我们的进程继续工作，也就是说它没有被阻塞。当数据报准备好读取时，内核就为该进程产生一个SIGIO信号。我们随后既可以在信号处理函数中调用recvfrom读取数据报，并通知主循环数据已经准备好待处理。**特点：等待数据报到达期间进程不被阻塞。主循环可以继续执行，只要等待来自信号处理函数的通知：既可以是数据已准备好被处理，也可以是数据报已准备好被读取**



10.异步 I/O(asynchronous IO)

异步IO告知内核启动某个操作，并让内核在整个操作(包括将内核数据复制到我们自己的缓冲区)完成后通知我们，调用aio_read（Posix异步I/O函数以aio*或lio*开头）函数，给内核传递描述字、缓冲区指针、缓冲区大小（与read相同的3个参数）、文件偏移以及通知的方式，然后系统立即返回。我们的进程不阻塞于等待I/0操作的完成。当内核将数据拷贝到缓冲区后，再通知应用程序。

用户进程发起read操作之后，立刻就可以开始去做其它的事。而另一方面，从kernel的角度，当它受到一个asynchronous read之后，首先它会立刻返回，所以不会对用户进程产生任何block。然后，kernel会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，kernel会给用户进程发送一个signal，告诉它read操作完成了



11.NIO与IO的区别?

NIO即New IO，这个库是在JDK1.4中才引入的。NIO和IO有相同的作用和目的，但实现方式不同，NIO主要用到的是块，所以NIO的效率要比IO高很多。在Java API中提供了两套NIO，一套是针对标准输入输出NIO，另一套就是网络编程NIO。

IO	NIO
面向流	面向缓冲
阻塞IO	非阻塞IO
无	选择器

12.NIO和IO适用场景

NIO是为弥补传统IO的不足而诞生的，但是尺有所短寸有所长，**NIO也有缺点，因为NIO是面向缓冲区的操作，每一次的数据处理都是对缓冲区进行的，那么就会有一个问题，在数据处理之前必须要判断缓冲区的数据是否完整或者已经读取完毕，如果没有，假设数据只读取了一部分，那么对不完整的数据处理没有任何意义。**所以每次数据处理之前都要检测缓冲区数据。

那么NIO和IO各适用的场景是什么呢?

如果需要管理同时打开的成千上万个连接，这些**连接每次只是发送少量的数据**，例如聊天服务器，这时候用NIO处理数据可能是个很好的选择。

而如果**只有少量的连接**，而这些连接每次要发送大量的数据，这时候传统的IO更合适。使用哪种处理数据，需要在数据的响应等待时间和检查缓冲区数据的时间上作比较来权衡选择。

13.NIO核心组件

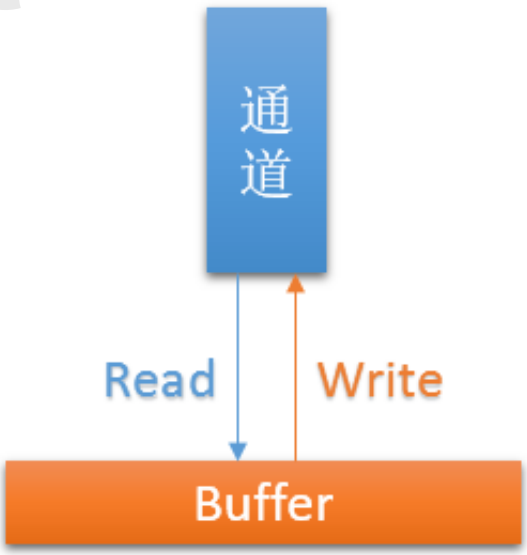
channel、buffer、selector

14.什么是channel

一个Channel（通道）代表和某一实体的连接，这个实体可以是文件、网络套接字等。也就是说，通道是Java NIO提供的一座桥梁，用于我们的程序和操作系统底层I/O服务进行交互。

通道是一种很基本很抽象的描述，和不同的I/O服务交互，执行不同的I/O操作，实现不一样，因此具体的有FileChannel、SocketChannel等。

通道使用起来跟Stream比较像，可以读取数据到Buffer中，也可以把Buffer中的数据写入通道。



当然，也有区别，主要体现在如下两点：

- 一个通道，既可以读又可以写，而一个Stream是单向的（所以分 InputStream 和 OutputStream）
- 通道有非阻塞I/O模式

15.Java NIO中最常用的通道实现?

- FileChannel：读写文件
- DatagramChannel: UDP协议网络通信
- SocketChannel：TCP协议网络通信
- ServerSocketChannel：监听TCP连接

16.Buffer是什么?

NIO中所使用的缓冲区不是一个简单的byte数组，而是封装过的Buffer类，通过它提供的API，我们可以灵活的操纵数据。

与Java基本类型相对应，NIO提供了多种 Buffer 类型，如ByteBuffer、CharBuffer、IntBuffer等，区别就是读写缓冲区时的单位长度不一样（以对应类型的变量为单位进行读写）。

17.核心Buffer实现有哪些?

核心的buffer实现有这些：ByteBuffer、CharBuffer、DoubleBuffer、FloatBuffer、IntBuffer、LongBuffer、ShortBuffer，涵盖了所有的基本数据类型（4类8种，除了Boolean）。也有其他的buffer如MappedByteBuffer。

18.buffer读写数据基本操作

1）、将数据写入buffer
2）、调用buffer.flip()
3）、将数据从buffer中读取出来
4）、调用buffer.clear()或者buffer.compact()

在写buffer的时候，buffer会跟踪写入了多少数据，需要读buffer的时候，需要调用flip()来将buffer从写模式切换成读模式，读模式中只能读取写入的数据，而非整个buffer。

当数据都读完了，你需要清空buffer以供下次使用，可以有2种方法来操作：调用clear() 或者 调用compact()。

区别：clear方法清空整个buffer，compact方法只清除你已经读取的数据，未读取的数据会被移到buffer的开头，此时写入数据会从当前数据的末尾开始。

```
// 创建一个容量为48的ByteBuffer
ByteBuffer buf = ByteBuffer.allocate(48);
// 从channel中读（取数据然后写）入buffer
int bytesRead = inChannel.read(buf);
// 下面是读取buffer
while (bytesRead != -1) {
    buf.flip();  // 转换buffer为读模式
    System.out.print((char) buf.get()); // 一次读取一个byte
    buf.clear();  //清空buffer准备下一次写入
}
```

19.Selector是什么?

Selector（选择器）是一个特殊的组件，用于采集各个通道的状态（或者说事件）。我们先将通道注册到选择器，并设置好关心的事件，然后就可以通过调用select()方法，静静地等待事件发生。

20.通道可以监听那几个事件?

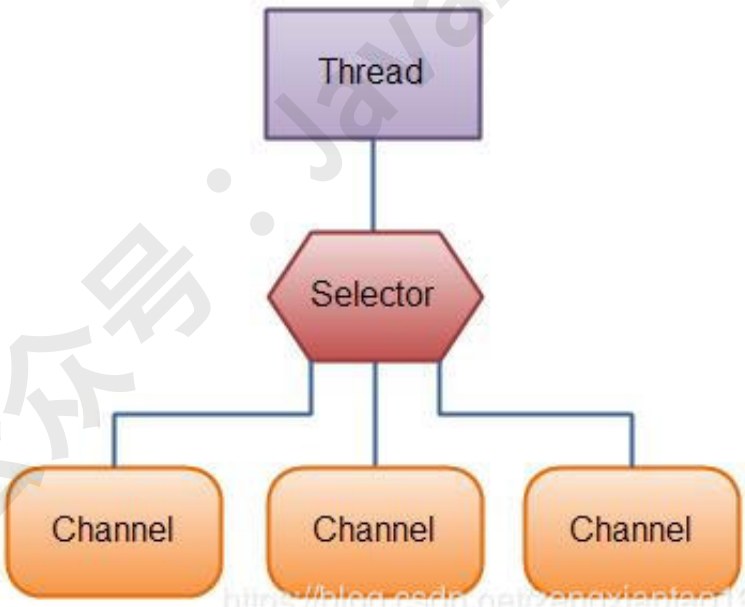
通道有如下4个事件可供我们监听：

- Accept：有可以接受的连接
- Connect：连接成功
- Read：有数据可读
- Write：可以写入数据了

21.为什么要用Selector?

如果用阻塞I/O，需要多线程（浪费内存），如果用非阻塞I/O，需要不断重试（耗费CPU）。Selector的出现解决了这尴尬的问题，非阻塞模式下，通过Selector，我们的线程只为已就绪的通道工作，不用盲目的重试了。比如，当所有通道都没有数据到达时，也就没有Read事件发生，我们的线程会在select()方法处被挂起，从而让出了CPU资源。

22.Selector处理多Channel图文说明



要使用一个Selector，你要先注册这个Selector的Channels。然后你调用Selector的select()方法。这个方法会阻塞，直到它注册的Channels当中有一个准备好了的事件发生了。当select()方法返回的时候，线程可以处理这些事件，如新的连接的到来，数据收到了等。

参考:

- https://www.cnblogs.com/sharing-java/p/10791802.html
- https://blog.csdn.net/zengxiantao1994/article/details/88094910

- https://www.cnblogs.com/xueSpring/p/9513266.html

多线程

1.什么是进程?

进程是系统中正在运行的一个程序，程序一旦运行就是进程。

进程可以看成程序执行的一个实例。进程是系统资源分配的独立实体，每个进程都拥有独立的地址空间。一个进程无法访问另一个进程的变量和数据结构，如果想让一个进程访问另一个进程的资源，需要使用进程间通信，比如管道，文件，套接字等。

2.什么是线程?

是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

3.线程的实现方式?

1.继承Thread类

2.实现Runnable接口

3.使用Callable和Future

4.Thread 类中的start() 和 run() 方法有什么区别?

1.start（）方法来启动线程，真正实现了多线程运行。这时无需等待run方法体代码执行完毕，可以直接继续执行下面的代码；通过调用Thread类的start()方法来启动一个线程， 这时此线程是处于就绪状态，并没有运行。 然后通过此Thread类调用方法run()来完成其运行操作的， 这里方法run()称为线程体，它包含了要执行的这个线程的内容， Run方法运行结束， 此线程终止。然后CPU再调度其它线程。
2.run（）方法当作普通方法的方式调用。程序还是要顺序执行，要等待run方法体执行完毕后，才可继续执行下面的代码； 程序中只有主线程——这一个线程， 其程序执行路径还是只有一条， 这样就没有达到写线程的目的。

5.线程NEW状态

new创建一个Thread对象时，并没处于执行状态，因为没有调用start方法启动改线程，那么此时的状态就是新建状态。

6.线程RUNNABLE状态

线程对象通过start方法进入runnable状态，启动的线程不一定会立即得到执行，线程的运行与否要看cpu的调度，我们把这个中间状态叫可执行状态（RUNNABLE)。

7.线程的RUNNING状态

一旦cpu通过轮询货其他方式从任务可以执行队列中选中了线程，此时它才能真正的执行自己的逻辑代码。

8.线程的BLOCKED状态

线程正在等待获取锁。

- 进入BLOCKED状态，比如调用了sleep,或者wait方法
- 进行某个阻塞的io操作，比如因网络数据的读写进入BLOCKED状态
- 获取某个锁资源，从而加入到该锁的阻塞队列中而进入BLOCKED状态

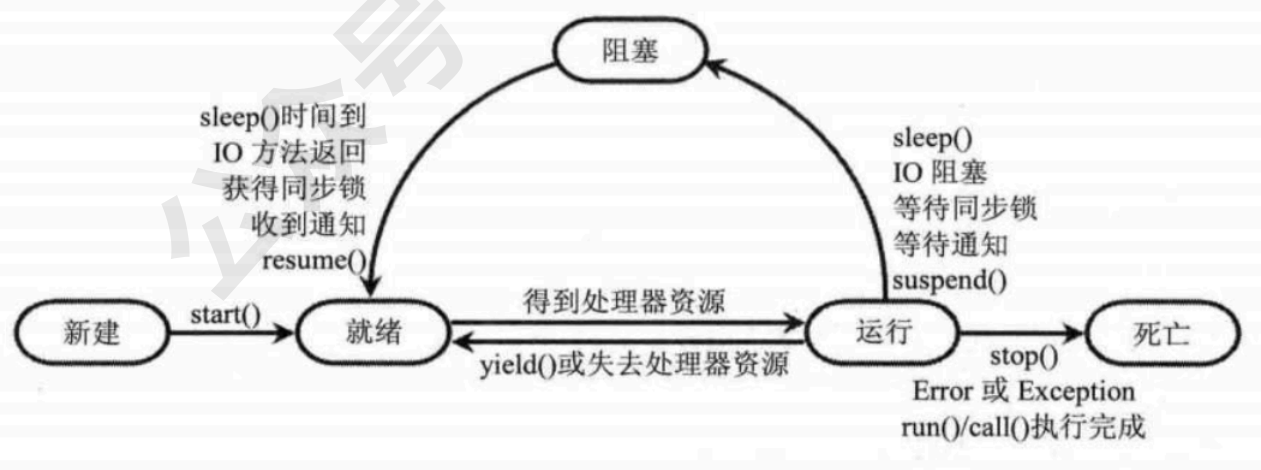
9.线程的TERMINATED状态

TERMINATED是一个线程的最终状态，在该状态下线程不会再切换到其他任何状态了，代表整个生命周期都结束了。

下面几种情况会进入TERMINATED状态:

- 线程运行正常结束，结束生命周期
- 线程运行出错意外结束
- JVM Crash 导致所有的线程都结束

10.线程状态转化图



11.i————————与System.out.println()的异常

示例代码:

```
public class XkThread extends Thread {

    private int i = 5;
```

```
    @Override
    public void run() {
       System.out.println("i=" + (i------) + " threadName=" +
Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        XkThread xk = new XkThread();
        Thread t1 = new Thread(xk);
        Thread t2 = new Thread(xk);
        Thread t3 = new Thread(xk);
        Thread t4 = new Thread(xk);
        Thread t5 = new Thread(xk);

        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();
    }
}
```

结果:

```
i=5 threadName=Thread-1
i=2 threadName=Thread-5
i=5 threadName=Thread-2
i=4 threadName=Thread-3
i=3 threadName=Thread-4
```

虽然println()方法在内部是同步的，但i----------的操作却是在进入println()之前发生的，所以有发生非线程安全的概率。

println()源码:

```
    public void println(String x) {
        synchronized (this) {
            print(x);
            newLine();
        }
    }
```

12.如何知道代码段被哪个线程调用?

```
    System.out.println(Thread.currentThread().getName());
```

13.线程活动状态?

```
public class XKThread extends Thread {

    @Override
    public void run() {
        System.out.println("run run run is "  + this.isAlive() );
    }

    public static void main(String[] args) {
        XKThread xk = new XKThread();
        System.out.println("begin ——— " + xk.isAlive());
        xk.start();
        System.out.println("end ——— " + xk.isAlive());

    }
}
```

14.sleep()方法

方法sleep()的作用是在指定的毫秒数内让当前的“正在执行的线程”休眠（暂停执行）。

15.如何优雅的设置睡眠时间?

jdk1.5 后，引入了一个枚举TimeUnit,对sleep方法提供了很好的封装。

比如要表达2小时22分55秒899毫秒。

```
Thread.sleep(8575899L);
TimeUnit.HOURS.sleep(3);
TimeUnit.MINUTES.sleep(22);
TimeUnit.SECONDS.sleep(55);
TimeUnit.MILLISECONDS.sleep(899);
```

可以看到表达的含义更清晰，更优雅。

16.停止线程

run方法执行完成，自然终止。

stop()方法，suspend()以及resume()都是过期作废方法，使用它们结果不可预期。

大多数停止一个线程的操作使用Thread.interrupt()等于说给线程打一个停止的标记, 此方法不回去终止一个正在运行的线程，需要加入一个判断才能可以完成线程的停止。

17.interrupted 和 isInterrupted

interrupted： 判断当前线程是否已经中断,会清除状态。

isInterrupted： 判断线程是否已经中断，不会清除状态。

18.yield

放弃当前cpu资源，将它让给其他的任务占用cpu执行时间。但放弃的时间不确定，有可能刚刚放弃，马上又获得cpu时间片。

测试代码:(cpu独占时间片)

```
public class XKThread extends Thread {

    @Override
    public void run() {
        long beginTime = System.currentTimeMillis();
        int count = 0;
        for (int i = 0; i < 50000000; i++) {
            count = count + (i + 1);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("用时 = " + (endTime - beginTime) + " 毫秒! ");
    }

    public static void main(String[] args) {
        XKThread xkThread = new XKThread();
        xkThread.start();
    }

}
```

结果:

```
用时 = 20 毫秒!
```

加入yield，再来测试。(cpu让给其他资源导致速度变慢)

```
public class XKThread extends Thread {

    @Override
    public void run() {
        long beginTime = System.currentTimeMillis();
        int count = 0;
        for (int i = 0; i < 50000000; i++) {
            Thread.yield();
            count = count + (i + 1);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("用时 = " + (endTime - beginTime) + " 毫秒! ");
    }

    public static void main(String[] args) {
        XKThread xkThread = new XKThread();
        xkThread.start();
    }

}
```

结果:

```
用时 = 38424 毫秒!
```

19.线程的优先级

在操作系统中，线程可以划分优先级，优先级较高的线程得到cpu资源比较多，也就是cpu有限执行优先级较高的线程对象中的任务，但是不能保证一定优先级高，就先执行。

Java的优先级分为1～10个等级，数字越大优先级越高，默认优先级大小为5。超出范围则抛出：java.lang.IllegalArgumentException。

20.优先级继承特性

线程的优先级具有继承性，比如a线程启动b线程，b线程与a优先级是一样的。

21.谁跑的更快?

设置优先级高低两个线程，累加数字，看谁跑的快，上代码。

```
public class Run extends Thread{

    public static void main(String[] args) {
        try {
            ThreadLow low = new ThreadLow();
            low.setPriority(2);
            low.start();

            ThreadHigh high = new ThreadHigh();
            high.setPriority(8);
            high.start();

            Thread.sleep(2000);
            low.stop();
            high.stop();
            System.out.println("low  = " + low.getCount());
            System.out.println("high = " + high.getCount());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }

}

class ThreadHigh extends Thread {
    private int count = 0;

    public int getCount() {
```

```
            return count;
        }


        @Override
        public void run() {
            while (true) {
                count++;
            }
        }
    }

    class ThreadLow extends Thread {
        private int count = 0;

        public int getCount() {
            return count;
        }


        @Override
        public void run() {
            while (true) {
                count++;
            }
        }
    }
```

结果:

```
low  = 1193854568
high = 1204372373
```

22.线程种类

Java线程有两种，一种是用户线程，一种是守护线程。

23.守护线程的特点

守护线程是一个比较特殊的线程，主要被用做程序中后台调度以及支持性工作。当Java虚拟机中不存在非守护线程时，守护线程才会随着JVM一同结束工作。

24.Java中典型的守护线程

GC（垃圾回收器）

25.如何设置守护线程

Thread.setDaemon(true)

PS:Daemon属性需要再启动线程之前设置，不能再启动后设置。

25.Java虚拟机退出时Daemon线程中的finally块一定会执行?

Java虚拟机退出时Daemon线程中的finally块并不一定会执行。

代码示例:

```
public class XKDaemon {
    public static void main(String[] args) {
        Thread thread = new Thread(new DaemonRunner(),"xkDaemonRunner");
        thread.setDaemon(true);
        thread.start();

    }

    static class DaemonRunner implements Runnable {

        @Override
        public void run() {
            try {
                SleepUtils.sleep(10);
            } finally {
                System.out.println("Java小咖秀 daemonThread finally run …");
            }

        }
    }
}
```

结果:

```

```

没有任何的输出，说明没有执行finally。

26.设置线程上下文类加载器

获取线程上下文类加载器

```
public ClassLoader getContextClassLoader()
```

设置线程类加载器（可以打破Java类加载器的父类委托机制）

```
public void setContextClassLoader(ClassLoader cl)
```

27.join

join是指把指定的线程加入到当前线程，比如join某个线程a,会让当前线程b进入等待,直到a的生命周期结束，此期间b线程是处于blocked状态。

28.什么是synchronized?

synchronized关键字可以时间一个简单的策略来防止线程干扰和内存一致性错误，如果一个对象是对多个线程可见的，那么对该对想的所有读写都将通过同步的方式来进行。

29.synchronized包括哪两个jvm重要的指令?

monitor enter 和 monitor exit

30.synchronized关键字用法?

可以用于对代码块或方法的修饰

31.synchronized锁的是什么?

普通同步方法 ————————> 锁的是当前实力对象。

静态同步方法————————> 锁的是当前类的Class对象。

同步方法快 ————————> 锁的是synchonized括号里配置的对象。

32.Java对象头

synchronized用的锁是存在Java对象头里的。对象如果是数组类型，虚拟机用3个字宽(Word)存储对象头，如果对象是非数组类型，用2字宽存储对象头。

Tips:32位虚拟机中一个字宽等于4字节。

33.Java对象头长度

长　　度	内　　容	说　　明
32/64bit	Mark Word	存储对象的 hashCode 或锁信息等
32/64bit	Class Metadata Address	存储到对象类型数据的指针
32/32bit	Array length	数组的长度（如果当前对象是数组）

34.Java对象头的存储结构

32位JVM的Mark Word 默认存储结构

锁状态	25bit	4bit	1bit 是否是偏向锁	2bit 锁标志位
无锁状态	对象的 hashCode	对象分代年龄	0	01

35.Mark Word的状态变化

Mark Word 存储的数据会随着锁标志为的变化而变化。

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量（重量级锁）的指针				10
GC 标记	空				11
偏向锁	线程 ID	Epoch	对象分代年龄	1	01

64位虚拟机下，Mark Word是64bit大小的

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

36.锁的升降级规则

Java SE 1.6 为了提高锁的性能。引入了“偏向锁”和轻量级锁“。

Java SE 1.6 中锁有4种状态。级别从低到高依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态。

锁只能升级不能降级。

37.偏向锁

大多数情况，锁不仅不存在多线程竞争，而且总由同一线程多次获得。当一个线程访问同步块并获取锁时，会在对象头和栈帧中记录存储锁偏向的线程ID,以后该线程在进入和退出同步块时不需要进行 cas操作来加锁和解锁，只需测试一下对象头 Mark Word里是否存储着指向当前线程的偏向锁。如果测试成功，表示线程已经获得了锁，如果失败，则需要测试下Mark Word中偏向锁的标示是否已经设置成1（表示当前时偏向锁),如果没有设置，则使用cas竞争锁，如果设置了，则尝试使用cas将对象头的偏向锁只想当前线程。

38.关闭偏向锁延迟

java6和7中默认启用，但是会在程序启动几秒后才激活，如果需要关闭延迟，

-XX:BiasedLockingStartupDelay=0。

39.如何关闭偏向锁

JVM参数关闭偏向锁:-XX:-UseBiasedLocking=false,那么程序默认会进入轻量级锁状态。

Tips:如果你可以确定程序的所有锁通常情况处于竞态，则可以选择关闭。

40.轻量级锁

线程在执行同步块，jvm会现在当前线程的栈帧中创建用于储存锁记录的空间。并将对象头中的Mark Word复制到锁记录中。然后线程尝试使用cas将对象头中的Mark Word替换为之乡锁记录的指针。如果成功，当前线程获得锁，如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。

41.轻量锁的解锁

轻量锁解锁时，会使原子操作cas将 displaced Mark Word 替换回对象头，如果成功则表示没有竞争发生，如果失败，表示存在竞争，此时锁就会膨胀为重量级锁。

42.锁的优缺点对比

锁	优　点	缺　点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳秒级的差距	如果线程间存在锁竞争，会带来额外的锁撤销的消耗	适用于只有一个线程访问同步块场景
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度	如果始终得不到锁竞争的线程，使用自旋会消耗 CPU	追求响应时间 同步块执行速度非常快
重量级锁	线程竞争不使用自旋，不会消耗 CPU	线程阻塞，响应时间缓慢	追求吞吐量 同步块执行速度较长

43.什么是原子操作

不可被中断的一个或一系列操作

44.Java如何实现原子操作

Java中通过锁和循环cas的方式来实现原子操作，JVM的CAS操作利用了处理器提供的CMPXCHG指令来实现的。自旋CAS实现的基本思路就是循环进行CAS操作直到成功为止。

45.CAS实现原子操作的3大问题

ABA问题，循环时间长消耗资源大，只能保证一个共享变量的原子操作

46.什么是ABA问题

问题:

因为cas需要在操作值的时候，检查值有没有变化，如果没有变化则更新，如果一个值原来是A,变成了B,又变成了A,那么使用cas进行检测时会发现发的值没有发生变化，其实是变过的。

解决:

添加版本号，每次更新的时候追加版本号，A-B-A —> 1A-2B-3A。

从jdk1.5开始,Atomic包提供了一个类AtomicStampedReference来解决ABA的问题。

47.CAS循环时间长占用资源大问题

如果jvm能支持处理器提供的pause指令，那么效率会有一定的提升。

一、它可以延迟流水线执行指令(de-pipeline),使cpu不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，有些处理器延迟时间是0。

二、它可以避免在退出循环的时候因内存顺序冲突而引起的cpu流水线被清空，从而提高cpu执行效率。

48.CAS只能保证一个共享变量原子操作

一、对多个共享变量操作时，可以用锁。

二、可以把多个共享变量合并成一个共享变量来操作。比如,x=1,k=a,合并xk=1a，然后用cas操作xk。

Tips:java 1.5开始,jdk提供了AtomicReference类来保证饮用对象之间的原子性，就可以把多个变量放在一个对象来进行cas操作。

49.volatile关键字

volatile 是轻量级的synchronized,它在多处理器开发中保证了共享变量的“可见性“。

Java语言规范第3版对volatile定义如下，Java允许线程访问共享变量，为了保证共享变量能准确和一致的更新，线程应该确保排它锁单独获得这个变量。如果一个字段被声明为volatile,Java线程内存模型所有线程看到这个变量的值是一致的。

50.等待/通知机制

一个线程修改了一个对象的值，而另一个线程感知到了变化，然后进行相应的操作。

51.wait

方法wait()的作用是使当前执行代码的线程进行等待，wait()是Object类通用的方法，该方法用来将当前线程置入“预执行队列”中，并在 wait()所在的代码处停止执行，直到接到通知或中断为止。

在调用wait之前线程需要获得该对象的对象级别的锁。代码体现上，即只能是同步方法或同步代码块内。调用wait()后当前线程释放锁。

52.notify

notify()也是Object类的通用方法，也要在同步方法或同步代码块内调用，该方法用来通知哪些可能灯光该对象的对象锁的其他线程，如果有多个线程等待，则随机挑选出其中一个呈wait状态的线程，对其发出 通知 notify，并让它等待获取该对象的对象锁。

53.notify/notifyAll

notify等于说将等待队列中的一个线程移动到同步队列中，而notifyAll是将等待队列中的所有线程全部移动到同步队列中。

54.等待/通知经典范式

等待

```
synchronized(obj) {
    while(条件不满足) {
        obj.wait();
    }
    执行对应逻辑
}
```

通知

```
synchronized(obj) {
    改变条件
    obj.notifyAll();
}
```

55.ThreadLocal

主要解决每一个线程想绑定自己的值，存放线程的私有数据。

56.ThreadLocal使用

获取当前的线程的值通过get(),设置set(T) 方式来设置值。

```
public class XKThreadLocal {

    public static ThreadLocal threadLocal = new ThreadLocal();

    public static void main(String[] args) {
        if (threadLocal.get() == null) {
            System.out.println("未设置过值");
            threadLocal.set("Java小咖秀");
        }
        System.out.println(threadLocal.get());
    }

}
```

输出:

```
未设置过值
Java小咖秀
```

Tips:默认值为null

57.解决get()返回null问题

通过继承重写initialValue()方法即可。

代码实现:

```
public class ThreadLocalExt extends ThreadLocal{

    static ThreadLocalExt threadLocalExt = new ThreadLocalExt();

    @Override
    protected Object initialValue() {
        return "Java小咖秀";
    }

    public static void main(String[] args) {
        System.out.println(threadLocalExt.get());
    }
}
```

输出结果:

```
Java小咖秀
```

58.Lock接口

锁可以防止多个线程同时共享资源。Java5前程序是靠synchronized实现锁功能。Java5之后，并发包新增Lock接口来实现锁功能。

59.Lock接口提供 synchronized不具备的主要特性

特　性	描　述
尝试非阻塞地获取锁	当前线程尝试获取锁，如果这一时刻锁没有被其他线程获取到，则成功获取并持有锁
能被中断地获取锁	与 synchronized 不同，获取到锁的线程能够响应中断，当获取到锁的线程被中断时，中断异常将会被抛出，同时锁会被释放
超时获取锁	在指定的截止时间之前获取锁，如果截止时间到了仍旧无法获取锁，则返回

60.重入锁 ReentrantLock

支持重进入的锁，它表示该锁能够支持一个线程对资源的重复加锁。除此之外，该锁的还支持获取锁时的公平和非公平性选择。

61.重进入是什么意思?

重进入是指任意线程在获取到锁之后能够再次获锁而不被锁阻塞。

该特性主要解决以下两个问题:

一、锁需要去识别获取锁的线程是否为当前占据锁的线程，如果是则再次成功获取。

二、所得最终释放。线程重复n次是获取了锁，随后在第n次释放该锁后，其他线程能够获取到该锁。

62.ReentrantLock默认锁?

默认非公平锁

代码为证:

```
final boolean nonfairTryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            int c = getState();
            if (c == 0) {
                if (compareAndSetState(0, acquires)) {
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            else if (current == getExclusiveOwnerThread()) {
                int nextc = c + acquires;
                if (nextc < 0) // overflow
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);
                return true;
            }
            return false;
        }
```

63.公平锁和非公平锁的区别

公平性与否针对获取锁来说的，如果一个锁是公平的，那么锁的获取顺序就应该符合请求的绝对时间顺序，也就是FIFO。

64.读写锁

读写锁允许同一时刻多个读线程访问，但是写线程和其他写线程均被阻塞。读写锁维护一个读锁一个写锁，读写分离，并发性得到了提升。

Java中提供读写锁的实现类是ReentrantReadWriteLock。

65.LockSupport工具

定义了一组公共静态方法，提供了最基本的线程阻塞和唤醒功能。

方法名称	描　　述
void park()	阻塞当前线程，如果调用 unpark(Thread thread) 方法或者当前线程被中断，才能从 park() 方法返回
void parkNanos(long nanos)	阻塞当前线程，最长不超过 nanos 纳秒，返回条件在 park() 的基础上增加了超时返回
void parkUntil(long deadline)	阻塞当前线程，直到 deadline 时间（从 1970 年开始到 deadline 时间的毫秒数）
void unpark(Thread thread)	唤醒处于阻塞状态的线程 thread

66.Condition接口

提供了类似Object监视器方法，与 Lock配合使用实现等待/通知模式。

67.Condition使用

代码示例:

```
public class XKCondition {
    Lock lock = new ReentrantLock();
    Condition cd = lock.newCondition();

    public void await() throws InterruptedException {
        lock.lock();
        try {
            cd.await();//相当于Object 方法中的wait()
        } finally {
            lock.unlock();
        }
    }

    public void signal() {
        lock.lock();
        try {
            cd.signal(); //相当于Object 方法中的notify()
        } finally {
            lock.unlock();
        }
    }

}
```

68.ArrayBlockingQueue?

一个由数据支持的有界阻塞队列，此队列FIFO原则对元素进行排序。队列头部在队列中存在的时间最长，队列尾部存在时间最短。

69.PriorityBlockingQueue?

一个支持优先级排序的无界阻塞队列，但它不会阻塞数据生产者，而只会在没有可消费的数据时，阻塞数据的消费者。

70.DelayQueue?

是一个支持延时获取元素的使用优先级队列的实现的无界阻塞队列。队列中的元素必须实现Delayed接口和 Comparable接口，在创建元素时可以指定多久才能从队列中获取当前元素。

71.Java并发容器，你知道几个?

ConcurrentHashMap、CopyOnWriteArrayList 、CopyOnWriteArraySet 、ConcurrentLinkedQueue、

ConcurrentLinkedDeque、ConcurrentSkipListMap、ConcurrentSkipListSet、ArrayBlockingQueue、LinkedBlockingQueue、LinkedBlockingDeque、PriorityBlockingQueue、SynchronousQueue、LinkedTransferQueue、DelayQueue

72.ConcurrentHashMap

并发安全版HashMap,java7中采用分段锁技术来提高并发效率，默认分16段。Java8放弃了分段锁，采用CAS，同时当哈希冲突时，当链表的长度到8时，会转化成红黑树。（如需了解细节，见jdk中代码）

73.ConcurrentLinkedQueue

基于链接节点的无界线程安全队列，它采用先进先出的规则对节点进行排序，当我们添加一个元素的时候，它会添加到队列的尾部，当我们获取一个元素时，它会返回队列头部的元素。它采用cas算法来实现。（如需了解细节，见jdk中代码）

74.什么是阻塞队列?

阻塞队列是一个支持两个附加操作的队列，这两个附加操作支持阻塞的插入和移除方法。

1、支持阻塞的插入方法：当队列满时，队列会阻塞插入元素的线程，直到队列不满。

2、支持阻塞的移除方法：当队列空时，获取元素的线程会等待队列变为非空。

75.阻塞队列常用的应用场景?

常用于生产者和消费者场景，生产者是往队列里添加元素的线程，消费者是从队列里取元素的线程。阻塞队列正好是生产者存放、消费者来获取的容器。

76.Java里的阻塞的队列

```
ArrayBlockingQueue:     数组结构组成的 |有界阻塞队列
LinkedBlockingQueue:    链表结构组成的|有界阻塞队列
PriorityBlockingQueue:  支持优先级排序|无界阻塞队列
DelayQueue:             优先级队列实现|无界阻塞队列
SynchronousQueue:       不存储元素| 阻塞队列
LinkedTransferQueue:    链表结构组成|无界阻塞队列
LinkedBlockingDeque:    链表结构组成|双向阻塞队列
```

77.Fork/Join

java7提供的一个用于并行执行任务的框架，把一个大任务分割成若干个小任务，最终汇总每个小任务结果的后得到大任务结果的框架。

78.工作窃取算法

是指某个线程从其他队列里窃取任务来执行。当大任务被分割成小任务时，有的线程可能提前完成任务，此时闲着不如去帮其他没完成工作线程。此时可以去其他队列窃取任务，为了减少竞争，通常使用双端队列，被窃取的线程从头部拿，窃取的线程从尾部拿任务执行。

79.工作窃取算法的有缺点

优点：充分利用线程进行并行计算，减少了线程间的竞争。

缺点：有些情况下还是存在竞争，比如双端队列中只有一个任务。这样就消耗了更多资源。

80.Java中原子操作更新基本类型，Atomic包提供了哪几个类?

AtomicBoolean:原子更新布尔类型

AtomicInteger:原子更新整形

AtomicLong:原子更新长整形

81.Java中原子操作更新数组，Atomic包提供了哪几个类?

AtomicIntegerArray:　原子更新整形数据里的元素

AtomicLongArray:　原子更新长整形数组里的元素

AtomicReferenceArray:　原子更新饮用类型数组里的元素

AtomicIntegerArray:　主要提供原子方式更新数组里的整形

82.Java中原子操作更新引用类型，Atomic包提供了哪几个类?

如果原子需要更新多个变量，就需要用引用类型了。

AtomicReference：原子更新引用类型

AtomicReferenceFieldUpdater: 原子更新引用类型里的字段。

AtomicMarkableReference: 原子更新带有标记位的引用类型。标记位用boolean类型表示，构造方法时AtomicMarkableReference(V initialRef,boolean initialMark)

83.Java中原子操作更新字段类，Atomic包提供了哪几个类?

AtomiceIntegerFieldUpdater:　原子更新整形字段的更新器

AtomiceLongFieldUpdater:　原子更新长整形字段的更新器

AtomiceStampedFieldUpdater:　原子更新带有版本号的引用类型，将整数值

84.JDK并发包中提供了哪几个比较常见的处理并发的工具类?

提供并发控制手段: CountDownLatch、CyclicBarrier、Semaphore

线程间数据交换:　Exchanger

85.CountDownLatch

允许一个或多个线程等待其他线程完成操作。

CountDownLatch的构造函数接受一个int类型的参数作为计数器，你想等待n个点完成，就传入n。

两个重要的方法:

countDown() : 调用时，n会减1。

await() : 调用会阻塞当前线程，直到n变成0。

await(long time,TimeUnit unit) : 等待特定时间后，就不会继续阻塞当前线程。

tips:计数器必须大于等于0，当为0时，await就不会阻塞当前线程。

不提供重新初始化或修改内部计数器的值的功能。

86.CyclicBarrier

可循环使用的屏障。

让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续运行。

CyclicBarrier默认构造放时CyclicBarrier(int parities) ,其参数表示屏障拦截的线程数量，每个线程调用await方法告诉CyclicBarrier我已经到达屏障，然后当前线程被阻塞。

87.CountDownLatch与CyclicBarrier区别

CountDownLatch：

计数器：计数器只能使用一次。

等待： 一个线程或多个等待另外n个线程完成之后才能执行。

CyclicBarrier：

计数器：计数器可以重置（通过reset()方法)。

等待： n个线程相互等待，任何一个线程完成之前，所有的线程都必须等待。

88.Semaphore

用来控制同时访问资源的线程数量，通过协调各个线程，来保证合理的公共资源的访问。

应用场景：流量控制，特别是公共资源有限的应用场景，比如数据链接，限流等。

89.Exchanger

Exchanger是一个用于线程间协作的工具类，它提供一个同步点，在这个同步点上，两个线程可以交换彼此的数据。比如第一个线程执行exchange()方法，它会一直等待第二个线程也执行exchange，当两个线程都到同步点，就可以交换数据了。

一般来说为了避免一直等待的情况，可以使用exchange(V x,long timeout,TimeUnit unit),设置最大等待时间。

Exchanger可以用于遗传算法。

90.为什么使用线程池

几乎所有需要异步或者并发执行任务的程序都可以使用线程池。合理使用会给我们带来以下好处。

- 降低系统消耗：重复利用已经创建的线程降低线程创建和销毁造成的资源消耗。
- 提高响应速度： 当任务到达时，任务不需要等到线程创建就可以立即执行。
- 提供线程可以管理性： 可以通过设置合理分配、调优、监控。

91.线程池工作流程

1、判断核心线程池里的线程是否都有在执行任务，否->创建一个新工作线程来执行任务。是->走下个流程。

2、判断工作队列是否已满，否->新任务存储在这个工作队列里，是->走下个流程。

3、判断线程池里的线程是否都在工作状态，否->创建一个新的工作线程来执行任务，

是->走下个流程。

4、按照设置的策略来处理无法执行的任务。

92.创建线程池参数有哪些，作用?

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

1.corePoolSize:核心线程池大小，当提交一个任务时，线程池会创建一个线程来执行任务，即使其他空闲的核心线程能够执行新任务也会创建，等待需要执行的任务数大于线程核心大小就不会继续创建。

2.maximumPoolSize:线程池最大数，允许创建的最大线程数，如果队列满了，并且已经创建的线程数小于最大线程数，则会创建新的线程执行任务。如果是无界队列，这个参数基本没用。

3.keepAliveTime: 线程保持活动时间，线程池工作线程空闲后，保持存活的时间，所以如果任务很多，并且每个任务执行时间较短，可以调大时间，提高线程利用率。

4.unit: 线程保持活动时间单位，天（DAYS)、小时(HOURS)、分钟(MINUTES、毫秒MILLISECONDS)、微秒(MICROSECONDS)、纳秒(NANOSECONDS)

5.workQueue: 任务队列，保存等待执行的任务的阻塞队列。

一般来说可以选择如下阻塞队列:

ArrayBlockingQueue:基于数组的有界阻塞队列。

LinkedBlockingQueue:基于链表的阻塞队列。

SynchronizedQueue:一个不存储元素的阻塞队列。

PriorityBlockingQueue:一个具有优先级的阻塞队列。

6.threadFactory：设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字。

7. handler: 饱和策略也叫拒绝策略。当队列和线程池都满了，即达到饱和状态。所以需要采取策略来处理新的任务。默认策略是AbortPolicy。

   AbortPolicy:直接抛出异常。

   CallerRunsPolicy: 调用者所在的线程来运行任务。

   DiscardOldestPolicy:丢弃队列里最近的一个任务，并执行当前任务。

   DiscardPolicy:不处理，直接丢掉。

   当然可以根据自己的应用场景，实现RejectedExecutionHandler接口自定义策略。

93.向线程池提交任务

可以使用execute()和submit() 两种方式提交任务。

execute():无返回值，所以无法判断任务是否被执行成功。

submit():用于提交需要有返回值的任务。线程池返回一个future类型的对象，通过这个future对象可以判断任务是否执行成功，并且可以通过future的get()来获取返回值，get()方法会阻塞当前线程知道任务完成。get(long timeout,TimeUnit unit)可以设置超市时间。

94.关闭线程池

可以通过shutdown()或shutdownNow()来关闭线程池。它们的原理是遍历线程池中的工作线程，然后逐个调用线程的interrupt来中断线程，所以无法响应终端的任务可以能永远无法停止。

shutdownNow首先将线程池状态设置成STOP,然后尝试停止所有的正在执行或者暂停的线程，并返回等待执行任务的列表。

shutdown只是将线程池的状态设置成shutdown状态，然后中断所有没有正在执行任务的线程。

只要调用两者之一，isShutdown就会返回true,当所有任务都已关闭，isTerminaed就会返回true。

一般来说调用shutdown方法来关闭线程池，如果任务不一定要执行完，可以直接调用shutdownNow方法。

95.线程池如何合理设置

配置线程池可以从以下几个方面考虑。

- 任务是cpu密集型、IO密集型或者混合型
- 任务优先级，高中低。
- 任务时间执行长短。
- 任务依赖性：是否依赖其他系统资源。

  cpu密集型可以配置可能小的线程,比如 n + 1个线程。

  io密集型可以配置较多的线程，如 2n个线程。

  混合型可以拆成io密集型任务和cpu密集型任务，

  如果两个任务执行时间相差大，否->分解后执行吞吐量将高于串行执行吞吐量。

否->没必要分解。

可以通过Runtime.getRuntime().availableProcessors()来获取cpu个数。

建议使用有界队列，增加系统的预警能力和稳定性。

96.Executor

从JDK5开始，把工作单元和执行机制分开。工作单元包括Runnable和Callable,而执行机制由Executor框架提供。

97.Executor框架的主要成员

ThreadPoolExecutor ：可以通过工厂类Executors来创建。

可以创建3种类型的ThreadPoolExecutor：SingleThreadExecutor、FixedThreadPool、CachedThreadPool。

ScheduledThreadPoolExecutor ： 可以通过工厂类Executors来创建。

可以创建2中类型的ScheduledThreadPoolExecutor：ScheduledThreadPoolExecutor、SingleThreadScheduledExecutor

Future接口:Future和实现Future接口的FutureTask类来表示异步计算的结果。

Runnable和Callable:它们的接口实现类都可以被ThreadPoolExecutor或ScheduledThreadPoolExecutor执行。Runnable不能返回结果，Callable可以返回结果。

98.FixedThreadPool

可重用固定线程数的线程池。

查看源码:

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

corePoolSize 和maxPoolSize都被设置成我们设置的nThreads。

当线程池中的线程数大于corePoolSize ,keepAliveTime为多余的空闲线程等待新任务的最长时间，超过这个时间后多余的线程将被终止，如果设为0，表示多余的空闲线程会立即终止。

工作流程:

1.当前线程少于corePoolSize,创建新线程执行任务。

2.当前运行线程等于corePoolSize,将任务加入LinkedBlockingQueue。

3.线程执行完1中的任务，会循环反复从LinkedBlockingQueue获取任务来执行。

LinkedBlockingQueue作为线程池工作队列（默认容量Integer.MAX_VALUE)。因此可能会造成如下赢下。

1.当线程数等于corePoolSize时，新任务将在队列中等待，因为线程池中的线程不会超过corePoolSize。

2.maxnumPoolSize等于说是一个无效参数。

3.keepAliveTime等于说也是一个无效参数。

4.运行中的FixedThreadPool(未执行shundown或shundownNow))则不会调用拒绝策略。

5.由于任务可以不停的加到队列，当任务越来越多时很容易造成OOM。

99.SingleThreadExecutor

是使用单个worker线程的Executor。

查看源码:

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

corePoolSize和maxnumPoolSize被设置为1。其他参数和FixedThreadPool相同。

执行流程以及造成的影响同FixedThreadPool.

100.CachedThreadPool

根据需要创建新线程的线程池。

查看源码:

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

corePoolSize设置为0，maxmumPoolSize为Integer.MAX_VALUE。keepAliveTime为60秒。

工作流程:

1.首先执行SynchronousQueue.offer (Runnable task)。如果当前maximumPool 中有空闲线程正在执行SynchronousQueue.poll(keepAliveTIme,TimeUnit.NANOSECONDS)，那么主线程执行offer操作与空闲线程执行的poll操作配对成功，主线程把任务交给空闲线程执行,execute方 法执行完成;否则执行下面的步骤2。

2. 当初始maximumPool为空或者maximumPool中当前没有空闲线程时，将没有线程执行SynchronousQueue.poll (keepAliveTime，TimeUnit.NANOSECONDS)。这种情况下，步骤 1将失败。此时CachedThreadPool会创建一个新线程执行任务，execute()方法执行完成。

3.在步骤2中新创建的线程将任务执行完后，会执行SynchronousQueue.poll (keepAliveTime，TimeUnit.NANOSECONDS)。这个poll操作会让空闲线程最多在SynchronousQueue中等待60秒钟。如果60秒钟内主线程提交了一个新任务(主线程执行步骤1)，那么这个空闲线程将执行主线程提交的新任务;否则，这个空闲线程将终止。由于空闲60秒的空闲线程会被终止,因此长时间保持空闲的CachedThreadPool不会使用任何资源。

一般来说它适合处理时间短、大量的任务。

参考:

- 《Java多线程编程核心技术》
- 《Java高并发编程详解》
- 《Java 并发编程的艺术》

JVM

1.JDK、JRE、JVM关系?

Jdk (Java Development Kit) : java语言的软件开发包。包括Java运行时环境Jre。

Jre （Java Runtime Environment) :Java运行时环境，包括Jvm。

Jvm (Java Virtual Machine) :

- 一种用于计算机设备的规范。
- Java语言在不同平台上运行时不需要重新编译。Java语言使用Java虚拟机屏蔽了与具体平台相关的信息，使得Java语言编译程序只需生成在Java虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。

Jdk包括Jre，Jre包括jvm。

2.启动程序如何查看加载了哪些类，以及加载顺序?

java -XX:+TraceClassLoading 具体类

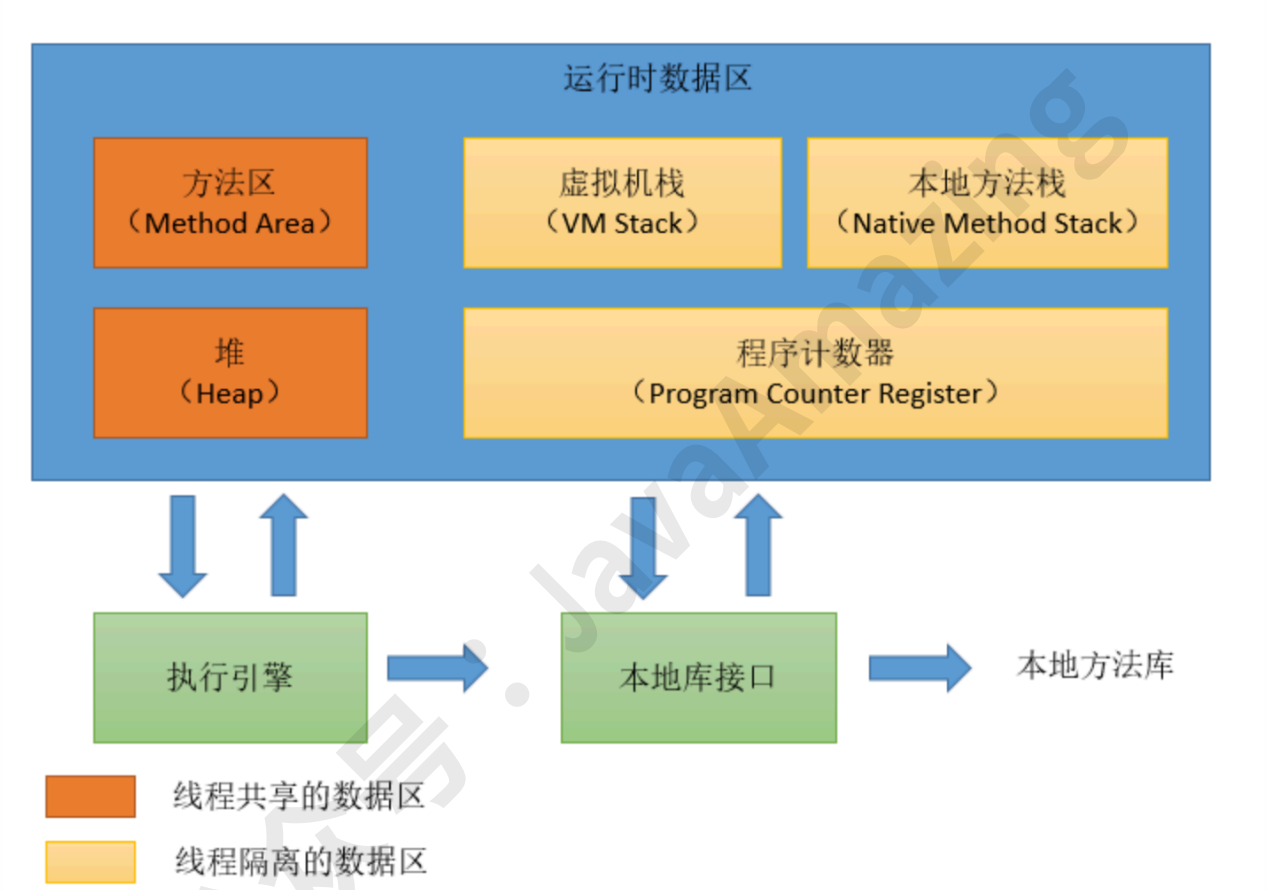
Java -verbose 具体类

3. class字节码文件10个主要组成部分?

- MagicNumber

- Version
- Constant_pool
- Access_flag
- This_class
- Super_class
- Interfaces
- Fields
- Methods
- Attributes

4.画一下jvm内存结构图?



5.程序计数器

属于线程私有内存。占用一块非常小的空间，它的作用可以看作是当前线程所执行的字节码的行号指示器。字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的指令的字节码，分支、循环、跳转、异常处理、线程恢复等基础功能都依赖这个计数器来完成。

6.Java虚拟机栈

属于线程私有内存。它的生命周期与线程相同，虚拟机栈描述的是Java方法执行内存模型；每个方法被执行的时候都会同时创建一个栈桢用于存储局部变量表、操作栈、动态链接、方法出口信息等。每一个方法被调用直至执行完成的过程，就对应着一个栈帧再虚拟机中从入栈到出栈的过程。

7.本地方法栈

本地方法栈与虚拟机栈所发挥的作用是非常相似的，只不过虚拟机栈对虚拟机执行Java方法服务，而本地栈是为虚拟机使用到Native方法服务。

8.Java堆

是Java虚拟机所管理的内存中最大的一块。Java堆事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。

Tips：但随着JIT编译器的发展与逃逸分析技术的逐渐成熟，栈上分配、标亮替换优化技术将会导师一些微妙的变化发生，所有的对象都分配在堆上就不那么绝对了。

9.方法区

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。

10.运行时常量池?

是方法区的一部分，Class文件中除了有类的版本、字段、方法、接口等描述信息，还有一项是常量池（Constant PoolTable）用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后存放道方法区的运行时常量池中。

11.什么时候抛出StackOverflowError?

如果线程请求的栈深度大于虚拟机所允许的深度，则抛出StackOverflowError。

12.Java7和Java8在内存模型上有什么区别?

Java8取消了永久代，用元空间（Metaspace）代替了，元空间是存在本地内存（Native memory)中。

13.程序员最关注的两个内存区域?

堆(Heap)和栈（Stack)，一般大家会把Java内存分为堆内存和栈内存，这是一种比较粗糙的划分方式，但实际上Java内存区域是很复杂的。

14.直接内存是什么?

直接内存并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域，但这部分内存也频繁被实用，也有OutOfMemoryError异常的出现的可能。

Tips:JDK1.4中加入了NIO（new input/output)类，引入了一种基于通道(Channe)与缓冲区（Buffer)的I/O方式，也可以使用Native函数库直接分配堆外内存,然后通过一个存储在Java堆里面的DirectByteBuffer的对象作为这块内存的引用进行操作。

15.除了哪个区域外，虚拟机内存其他运行时区域都会发生OutOfMemoryError?

程序计数器。

16.什么情况下会出现堆内存溢出?

堆内存存储对象实例。我们只要不断地创建对象。并保证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。就会在对象数量到达最大。堆容量限制后，产生内存溢出异常。

17.如何实现一个堆内存溢出?

```
public class Cat {

    public static void main(String[] args) {
        List list = new ArrayList();
        while (true) {
           list.add(new Cat());
        }
    }

}
```

18.空间什么情况下会抛出OutOfMemoryError?

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。

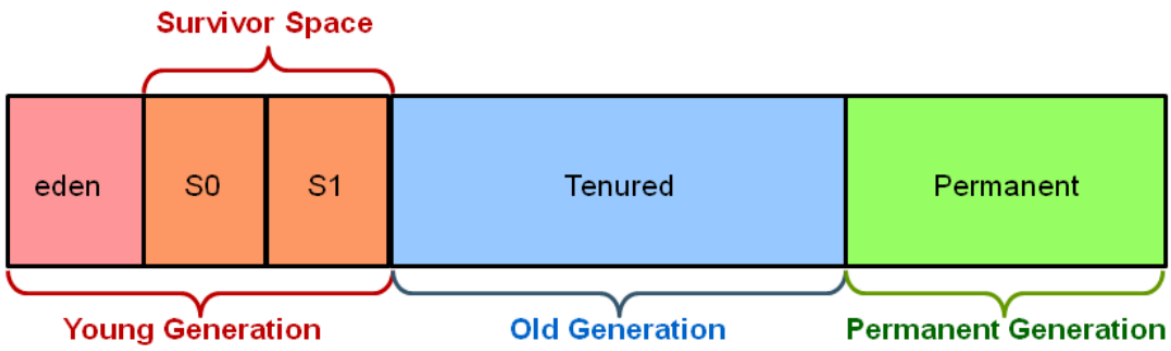
19.如何实现StrackOverflowError?

```
public static void main(String[] args) {
    eat();
}

public static void eat () {
    eat();
}
```

20.如何设置直接内存容量?

通过 -XX:MaxDirectMemorySize指定，如果不指定，则默认与Java堆的最大值一样。

21.Java堆内存组成?



堆大小 = 新生代 + 老年代。如果是Java8则没有Permanent Generation。

其中新生代(Young) 被分为 Eden和S0（from)和S1(to)。

22.Edem : from : to默认比例是?

Edem : from : to = 8 : 1 : 1

此比例可以通过 –XX:SurvivorRatio 来设定

23.垃圾标记阶段?

在GC执行垃圾回收之前，为了区分对象存活与否，当对象被标记为死亡时，GC才回执行垃圾回收，这个过程就是垃圾标记阶段。

24.引用计数法?

比如对象a,只要任何一个对象引用了a,则a的引用计数器就加1，当引用失效时，引用计数器就减1，当计数器为0时，就可以对其回收。

但是无法解决循环引用的问题。

25.根搜索算法?

跟搜索算法是以跟为起始点，按照从上到下的方式搜索被根对象集合所连接的目标对象是否可达(使用根搜索算法后，内存中 的存活对象都会被根对象集合直接或间接连接着)，如果目标对象不可达，就意味着该对象已经死亡，便可以在 instanceOopDesc的 Mark World 中将其标记为垃圾对象。

在根搜索算法中， 只有能够被根对象集合直接或者间接连接的对象才是存活对象。

26.JVM中三种常见的垃圾收集算法?

标记-清除算法（Mark_Sweep)

复制算法(Copying)

标记-压缩算法（Mark-Compact)

27.标记-清除算法?

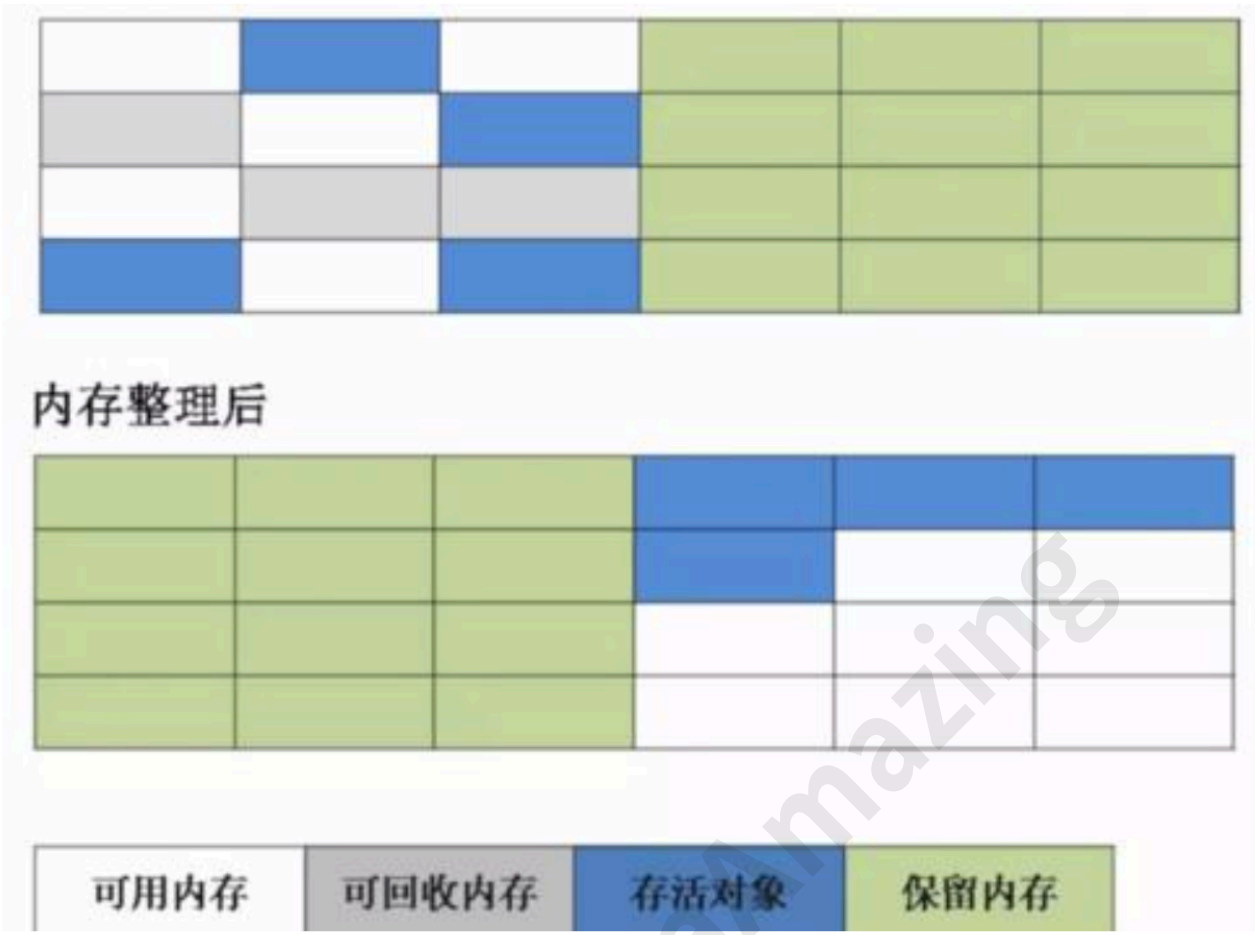
首先标记出所有需要回收的对象，在标记完成后统一回收掉所有的被标记对象。

缺点：

- 标记和清除的效率都不高。
- 空间问题，清除后产生大量不连续的内存随便。如果有大对象会出现空间不够的现象从而不得不提前触发另一次垃圾收集动作。

28.复制算法?

他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，当这一块内存用完了，就将还存活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。



优点:

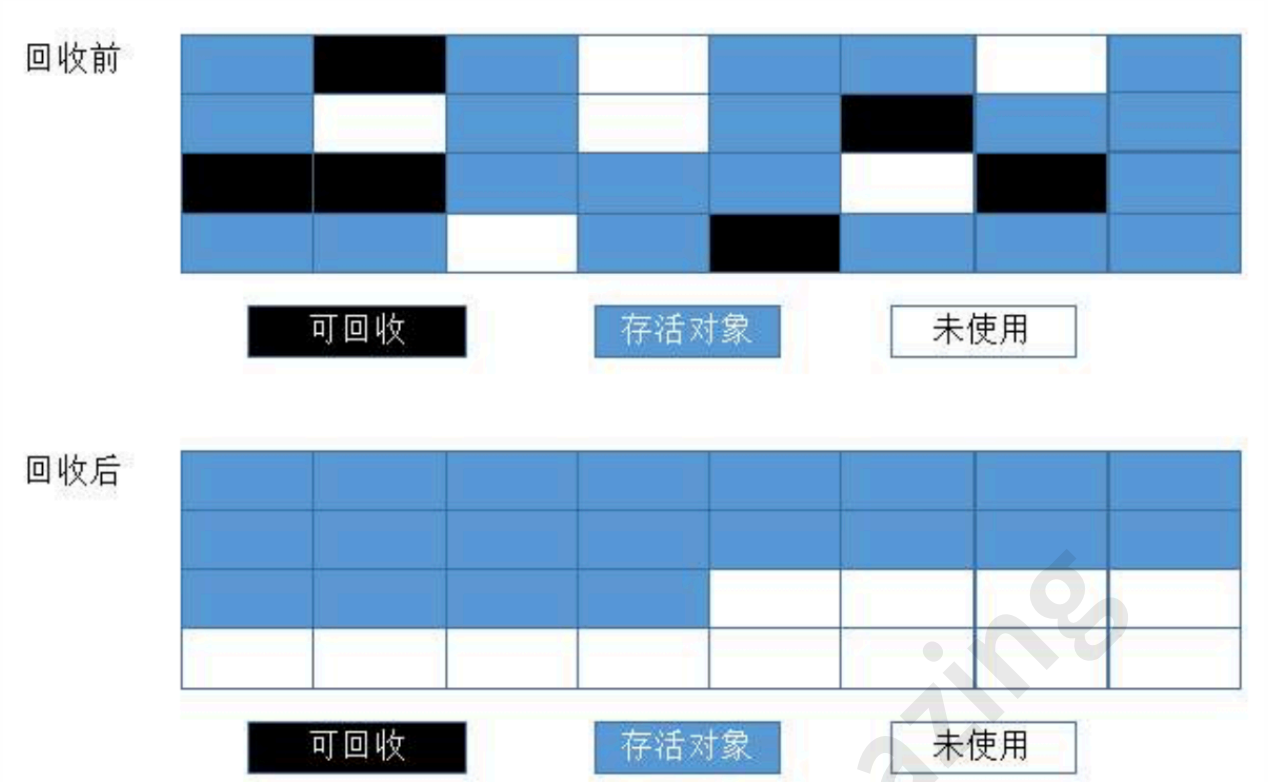
解决了内存碎片问题。

缺点:

将原来的内存缩小为原来的一半，存活对象越多效率越低。

29.标记-整理算法?

先标记出要被回收的对象，然后让所有存活的对象都向一端移动，然后直接清理掉边界以外的内存。解决了复制算法和标记清理算法的问题。

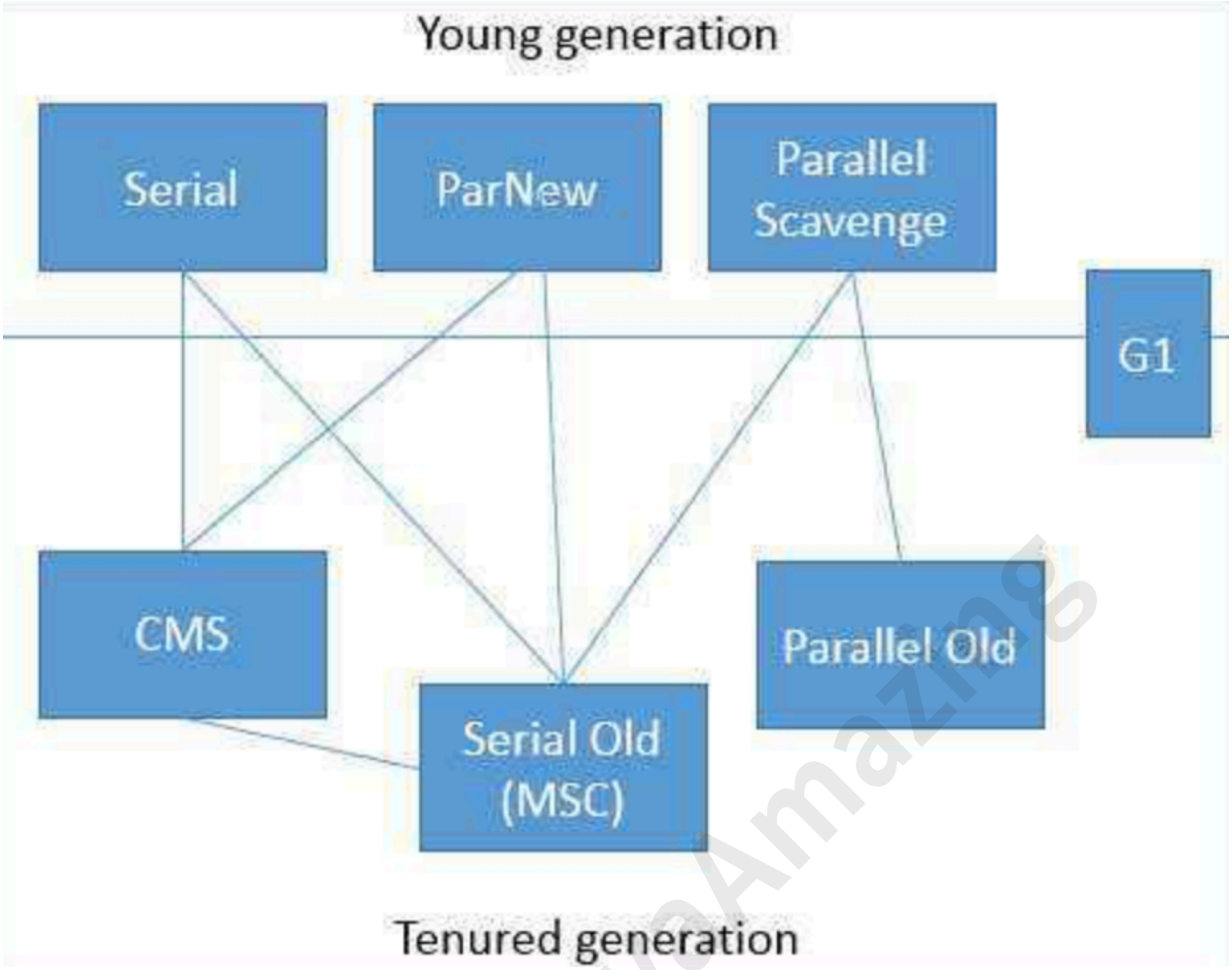


30.分代收集算法?

当前商业虚拟机的垃圾收集都采用“分代手机算法”，其实就根据对象存活周期的不同将内存划分为几块，一般是新老年代。根据各个年代的特点采用最适当的收集算法。

31.垃圾收集器?

如果说垃圾收集算法是方法论，那么垃圾收集器就是具体实现。连线代表可以搭配使用。



32.Stop The World?

进行垃圾收集时，必须暂停其他所有工作线程，Sun将这种事情叫做"Stop The World"。

33.Serial收集器?

单线程收集器，单线程的含义在于它会 stop the world。垃圾回收时需要stop the world ,直到它收集结束。所以这种收集器体验比较差。

34.PartNew收集器?

Serial收集器的多线程版本，除了使用采用并行收回的方式回收内存外，其他行为几乎和Serial没区别。

可以通过选项“-XX:+UseParNewGC”手动指定使用 ParNew收集器执行内存回收任务。

36.Parallel Scavenge?

是一个新生代收集器，也是复制算法的收集器，同时也是多线程并行收集器，与PartNew 不同是，它重点关注的是程序达到一个可控制的吞吐量(Thoughput，CPU 用于运行用户代码 的时间/CPU 总消耗时间，即吞吐量=运行用户代码时间/(运行用户代码时间+垃圾收集时间))， 高吞吐量可以最高效率地利用 CPU 时间，尽快地完成程序的运算任务，主要适用于在后台运算而不需要太多交互的任务。

他可以通过2个参数精确的控制吞吐量,更高效的利用cpu。

分别是: -XX:MaxCcPauseMillis 和 -XX:GCTimeRatio

37.Parallel Old收集器?

Parallel Scavenge收集器的老年代版本，使用多线程和标记-整理算法。JDK 1.6中才开始提供。

38.CMS 收集器?

Concurrent Mar Sweep 收集器是一种以获取最短回收停顿时间为目标的收集器。重视服务的响应速度，希望系统停顿时间最短。采用标记-清除的算法来进行垃圾回收。

39.CMS垃圾回收的步骤?

1. 初始标记 (stop the world)
2. 并发标记
3. 重新标记 (stop the world)
4. 并发清除

   初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快。

   并发标记就是进行Gc Roots Tracing的过程。

   重新标记则是为了修正并发标记期间，因用户程序继续运行而导致的标记产生变动的那一部分对象的标记记录，这个阶段停顿时间一般比初始标记时间长，但是远比并发标记时间短。

   整个过程中并发标记时间最长，但此时可以和用户线程一起工作。

41.CMS收集器优点？缺点?

优点:

并发收集、低停顿

缺点:

- 对cpu资源非常敏感。
- 无法处理浮动垃圾。
- 内存碎片问题。

42.G1收集器?

Garbage First 收集器是当前收集器技术发展的最前沿成果。jdk 1.6_update14中提供了 g1收集器。

G1收集器是基于标记-整理算法的收集器，它避免了内存碎片的问题。

可以非常精确控制停顿时间，既能让使用者明确指定一个长度为 M毫秒的时间片段内，消耗在垃圾收集上的时间不多超过N毫秒，这几乎已经是实时java(rtsj)的垃圾收集器特征了。

42. G1收集器是如何改进收集方式的?

极力避免全区域垃圾收集，之前的收集器进行收集的范围都是整个新生代或者老年代。而g1将整个Java堆（包括新生代、老年代）划分为多个大小固定的独立区域，并且跟踪这些区域垃圾堆积程度，维护一个优先级李彪，每次根据允许的收集时间，优先回收垃圾最多的区域。从而获得更高的效率。

43.虚拟机进程状况工具?

jps (Jvm process status tool),他的功能与ps类似。

可以列出正在运行的虚拟机进程，并显示执行主类(Main Class,main()函数所在的类）的名称，以及浙西进程的本地虚拟机的唯一ID。

语法： jps [options] [hostid]

-q 主输出lvmid,省略主类的名称

-m 输出虚拟机进程启动时传递给主类main()函数的参数

-l 输出主类全名，如果进程执行是Jar包，输出Jar路径

-v 输出虚拟机进程启动时JVM参数

44.虚拟机统计信息工具?

jstat(JVM Statistics Montoring Tool)是用于监视虚拟机各种运行状态信息命令行工机具。他可以显示本地或远程虚拟机进程中的类装载、内存、垃圾收集、jit编译等运行数据。

jstat [option vmid [interval[s|ms] [count]]]

interval 查询间隔

count 查询次数

如果不用这两个参数，就默认查询一次。

option代表用户希望查询的虚拟机信息，主要分3类:

- 类装载
- 垃圾收集
- 运行期编译状况

45.jstat 工具主要选项?

-class 监视类装载、卸载数量、总空间及类装载锁消耗的时间

-gc 监视Java堆状况，包括Eden区，2个survivor区、老年代

-gccapacity 监视内容与-gc基本相同，但输出主要关注Java堆各个区域使用的最大和最小空间

-gcutil 监视内容与-gc基本相同，主要关注已经使用空间站空间百分比

-gccause 与-gcutil 功能一样，但是会额外输出导致上一次GC产生的原因

-gcnew 监视新生代的GC的状况

-gcnewcapacity 监视内容与 -gcnew基本相同，输出主要关注使用到的最大最小空间

-gcold 监视老年代的GC情况

-gcoldcapacity 监控内容与 -gcold基本相同，主要关注使用到的最大最小空间

-compiler 输出jit 编译器编译过的方法、耗时等信息

45.配置信息工具?

jinfo(Configuration Info for Java)的作用是实时地查看和调整虚拟机的各项参数。

使用jps 命令的 -v 参数可以查看虚拟机启动时显示指定的参数列表。

jinfo 语法: jinfo [option] pid

46.内存映像工具?

jmap(Memory Map for Java) 命令用于生成堆转储快照（一般称为heapdump或dump文件）。

语法： jmap [option] vmid

它还可以查询finalize执行队列，Java堆和永久代的详细信息，如果空间使用率、当前用的是哪种收集器等。

- -dump 生成Java堆转储快照，其中live自参数说明是否只dump出存活对象
- -finalizerinfo 显示在F -Queue 中等待Finalizer线程执行finalize方法的对象。只在Linux/Solaris平台下有效
- -heap 显示Java堆详细信息，如使用哪种回收器、参数配置、分代状况。
- -histo 显示堆中对象统计信息、包括类、实例数量和合计容量。
- -F 当虚拟机进程对-dump选项没有响应时，可使用这个选项强制生成dump快照。

47.虚拟机堆转存储快照分析工具?

jhat (JVM Heap Analysis Tool) 用来分析jmap生成的堆转储快照。

48.堆栈跟踪工具?

jstack(Stack Trace for Java) 命令用于生成虚拟机当前时刻的线程快照（一般称为thread dump 或javacore文件）。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因。

jstack [option] vmid

-F 当正常输出的请求不被响应时，强制输出线程堆栈

-l 除堆栈外，显示关于锁的附加信息

-m 如果调用本地方法的花，可以显示C/C++ 的堆栈

49.除了命令行，还有什么可视化工具?

JConsole 和 VisualVM，这两个工具是JDK的正式成员。

50.类加载过程?

加载-》验证-》准备-》解析-》初始化-》使用-》卸载

参考:

- 《深入理解JVM & G1 GC》
- 《深入理解Java虚拟机：JVM高级特性与最佳实践》
- 《实战Java虚拟机：JVM故障诊断与性能优化》

Linux

1.什么是Linux?

是一套免费使用和自由传播的类UNIX操作系统，其内核由林纳斯·本纳第克特·托瓦兹于1991年第一次释出，它主要受到Minix和Unix思想的启发，是一个基于POSIX和Unix的多用户、多任务、支持多线程和多CPU的操作系统。它能运行主要的Unix工具软件、应用程序和网络协议。它支持32位和64位硬件。

2.Linux内核主要负责哪些功能

- 系统内存管理
- 软件程序管理
- 硬件设备管理
- 文件系统管理

3.交互方式

控制台终端、图形化终端

4.启动shell

GNU bash shell能提供对linux 系统的交互式访问。作为普通程序运行，通常在用户登陆终端时启动。登录时系统启动的shell依赖与用户账户的配置。

5.bash手册

大多数linux发行版自带以查找shell命令及其他GNU工具信息的在线手册。man命令用来访问linux系统上的手册页面。当用man命令查看手册，使用分页的程序来现实的。

6.登陆后你在的位置?

一般登陆后，你的位置位于自己的主目录中。

7.绝对文件路径?相对文件路径？快捷方式?

绝对文件路径：描述了在虚拟目录结构中该目录的确切位置，以虚拟目录跟目录开始，相当于目录全名。

以正斜线(/)开始，比如 /usr/local。

相对文件路径：允许用户执行一个基于当前位置的目标文件路径。

比如:当前在/usr/local

```
➜  local ls
Caskroom   Frameworks bin        go         lib        sbin       var
Cellar     Homebrew   etc        include    opt        share
➜  local cd go
```

快捷方式(在相对路径中使用):

单点符(.) : 表示当前目录;
双点符(..) : 表示当前目录的父目录。

8.迷路，我的当前位置在哪?

pwd 显示当前目录

```
[root@iz2ze76ybn73dvwmdij06zz local]# pwd
/usr/local
```

9.如何切换目录?

语法: cd destination

destination : 相对文件路径或绝对文件路径

可以跳到存在的任意目录。

10.如何查看目录中的文件？区分哪些是文件哪些是目录?递归查?

ls 命令会用最基本的形式显示当前目录下的文件和目录：

```
➜  local ls
Caskroom   Frameworks bin        go         lib        sbin       var
Cellar     Homebrew   etc        include    opt        share
```

可以看出默认是按照字母序展示的

一般来说，ls命令回显示不同的颜色区分不同的文件类型,如果没有安装颜色插件可以用ls -F来区分哪些是目录（目录带/），哪些是文件（文件不带/）

ls -R 递归展示出目录下以及子目录的文件，目录越多输出越多

11.创建文件？创建目录？批量创建?

创建文件:touch 文件名

批量创建文件: touch 文件名 文件名 …

```
➜  test touch a
➜  test ls
a
➜  test touch b c
➜  test ls
a b c
```

创建目录：mkdir 目录名

批量创建目录: mkdir 目录名 目录名 …

```
➜  test mkdir aa
➜  test mkdir bb cc
➜  test ls
a  aa b  bb c  cc
➜  test ls -F
a   aa/ b   bb/ c   cc/
```

12.删除文件?强制删除？递归删除?

语法: rm destination

-i 询问是否删除,-r 递归删除，-f 强制删除。

rm不能删除有文件的目录,需要递归删除。

```
➜  xktest rm jdk
rm: jdk: is a directory
➜  xktest rm -r jdk
➜  xktest ls
```

rm -i 询问删除,建议大家平时删除多用 -i，确定一下再删除。

```
➜  xktest touch tomcat
➜  xktest rm -i tomcat
remove tomcat? n
```

rm -rf 会直接删除，没有警告信息，使用必须谨慎**。

13.制表符自动补全?

有的时候文件的名字很长，很容易拼出错即使拼写对了也很浪费时间。

```
➜  xktest ls java*
javaxiaokaxiu
```

比如操作javaxiaokaxiu这个文件时，输入到java的时候，然后按制表键(tab)就会补全成javaxiaokaxiu,是不是方便多了。

14.复制文件

语法: cp source target

如果target不存在则直接创建，如果存在，默认不会提醒你是否需要覆盖，需要加-i就会询问你是否覆盖,n否y是。

```
➜  xktest cp a c
➜  xktest cp -i a c
overwrite c? (y/n [n]) y
➜  xktest ls
a c
```

15.重新命名文件？移动文件？

语法： mv soucre target

重命名:

```
➜  xktest ls
➜  xktest touch java
➜  xktest ls
java
➜  xktest mv java java1.8
➜  xktest ls
java1.8
```

移动文件:

新建jdk目录把java1.8文件移动到jdk目录下。

```
➜  xktest ls
java1.8
➜  xktest mkdir jdk
➜  xktest mv java1.8 jdk
➜  xktest ls -R
jdk

./jdk:
java1.8
```

16.什么是链接文件？

如过需要在系统上维护同一文件的两份或者多份副本，除了保存多分单独的物理文件副本之外。还可以采用保存一份物理文件副本和多个虚拟副本的方法，这种虚拟的副本就叫做链接。

17.查看文件类型?字符编码?

语法: file destination

```
➜  apache file tomcat
tomcat: ASCII text
```

可以看出，file命令可以显示文件的类型text以及字符编码ASCII

18.查看整个文件？按照有文本显示行号？无文本显示行号？

语法 : cat destination

-n 显示行号，-b 有文本的显示行号。 （默认是不显示行号的）

```
➜  apache cat -n tomcat
     1  text
     2  text
     3
     4  start
     5  stop
     6  restart
     7  end
➜  apache cat -b tomcat
     1  text
     2  text

     3  start
     4  stop
     5  restart
     6  end
```

19.查看部分文件

语法 : tail destination

默认情况会展示文件的末尾10行。 -n 行数，显示最后n行。

```
➜  apache tail -n 2 tomcat
restart
end
```

语法: head destination

默认情况会展示文件的开头10行。 -n 行数，显示开头n行。

```
➜  apache head -n 2 tomcat
text
text
```

20.数据排序?对数字进行排序？对月份排序？

默认情况下，文件的数据展示是按照原顺序展示的。sort命令可以对文本文件中的数据进行排序。sort默认会把数据当成字符处理。

语法: sort destination

sort -n 所以排序数字时需要用-n，它的含义是说当前排序是的数字。

sort -M 比如月份Jan、Feb、Mar，如果希望它按照月份排序，加入-M就会按照月份的大小来排序。

21.查找匹配数据？反向搜？

语法: grep [options] pattern [file]

该命令会查找匹配执行模式的字符串的行，并输出。

```
➜  apache grep start tomcat
start
restart
```

-v 反向搜

```
➜  apache grep -v start tomcat
text
text

stop
end
```

-n 显示行号

-c 显示匹配的行数

22.压缩工具有哪些?

工　具	文件扩展名	描　述
bzip2	.bz2	采用Burrows-Wheeler块排序文本压缩算法和霍夫曼编码
compress	.Z	最初的Unix文件压缩工具，已经快没人用了
gzip	.gz	GNU压缩工具，用Lempel-Ziv编码
zip	.zip	Windows上PKZIP工具的Unix实现

23.如何压缩文件？如何解压文件?

比如以.gz的格式举例。

压缩语法: gzip destination

```
➜  apache gzip tomcat
➜  apache ls
tomcat.gz
```

解压语法: gunzip destination

```
➜  apache gunzip tomcat.gz
➜  apache ls
tomcat
```

24.Linux广泛使用的归档数据方法?

虽然zip命令能压缩和解压单个文件，但是更多的时候广泛使用tar命令来做归档。

语法: tar function [options] obj1 obj2

选　项	描　述
-C *dir*	切换到指定目录
-f *file*	输出结果到文件或设备*file*
-j	将输出重定向给bzip2命令来压缩内容
-p	保留所有文件权限
-v	在处理文件时显示文件
-z	将输出重定向给gzip命令来压缩内容

```
➜  apache tar -cvf service.tar service1 service2 // 创建规定文件service.tar
a service1
a service2
➜  apache tar -tf service.tar //查看文件中的目录内容
service1
service2
➜  apache tar zxvf service.tar //解压
x service1
x service2
```

25.如何查看命令历史记录?

history 命令可以展示你用的命令的历史记录。

```
 4463  touch service1 service2
 4464  ls
 4465  tar -cvf service.tar service1 service2
 4466  tar -tf service.tar
 4467  tar zxvf service
 4468  tar zxvf service.t
 4469  tar zxvf service.tar
 4470  ls
 4471  tar -zxvf  service.tar
 4472  ls
```

26.查看已有别名?建立属于自己的别名?

alias -p 查看当前可用别名

```
[root@iz2ze76ybn73dvwmdij06zz ~]# alias -p
alias cp='cp -i'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

alias li = 'ls -li' 创建别名

27.什么是环境变量?

bash shell用一个叫作环境变量(environment variable)的特性来存储有关shell会话和工作环境的信息。这项特性允许你在内存中存储数据，以便程序或shell中运行的脚本能够轻松访问到它们。这也是存储持久数据的一种简便方法。

在bash shell中，环境变量分为两类:

全局变量：对于 shell会话和所有生成的子shell都是可见的。
局部变量： 只对创建他们的shell可见。

28.储存用户的文件是?包括哪些信息?

/etc/passwd存储来一些用户有关的信息。

```
[root@iz2ze76ybn73dvwmdij06zz ~]# cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
```

文件信息包括如下内容。

- 登录用户名
- 用户密码
- 用户账户的UID(数字形式)
- 用户账户的组ID(GID)(数字形式)
- 用户账户的文本描述(称为备注字段)
- 用户HOME目录的位置
- 用户的默认shell

29.账户默认信息？添加账户？删除用户?

```
[root@iz2ze76ybn73dvwmdij06zz ~]# useradd -D//查看系统默认创建用户信息
GROUP=100
HOME=/home
INACTIVE=-1
EXPIRE=
SHELL=/bin/bash
SKEL=/etc/skel
CREATE_MAIL_SPOOL=yes
[root@iz2ze76ybn73dvwmdij06zz ~]# useradd xiaoka//添加用户

[root@iz2ze76ybn73dvwmdij06zz /]# userdel xiaoka//删除用户
```

30.查看组信息？如何创建组？删除组？

```
[root@iz2ze76ybn73dvwmdij06zz ~]# cat /etc/group
root:x:0:
bin:x:1:
daemon:x:2:
sys:x:3:
adm:x:4:
tty:x:5:
disk:x:6:
[root@iz2ze76ybn73dvwmdij06zz ~]# groupadd java //创建组
[root@iz2ze76ybn73dvwmdij06zz ~]# groupdel java //创建组
```

31.文件描述符?每个描述符的含义?

```
[root@iz2ze76ybn73dvwmdij06zz xiaoka]# ls -l
总用量 0
-rw-r--r-- 1 root root 0 4月  21 13:17 a
-rw-r--r-- 1 root root 0 4月  21 13:17 b
-rw-r--r-- 1 root root 0 4月  21 13:17 c
-rw-r--r-- 1 root root 0 4月  21 13:17 d
-rw-r--r-- 1 root root 0 4月  21 13:17 e
```

1、文件类型:

- -代表文件
- d代表目录
- l代表链接
- c代表字符型设备
- b代表块设备
- n代表网络设备

2、访问权限符号:

- r代表对象是可读的
- w代表对象是可写的
- x代表对象是可执行的

若没有某种权限，在该权限位会出现单破折线。

3、这3组权限分别对应对象的3个安全级别:

- 对象的属主
- 对象的属组
- 系统其他用户

31.修改权限?

chmod options mode file

比如给文件附加可以执行权限:

```
[root@xiaoka ~]# chmod +x filename
```

32.如何执行可以执行文件?

```
[root@xiaoka ~]# sh sleep.sh
hello,xiaoka
[root@xiaoka ~]# ./sleep.sh
hello,xiaoka
```

33.列出已经安装的包？安装软件？更新软件？卸载?

列出已经安装的包: yum list installed

安装软件: yum install package_name

更新软件: yum update package_name

卸载软件:yum remove package_name //只删除软件包保留数据文件和配置文件

如果不希望保留数据文件和配置文件

可以执行:yum erase package_name

34.源码安装通常的路子?

```
tar -zxvf xx.gz //解包
cd xx
./configure
make
make install
```

35.vim编辑器几种操作模式？基本操作?

操作模式:

- 普通模式
- 插入模式

基础操作:

- h:左移一个字符。
- j:下移一行(文本中的下一行)。
- k:上移一行(文本中的上一行)。
- l:右移一个字符。

vim提供了一些能够提高移动速度的命令:

- PageDown(或Ctrl+F):下翻一屏
- PageUp(或Ctrl+B):上翻一屏。
- G:移到缓冲区的最后一行。
- num G:移动到缓冲区中的第num行。
- gg:移到缓冲区的第一行。

退出vim:

- q:如果未修改缓冲区数据，退出。
- q!:取消所有对缓冲区数据的修改并退出。
- w filename:将文件保存到另一个文件中。
- wq:将缓冲区数据保存到文件中并退出。

36.查看设备还有多少磁盘空间?

df 可以查看所有已挂在磁盘的使用情况。

-m 用兆字节，G代替g字节

```
[root@iz2ze76ybn73dvwmdij06zz ~]# df
文件系统           1K-块     已用      可用 已用% 挂载点
devtmpfs        1931568        0   1931568    0% /dev
tmpfs           1940960        0   1940960    0% /dev/shm
tmpfs           1940960      720   1940240    1% /run
tmpfs           1940960        0   1940960    0% /sys/fs/cgroup
/dev/vda1      41152812  9068544  30180560   24% /
tmpfs            388192        0    388192    0% /run/user/0
```

快速判断某个特定目录是否有超大文件?

默认情况，du会显示当前目录的所有文件、目录、子目录的磁盘使用情况。

```
[root@iz2ze76ybn73dvwmdij06zz src]# du
4 ./debug
4 ./kernels
12
```

37.默认进程信息显示?

ps它能输出运行在系统上的所有程序的许多信息。

默认情况下ps值显示运行在当前控制台下的当前用户的进程。

```
[root@iz2ze76ybn73dvwmdij06zz ~]# ps
  PID TTY          TIME CMD
10102 pts/0    00:00:00 bash
10131 pts/0    00:00:00 ps
```

38.实时监测进程

与ps相比，top可以实时监控进程信息。

```
top - 11:47:01 up 208 days, 23:53,  2 users,  load average: 0.00, 0.01, 0.05
Tasks: 101 total,   1 running, 100 sleeping,   0 stopped,   0 zombie
%Cpu(s):  3.1 us,  3.1 sy,  0.0 ni, 93.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  3881920 total,   142896 free,  3385540 used,   353484 buff/cache
KiB Swap:        0 total,        0 free,        0 used.   239856 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
    1 root      20   0  191088   2884   1424 S   0.0  0.1   8:28.88 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.29 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:19.48 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    7 root      rt   0       0      0      0 S   0.0  0.0   0:29.60 migration/0
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
    9 root      20   0       0      0      0 S   0.0  0.0 113:30.48 rcu_sched
   10 root      rt   0       0      0      0 S   0.0  0.0   1:10.91 watchdog/0
   11 root      rt   0       0      0      0 S   0.0  0.0   1:02.71 watchdog/1
   12 root      rt   0       0      0      0 S   0.0  0.0   0:36.29 migration/1
   13 root      20   0       0      0      0 S   0.0  0.0   0:35.72 ksoftirqd/1
   15 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/1:0H
```

平均负载有3个值:最近1分钟的、最近5分钟的和最近15分钟的平均负载。值越大说明系统 的负载越高。由于进程短期的突发性活动，出现最近1分钟的高负载值也很常见，但如果近15分 钟内的平均负载都很高，就说明系统可能有问题。

39.如何中断一个进程?

在一个终端中， Ctrl + c

通过这个命令许多（不是全部）命令行程序都可以被中断。

40.如何把一个进程放到后台运行?

```
[root@iz2ze76ybn73dvwmdij06zz ~]# ./sleep.sh &
```

此时，进程并不能被Ctrl + c 中断。

41.如何停止一个进程?

kill命令被用来给程序发送信号。如果没有指定信号，默认发送TERM(终止）信号。

语法 : kill [-signal] PID …

1	HUP	挂起。这是美好往昔的痕迹，那时候终端机通过电话线和调制解调器连接到远端的计算机。这个信号被用来告诉程序，控制的终端机已经“挂起”。通过关闭一个终端会话，可以说明这个信号的作用。发送这个信号到终端机上的前台程序，程序会终止。许多守护进程也使用这个信号，来重新初始化。这意味着，当发送这个信号到一个守护进程后，这个进程会重新启动，并且重新读取它的配置文件。Apache 网络服务器守护进程就是一个例子。
2	INT	中断。实现和 Ctrl-c 一样的功能，由终端发送。通常，它会终止一个程序。
9	KILL	杀死。这个信号很特别。鉴于进程可能会选择不同的方式，来处理发送给它的信号，其中也包含忽略信号，这样呢，从不发送 Kill 信号到目标进程。而是内核立即终止这个进程。当一个进程以这种方式终止的时候，它没有机会去做些“清理”工作，或者是保存劳动成果。因为这个原因，把 KILL 信号看作杀手锏，当其它终止信号失败后，再使用它。
15	TERM	终止。这是 kill 命令发送的默认信号。如果程序仍然“活着”，可以接受信号，那么这个信号终止。
18	CONT	继续。在停止一段时间后，进程恢复运行。
19	STOP	停止。这个信号导致进程停止运行，而没有终止。像 KILL 信号，它不被发送到目标进程，因此它不能被忽略。

####

42.验证网络可链接命令是什么？什么原理?

ping。这个 ping 命令发送一个特殊的网络数据包(叫做 IMCP ECHO REQUEST)到一台指定的主机。大多数接收这个包的网络设备将会回复它，来允许网络连接验证。

```
➜  ~ ping www.baidu.com
PING www.a.shifen.com (220.181.38.149): 56 data bytes
64 bytes from 220.181.38.149: icmp_seq=0 ttl=51 time=32.352 ms
64 bytes from 220.181.38.149: icmp_seq=1 ttl=51 time=25.780 ms
64 bytes from 220.181.38.149: icmp_seq=2 ttl=51 time=44.454 ms
64 bytes from 220.181.38.149: icmp_seq=3 ttl=51 time=21.262 ms
64 bytes from 220.181.38.149: icmp_seq=4 ttl=51 time=21.265 ms
64 bytes from 220.181.38.149: icmp_seq=5 ttl=51 time=20.131 ms
```

一旦启动，ping会持续在特定时间（默认1秒）发送数据包。

43.查看某端口是否被占用?

netstat -ntulp | grep 8080

```
[root@iz2ze76ybn73dvwmdij06zz ~]# netstat -ntulp|grep 8080
tcp        0      0 0.0.0.0:8080            0.0.0.0:*               LISTEN
   4517/java
```

参数说明:

- -t (tcp) 仅显示tcp相关选项
- -u (udp)仅显示udp相关选项
- -n 拒绝显示别名，能显示数字的全部转化为数字
- -l 仅列出在Listen(监听)的服务状态
- -p 显示建立相关链接的程序名

44.如何查找匹配的文件？基于文件属性?

find 程序能基于各种各样的属性，搜索一个给 定目录(以及它的子目录)，来查找文件。

find 命令的最简单使用是，搜索一个或多个目录。

普通查找，按照name查找:

```
[root@iz2ze76ybn73dvwmdij06zz ~]# find -name xiaoka
./xiaoka
```

文件类型查找:

比如,输出我们的家目录文件数量

```
[root@iz2ze76ybn73dvwmdij06zz ~]# find ~|wc -l
17130
```

根据文件类型查:

```
[root@iz2ze76ybn73dvwmdij06zz ~]#  find ~ -type d | wc -l
7340
```

find支持的类型: b 块设备文件、 c 字符设备文件、d 目录、f 普通文件、l 符号链接

45.如何查看当前主机名？如何修改？如何重启后生效?

```
[root@iz2ze76ybn73dvwmdij06zz ~]# hostname//查看当前主机名
iz2ze76ybn73dvwmdij06zz
[root@iz2ze76ybn73dvwmdij06zz ~]# hostname xiaoka//修改当前主机名
[root@iz2ze76ybn73dvwmdij06zz ~]# hostname
xiaoka
```

大家知道一般来讲命令重启就会失效，目前基本上用的centos7的比较多，两种方式可以支持重启生效。

一、命令

```
[root@iz2ze76ybn73dvwmdij06zz ~]# hostnamectl set-hostname xiaoka
[root@iz2ze76ybn73dvwmdij06zz ~]# hostname
xiaoka
[root@xiaoka ~]#
```

二、修改配置文件:/etc/hostname

```
[root@xiaoka ~]# vim /etc/hostname
```

46.如何写一条规则，拒绝某个ip访问本机8080端口?

```
iptables -I INPUT -s ip -p tcp -dport 8080 -j REJECT
```

47.哪个文件包含了主机名和ip的映射关系?

/etc/hosts

48.如何用sed只打印第5行?删除第一行？替换字符串?

只打印第5行:

```
➜  apache sed -n "5p" tomcat
stop
```

删除第一行:

```
[root@xiaoka ~]# cat story
Long ago a lion and a bear saw a kid.
They sprang upon it at the same time.
The lion said to the bear, “I caught this kid first, and so this is mine.”
[root@xiaoka ~]# cat story
They sprang upon it at the same time.
The lion said to the bear, “I caught this kid first, and so this is mine.”
```

替换字符串:

```
➜  apache cat story
Long ago a lion and a bear saw a kid.
They sprang upon it at the same time.
The lion said to the bear, “I caught this kid first, and so this is mine.”
➜  apache sed 's#this#that#g' story
Long ago a lion and a bear saw a kid.
They sprang upon it at the same time.
The lion said to the bear, “I caught that kid first, and so that is mine.”
```

49.打印文件第一行到第三行?

文件tomcat中内容:

```
➜  apache cat tomcat
text21
text22
text23
start
stop
restart
end
```

```
➜ apache head -3 tomcat
text21
text22
text23
➜ apache sed -n '1,3p' tomcat
text21
text22
text23
➜ apache awk 'NR>=1&&NR<=3' tomcat
text21
text22
text23
```

50.如何用awk查看第2行倒数第3个字段?

```
➜  apache awk 'NR==3{print $(NF-2)}' story
this
➜  apache cat story
Long ago a lion and a bear saw a kid.
They sprang upon it at the same time.
The lion said to the bear, “I caught this kid first, and so this is mine.”
```

参考:

- 《鸟哥Linux私房菜》
- 《快乐的命令行》
- 《Linux命令行与shell脚本编程大全(第3版）》
- 《Linux从入门到精通》
- 百度百科
-

Mysql

1.什么是数据库?

数据库是“按照数据结构来组织、存储和管理数据的仓库”。是一个长期存储在计算机内的、有组织的、可共享的、统一管理的大量数据的集合。

2.如何查看某个操作的语法?

比如看建表的语法:

```
mysql> ? create table
Name: 'CREATE TABLE'
Description:
Syntax:
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    (create_definition,…)
    [table_options]
    [partition_options]
```

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    [(create_definition,…)]
    [table_options]
    [partition_options]
    [IGNORE | REPLACE]
    [AS] query_expression

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
```

3.MySql的存储引擎有哪些?

MyISAM、 InnoDB、BDB、MEMORY、MERGE、EXAMPLE、NDB Cluster、 ARCHIVE、CSV、BLACKHOLE、FEDERATED。

Tips:InnoDB和BDB提供事务安全表，其他存储引擎都是非事务安全表。

4.常用的2种存储引擎?

1.Myisam是Mysql的默认存储引擎，当create创建新表时，未指定新表的存储引擎时，默认使用Myisam。

每个MyISAM 在磁盘上存储成三个文件。文件名都和表名相同，扩展名分别是 .frm (存储表定义)、.MYD (MYData，存储数据)、.MYI (MYIndex，存储索引)。

数据文件和索引文件可以放置在不同的目录，平均分布io，获得更快的速度。

2.InnoDB 存储引擎提供了具有提交、回滚和崩溃恢复能力的事务安全。但是对比 Myisam 的存储引擎，InnoDB 写的处理效率差一些并且会占用更多的磁盘空间以保留数据和索引。

6.可以针对表设置引擎吗? 如何设置?

可以, ENGINE=xxx 设置引擎。

代码示例:

```
create table person(
   id int primary key auto_increment,
   username varchar(32)
) ENGINE=InnoDB
```

6.选择合适的存储引擎?

选择标准: 根据应用特点选择合适的存储引擎,对于复杂的应用系统可以根据实际情况选择 多种存储引擎进行组合.
下面是常用存储引擎的适用环境:

1. MyISAM: 默认的 MySQL 插件式存储引擎, 它是在 Web、 数据仓储和其他应用环境下最常使用的存储引擎之一。
2. InnoDB:用于事务处理应用程序，具有众多特性，包括 ACID 事务支持。
3. Memory: 将 所有数据保存在RAM 中， 在 需要快速查找引用和其他类似数据的环境下，可 提供极快的访问。

4. Merge:允许 MySQL DBA 或开发人员将一系列等同的 MyISAM 表以逻辑方式组合在一起,并作为 1 个对象引用它们。对于诸如数据仓储等 VLDB 环境十分适合。

7.选择合适的数据类型

前提: 使用适合存储引擎。

选择原则: 根据选定的存储引擎,确定如何选择合适的数据类型下面的选择方法按存储引擎分类 :

1. MyISAM 数据存储引擎和数据列

   MyISAM数据表，最好使用固定长度的数据列代替可变长度的数据列。

2. MEMORY存储引擎和数据列

   MEMORY数据表目前都使用固定长度的数据行存储，因此无论使用CHAR或VARCHAR列都没有关系。两者都是作为CHAR类型处理的。

3. InnoDB 存储引擎和数据列

   建议使用 VARCHAR类型

   对于InnoDB数据表，内部的行存储格式没有区分固定长度和可变长度列(所有数据行 都使用指向数据列值的头指针)，因此在本质上，使用固定长度的CHAR列不一定比使 用可变长度VARCHAR列简单。 因而， 主要的性能因素是数据行使用的存储总量。 由于 CHAR 平均占用的空间多于 VARCHAR，因此使用VARCHAR来最小化需要处理的数据行的存储总 量和磁盘I/O是比较好的。

8.char & varchar

保存和检索的方式不同。它们的最大长度和是否尾部空格被保留等方面也不同。在存储或检索过程中不进行大小写转换。

9.Mysql字符集

mysql服务器可以支持多种字符集 (可以用show character set命令查看所有mysql支持 的字符集)，在同一台服务器、同一个数据库、甚至同一个表的不同字段都可以指定使用不 同的字符集。

mysql的字符集包括字符集(CHARACTER)和校对规则(COLLATION)两个概念。

10.如何选择字符集?

建议在能够完全满足应用的前提下,尽量使用小的字符集。因为更小的字符集意味着能够节省空间、 减少网络传输字节数,同时由于存储空间的较小间接的提高了系统的性能。

有很多字符集可以保存汉字，比如 utf8、gb2312、gbk、latin1 等等，但是常用的是 gb2312 和 gbk。因为 gb2312 字库比 gbk 字库小，有些偏僻字(例如:洺)不能保存，因此 在选择字符集的时候一定要权衡这些偏僻字在应用出现的几率以及造成的影响， 不能做出肯 定答复的话最好选用 gbk。

11.什么是索引?

在关系数据库中，索引是一种单独的、物理的对数据库表中一列或多列的值进行排序的一种存储结构，它是某个表中一列或若干列值的集合和相应的指向表中物理标识这些值的数据页的逻辑指针清单。索引的作用相当于图书的目录，可以根据目录中的页码快速找到所需的内容。

12.索引设计原则?

1. 搜索的索引列，不 一定是所要选择的列。最适合索引的列是出现在WHERE子句中的列，或连接子句中指定的列，而不是出现在SELECT 关键字后的选择列表中的列。
2. 使用惟一索引。考虑某列中值的分布。 对于惟一值的列，索引的效果最好，而具有多个 重复值的列，其索引效果最差。
3. 使用短索引。如果对串列进行索引，应该指定一个前缀长度,只要有可能就应该这做样。 例如，如果有一个 CHAR(200) 列，如果在前 10 个或 20 个字符内，多数值是惟一的， 那么就不要对整个列进行索引。
4. 利用最左前缀。在创建 一个 n 列的索引时，实际是创建了 MySQL 可利用的 n 个索引。 多列索引可起几个索引的作用，因为可利用索引中最左边的列集来匹配行。 这样的列集 称为最左前缀。（这与索引一个列的前缀不同，索引一个列的前缀是利用该的n前个字 符作为索引值 ）
5. 不要过度索引。每个额外的索引都要占用额外的磁盘空间，并降低写操作的性能，这一点我们前面已经介绍 过。在修改表的内容时，索引必须进行更新,有时可能需要重构, 因此， 索引越多，所花的时间越长。

   如果有一个索引很少利用或从不使用，那么会不必要地减缓表的修改速度。 此外，MySQL 在生成一个执行计划时，要考虑各个索引，这也要费时间。

   创建多余的索引给查询优化带来了更多的工作。索引太多，也可能会使 MySQL选择不到所要使用的最好索引。 只保持所需的索引有利于查询优化。 如果想给已索引的表增加索引， 应 该考虑所要增加的索引是否是现有多列索引的最左索引。
6. 考虑在列上进行的比较类型。 索引可用于“ <”、“ < = ”、“ = ”、“ > =”、“ > ”和 BETWEEN 运算。在模式具有一个直接量前缀时，索引也用于 LIKE 运算。如果只将某个列用于其他类型的运算时(如 STRCMP( )) ，对其进行索引没有价值。

13.MySql有哪些索引?

- 数据结构角度

1. BTREE
2. HASH
3. FULLTEXT
4. R-Tree

- 物理存储角度

1、聚集索引（clustered index）

2、非聚集索引（non-clustered index）

- 从逻辑角度

1. 普通索引：仅加速查询
2. 唯一索引：加速查询 + 列值唯一（可以有null）
3. 主键索引：加速查询 + 列值唯一（不可以有null）+ 表中只有一个
4. 组合索引：多列值组成一个索引，专门用于组合搜索，其效率大于索引合并
5. 全文索引：对文本的内容进行分词，进行搜索

14.Hash索引和B+树索引的底层实现原理:

hash索引底层就是hash表,进行查找时,调用一次hash函数就可以获取到相应的键值,之后进行回表查询获得实际数据.B+树底层实现是多路平衡查找树.对于每一次的查询都是从根节点出发,查找到叶子节点方可以获得所查键值,然后根据查询判断是否需要回表查询数据.

那么可以看出他们有以下的不同:

- hash索引进行等值查询更快(一般情况下),但是却无法进行范围查询.

因为在hash索引中经过hash函数建立索引之后,索引的顺序与原顺序无法保持一致,不能支持范围查询.而B+树的的所有节点皆遵循(左节点小于父节点,右节点大于父节点,多叉树也类似),天然支持范围.

- hash索引不支持使用索引进行排序,原理同上.
- hash索引不支持模糊查询以及多列索引的最左前缀匹配.原理也是因为hash函数的不可预测.**AAAA**和**AAAAB**的索引没有相关性.
- hash索引任何时候都避免不了回表查询数据,而B+树在符合某些条件(聚簇索引,覆盖索引等)的时候可以只通过索引完成查询.
- hash索引虽然在等值查询上较快,但是不稳定.性能不可预测,当某个键值存在大量重复的时候,发生hash碰撞,此时效率可能极差.而B+树的查询效率比较稳定,对于所有的查询都是从根节点到叶子节点,且树的高度较低.

因此,在大多数情况下,直接选择B+树索引可以获得稳定且较好的查询速度.而不需要使用hash索引.

15. 非聚簇索引一定会回表查询吗?

不一定,这涉及到查询语句所要求的字段是否全部命中了索引,如果全部命中了索引,那么就不必再进行回表查询.

举个简单的例子,假设我们在员工表的年龄上建立了索引,那么当进行`select age from employee where age < 20`的查询时,在索引的叶子节点上,已经包含了age信息,不会再次进行回表查询.

16.如何查询最后一行记录?

```
select * from table_name order by id desc limit 1;
```

17.MySQL自增id不连续问题?

- 唯一键冲突
- 事务回滚
- 批量申请自增id的策略

18.sql注入问题?

原因:用户传入的参数中注入符合sql的语法，从而破坏原有sql结构语意，达到攻击效果。

19.什么是3NF（范式）?

- 1NF 指的是数据库表中的任何属性都具有原子性的，不可再分解
- 2NF 是对记录的惟一性约束，要求记录有惟一标识，即实体的惟一性
- 3NF是对字段冗余性的约束，即任何字段不能由其他字段派生出来，它要求字段没有冗余

20. NULL和空串判断?

NULL值是没有值，它不是空串。如果指定''(两个单引号，其间没有字符)，这在NOT NULL列中是允许的。空串是一个有效的值，它不是无值。

判断NULL需要用 IS NULL 或者 IS NOT NULL。

21.什么是事务?

可以用来维护数据库的完整性，它保证成批的MySQL操作要么完全执行，要么完全不执行。

22.事务4个特性?

事务是必须满足4个条件（ACID）：

- 原子性 **Atomicity**：一个事务中的所有操作，要么全部完成，要么全部不完成，最小的执行单位。
- 一致性 **Consistency**：事务执行前后，都处于一致性状态。
- 隔离性 **Isolation**：数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。
- 持久性 **Durability**：事务执行完成后，对数据的修改就是永久的，即便系统故障也不会丢失。

23.事务隔离级别分别是?

- READ_UNCOMMITTED
  这是事务最低的隔离级别，它充许另外一个事务可以看到这个事务未提交的数据。解决第一类丢失更新的问题，但是会出现脏读、不可重复读、第二类丢失更新的问题，幻读 。
- READ_COMMITTED
  保证一个事务修改的数据提交后才能被另外一个事务读取，即另外一个事务不能读取该事务未提交的数据。解决第一类丢失更新和脏读的问题，但会出现不可重复读、第二类丢失更新的问题，幻读问题
- REPEATABLE_READ
  保证一个事务相同条件下前后两次获取的数据是一致的 （注意是 一个事务，可以理解为事务间的数据互不影响）解决第一类丢失更新，脏读、不可重复读、第二类丢失更新的问题，但会出幻读。
- SERIALIZABLE
  事务串行执行，解决了脏读、不可重复读、幻读。但效率很差，所以实际中一般不用。

24.InnoDB默认事务隔离级别?如何查看当前隔离级别

可重复读（REPEATABLE-READ）

查看:

```
mysql> select @@global.tx_isolation;
+-----------------+
| @@global.tx_isolation |
+-----------------+
| REPEATABLE-READ       |
+-----------------+
1 row in set, 1 warning (0.01 sec)
```

25.什么是锁?

数据库的锁是为了支持对共享资源进行并发访问，提供数据的完整性和一致性，这样才能保证在高并发的情况下，访问数据库的时候，数据不会出现问题。

26.死锁?

是指两个或两个以上进程执行过程中，因竞争共享资源造成的相互等待现象。

27.如何处理死锁?

- 设置超时时间。超时后自动释放。
- 发起死锁检测，主动回滚其中一条事务，让其他事务继续执行。

28.如何创建用户？授权?

创建用户:

```
CREATE USER 'username'@'host' IDENTIFIED BY 'password';
```

授权:

```
GRANT privileges ON databasename.tablename TO 'username'@'host';
```

- username：用户名
- host：可以登陆的主机地址。本地用户用localhost表示，任意远程主机用通配符%。
- password：登陆密码，密码可以为空表示不需要密码登陆服务器
- databasename: 数据库名称。
- tablename:表名称，*代表所有表。

29.如何查看表结构?

desc table_name;

```
mysql> desc zipkin_spans;
+---------------+---------------+------+-----+---------+-------+
| Field         | Type          | Null | Key | Default | Extra |
+---------------+---------------+------+-----+---------+-------+
| trace_id_high | bigint(20)    | NO   | PRI | 0       |       |
| trace_id      | bigint(20)    | NO   | PRI | NULL    |       |
| id            | bigint(20)    | NO   | PRI | NULL    |       |
| name          | varchar(255) | NO   | MUL | NULL    |       |
| parent_id     | bigint(20)    | YES  |     | NULL    |       |
| debug         | bit(1)        | YES  |     | NULL    |       |
| start_ts      | bigint(20)    | YES  | MUL | NULL    |       |
| duration      | bigint(20)    | YES  |     | NULL    |       |
+---------------+---------------+------+-----+---------+-------+
8 rows in set (0.01 sec)
```

30.Mysql删除表的几种方式？区别?

1.delete : 仅删除表数据，支持条件过滤，支持回滚。记录日志。因此比较慢。

```
delete from table_name;
```

2.truncate: 仅删除所有数据，不支持条件过滤，不支持回滚。不记录日志，效率高于delete。

```
truncate table table_name;
```

3.drop:删除表数据同时删除表结构。将表所占的空间都释放掉。删除效率最高。

```
drop table table_name;
```

31.like走索引吗?

Xxx% 走索引， %xxx不走索引。

32.什么是回表?

在普通索引查到主键索引后，再去主键索引定位记录。等于说非主键索引需要多走一个索引树。

33.如何避免回表?

索引覆盖被查询的字段。

34.索引覆盖是什么?

如果一个索引包含(或覆盖)所有需要查询的字段的值，称为‘覆盖索引’。

35.视图的优缺点?

优点

> 简单化，数据所见即所得
>
> 安全性，用户只能查询或修改他们所能见到得到的数据
>
> 逻辑独立性，可以屏蔽真实表结构变化带来的影响

缺点

> 性能相对较差，简单的查询也会变得稍显复杂
>
> 修改不方便，特变是复杂的聚合视图基本无法修改

36.主键和唯一索引区别?

本质区别，主键是一种约束，唯一索引是一种索引。

主键不能有空值（非空+唯一），唯一索引可以为空。

主键可以是其他表的外键，唯一索引不可以。

一个表只能有一个主键，唯一索引 可以多个。

都可以建立联合主键或联合唯一索引。

主键-》聚簇索引，唯一索引->非聚簇索引。

37.如何随机获取一条记录?

```
SELECT * FROM table_name ORDER BY rand() LIMIT 1;
```

38.Mysql中的数值类型?

整数类型	字节	最小值	最大值
TINYINT	1	有符号-128 无符号 0	有符号 127 无符号 255
SMALLINT	2	有符号-32768 无符号 0	有符号 32767 无符号 65535
MEDIUMINT	3	有符号-8388608 无符号 0	有符号 8388607 无符号 1677215
INT、INTEGER	4	有符号-2147483648 无符号 0	有符号 2147483647 无符号 4294967295
BIGINT	8	有符号-9223372036854775808 无符号 0	有符号 9223372036854775807 无符号 18446744073709551615
浮点数类型	**字节**	**最小值**	**最大值**
FLOAT	4	±1.175494351E-38	±3.402823466E+38
DOUBLE	8	±2.2250738585072014E-308	±1.7976931348623157E+308
定点数类型	**字节**	**描述**	
DEC(M,D), DECIMAL(M,D)	M+2	最大取值范围与 DOUBLE 相同，给定 DECIMAL 的有效取值范围由 M 和 D 决定	
位类型	**字节**	**最小值**	**最大值**
BIT(M)	1～8	BIT(1)	BIT(64)

39.查看当前表有哪些索引?

```
show index from table_name;
```

40.索引不生效的情况?

- 使用不等于查询
- NULL值
- 列参与了数学运算或者函数
- 在字符串like时左边是通配符.比如 %xxx
- 当mysql分析全表扫描比使用索引快的时候不使用索引.
- 当使用联合索引,前面一个条件为范围查询,后面的即使符合最左前缀原则,也无法使用索引.

41.MVVC?

MVCC 全称是多版本并发控制系统，InnoDB 的 MVCC 是通过在每行记录后面保存两个隐藏的列来实现，这两个列一个保存了行的创建时间，一个保存行的过期时间（删除时间）。当然存储的并不是真实的时间而是系统版本号（system version number）。每开始一个新的事务，系统版本号都会自动新增，事务开始时刻的系统版本号会作为事务的版本号，用来查询到每行记录的版本号进行比较。

42.sql语句的执行流程?

客户端连接数据库，验证身份。

获取当前用户权限。

当你查询时，会先去缓存看看，如果有返回。

如果没有，分析器对sql做词法分析。

优化器对sql进行“它认为比较好的优化”。

执行器负责具体执行sql语句。

最后把数据返回给客户端。

43.如何获取select 语句执行计划?

explain sql;

44.explain列有哪些？含义?

Column	含义
id	查询序号
select_type	查询类型
table	表名
partitions	匹配的分区
type	join类型
prossible_keys	可能会选择的索引
key	实际选择的索引
key_len	索引的长度
ref	与索引作比较的列
rows	要检索的行数(估算值)
filtered	查询条件过滤的行数的百分比
Extra	额外信息

一、 id

SQL查询中的序列号。

id列数字越大越先执行，如果说数字一样大，那么就从上往下依次执行。

二、select_type

select_type	类型说明
SIMPLE	简单SELECT(不使用UNION或子查询)
PRIMARY	最外层的SELECT
UNION	UNION中第二个或之后的SELECT语句
DEPENDENT UNION	UNION中第二个或之后的SELECT语句取决于外面的查询
UNION RESULT	UNION的结果
SUBQUERY	子查询中的第一个SELECT
DEPENDENT SUBQUERY	子查询中的第一个SELECT, 取决于外面的查询
DERIVED	衍生表(FROM子句中的子查询)
MATERIALIZED	物化子查询
UNCACHEABLE SUBQUERY	结果集无法缓存的子查询，必须重新评估外部查询的每一行
UNCACHEABLE UNION	UNION中第二个或之后的SELECT，属于无法缓存的子查询

三、table

显示这一行的数据是关于哪张表的。不一定是实际存在的表名。
可以为如下的值：

- <unionM,N>: 引用id为M和N UNION后的结果。
- : 引用id为N的结果派生出的表。派生表可以是一个结果集，例如派生自FROM中子查询的结果。
- : 引用id为N的子查询结果物化得到的表。即生成一个临时表保存子查询的结果。

四、type

这是**最重要的字段之一**，显示查询使用了何种类型。从最好到最差的连接类型依次为：

> system，const，eq_ref，ref，fulltext，ref_or_null，index_merge，unique_subquery，index_subquery，range，index，ALL

1、system

表中只有一行数据或者是空表，这是const类型的一个特例。且只能用于myisam和memory表。如果是Innodb引擎表，type列在这个情况通常都是all或者index

2、const

最多只有一行记录匹配。当联合主键或唯一索引的所有字段跟常量值比较时，join类型为const。其他数据库也叫做唯一索引扫描

3、eq_ref

多表join时，对于来自前面表的每一行，在当前表中只能找到一行。这可能是除了system和const之外最好的类型。当主键或唯一非NULL索引的所有字段都被用作join联接时会使用此类型。

eq_ref可用于使用'='操作符作比较的索引列。比较的值可以是常量，也可以是使用在此表之前读取的表的列的表达式。

> 相对于下面的ref区别就是它使用的唯一索引，即主键或唯一索引，而ref使用的是非唯一索引或者普通索引。
> eq_ref只能找到一行，而ref能找到多行。

4、ref

对于来自前面表的每一行，在此表的索引中可以匹配到多行。若联接只用到索引的最左前缀或索引不是主键或唯一索引时，使用ref类型（也就是说，此联接能够匹配多行记录）。

ref可用于使用'='或'<=>'操作符作比较的索引列。

5、 fulltext

使用全文索引的时候是这个类型。要注意，**全文索引的优先级很高**，若全文索引和普通索引同时存在时，mysql不管代价，优先选择使用全文索引

6、ref_or_null

跟ref类型类似，只是增加了null值的比较。实际用的不多。

7、index_merge

表示查询使用了两个以上的索引，最后取交集或者并集，常见and ，or的条件使用了不同的索引，官方排序这个在ref_or_null之后，但是实际上由于要读取多个索引，性能可能大部分时间都不如range

8、unique_subquery

用于where中的in形式子查询，子查询返回不重复值唯一值，可以完全替换子查询，效率更高。
该类型替换了下面形式的IN子查询的ref：

```
value IN (SELECT primary_key FROM single_table WHERE some_expr)
```

9、index_subquery

该联接类型类似于unique_subquery。适用于非唯一索引，可以返回重复值。

10、range

索引范围查询，常见于使用 =, <>, >, >=, <, <=, IS NULL, <=>, BETWEEN, IN()或者like等运算符的查询中。

11、index

索引全表扫描，把索引从头到尾扫一遍。这里包含两种情况：
一种是查询使用了覆盖索引，那么它只需要扫描索引就可以获得数据，这个效率要比全表扫描要快，因为索引通常比数据表小，而且还能避免二次查询。在extra中显示Using index，反之，如果在索引上进行全表扫描，没有Using index的提示。

12、all

全表扫描，性能最差。

五、possible_keys

查询可能使用到的索引都会在这里列出来。

六、Key

key列显示MySQL实际使用的键（索引）

要想强制MySQL使用或忽视possible_keys列中的索引，可以使用FORCE INDEX、USE INDEX或者IGNORE INDEX。

select_type为index_merge时，这里可能出现两个以上的索引，其他的select_type这里只会出现一个。

七、key_len

表示索引中使用的字节数。

key_len只计算where条件用到的索引长度，而排序和分组就算用到了索引，也不会计算到key_len中。

不损失精确性的情况下，长度越短越好 。

八、ref

表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值。

九、rows

rows 也是一个重要的字段。 这是mysql估算的需要扫描的行数（不是精确值）。

十、Extra

该列包含MySQL解决查询的详细信息,有以下几种情况：

- Using where:列数据是从仅仅使用了索引中的信息而没有读取实际的行动的表返回的，这发生在对表的全部的请求列都是同一个索引的部分的时候，表示mysql服务器将在存储引擎检索行后再进行过滤。
- Using temporary：表示MySQL需要使用临时表来存储结果集，常见于排序和分组查询。
- Using filesort：MySQL中无法利用索引完成的排序操作称为“文件排序”。
- Using join buffer：改值强调了在获取连接条件时没有使用索引，并且需要连接缓冲区来存储中间结果。如果出现了这个值，那应该注意，根据查询的具体情况可能需要添加索引来改进能。
- Impossible where：这个值强调了where语句会导致没有符合条件的行。
- Select tables optimized away：这个值意味着仅通过使用索引，优化器可能仅从聚合函数结果中返回一行。

链接：https://www.jianshu.com/p/8fab76bbf448

45.MySql最多创建多少列索引?

16

46.为什么最好建立一个主键?

主键是数据库确保数据行在整张表唯一性的保障,即使业务上本张表没有主键,也建议添加一个自增长的ID列作为主键.设定了主键之后,在后续的删改查的时候可能更加快速以及确保操作数据范围安全.

47.字段为什么要求建议为not null?

MySQL官网这样介绍:

> NULL columns require additional space in the rowto record whether their values are NULL. For MyISAM tables, each NULL columntakes one bit extra, rounded up to the nearest byte.

null值会占用更多的字节,且会在程序中造成很多与预期不符的情况.

48.varchar(10)和int(10)代表什么含义

varchar的10代表了申请的空间长度,也是可以存储的数据的最大长度,而int的10只是代表了展示的长度,不足10位以0填充.也就是说,int(1)和int(10)所能存储的数字大小以及占用的空间都是相同的,只是在展示时按照长度展示。

49.视图是什么？对比普通表优势?

视图(View)是一种虚拟存在的表，对于使用视图的用户来说基本上是透明的。视图并 不在数据库中实际存在，行和列数据来自定义视图的查询中使用的表，并且是在使用视图时 动态生成的。

视图相对于普通的表的优势主要包括以下几项。

- 简单:使用视图的用户完全不需要关心后面对应的表的结构、关联条件和筛选条件，对用户来说已经是过滤好的复合条件的结果集。
- 安全:使用视图的用户只能访问他们被允许查询的结果集，对表的权限管理并不能限制到某个行某个列，但是通过视图就可以简单的实现。
- 数据独立:一旦视图的结构确定了，可以屏蔽表结构变化对用户的影响，源表增加列对视图没有影响;源表修改列名，则可以通过修改视图来解决，不会造成对访问 者的影响。

50.count(*)在不同引擎的实现方式?

MyISAM :把一个表的总行数存在了磁盘上，执行 count(*) 的时候会直接返回这个数，效率很高。

InnoDB : 比较麻烦，它执行 count(*) 的时候，需要把数据一行一行地从引擎里面读出来，然后累积计数。

参考:

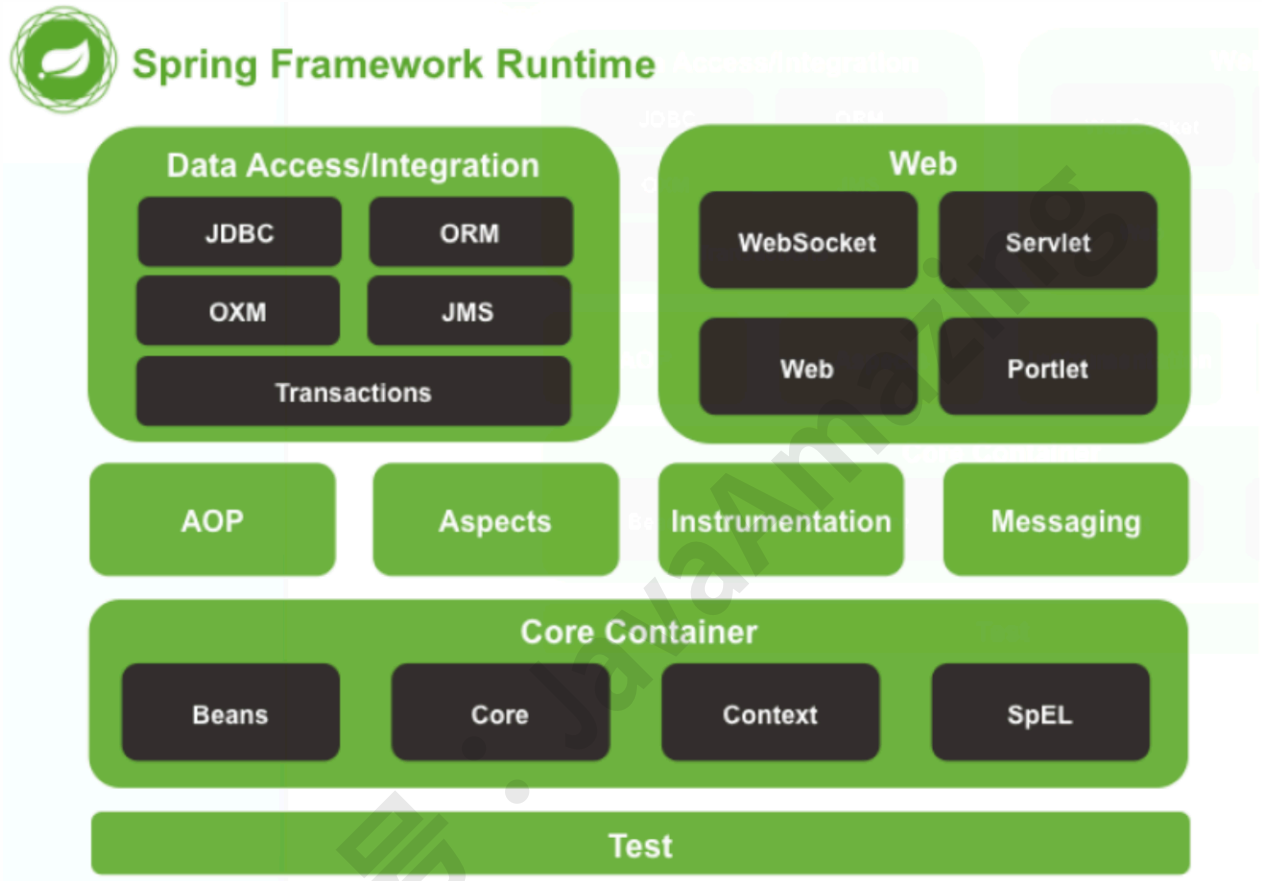
- 《深入浅出MySQL》
- 《高性能MySql》
- 《MySQL技术内幕（第5版)》
- 《MySQL必知必会》
- 极客时间：MySQL实战45讲
- 百度百科

Spring

1.Spring框架?

Spring框架是由于软件开发的复杂性而创建的，Spring使用的是基本的JavaBean来完成以前只可能由EJB完成的事。从简单性、可测性和松耦合性角度而言，绝大部分Java应用都可以用Spring。

2.Spring的整体架构?



大约分为20个模块。

3.Spring可以做什么?

4.Spring的优点?缺点?

优点:

- Spring属于低侵入设计。
- IOC将对象之间的依赖关系交给Spring,降低组件之间的耦合，实现各个层之间的解耦，让我们更专注于业务逻辑。
- 提供面向切面编程。
- 对各种主流插件提供很好的集成支持。
- 对事务支持的很好，只要配置即可，无须手动控制。

缺点:

- 依赖反射，影响性能。

5.你能说几个Spring5的新特性吗?

- spring5整个框架基于java8
- 支持http/2
- Spring Web MVC支持最新API
- Spring WebFlux 响应式编程
- 支持Kotlin函数式编程

6.IOC?

负责创建对象、管理对象(通过依赖注入)、整合对象、配置对象以及管理这些对象的生命周期。

7.什么是依赖注入?

依赖注入是Spring实现IoC的一种重要手段，将对象间的依赖关系的控制权从开发人员手里转移到容器。

8.IOC注入哪几种方式?

1.构造器注入

2.setter注入

3.接口注入（我们几乎不用）

9.IOC优点？缺点?

优点:

- 组件之间的解耦，提高程序可维护性、灵活性。

缺点:

- 创建对象步骤复杂，有一定学习成本。
- 利用反射创建对象，效率上有损。（对于代码的灵活性和可维护性来看，Spring对于我们的开发带来了很大的便利，这点损耗不算什么哦）

10.bean的生命周期?

1.Spring 对bean进行实例化。

2.Spring将值和bean的引用注入到 bean对应的属性中。

3.如果bean实现了BeanNameAware接口，Spring将bean的ID传递给setBeanName()方法。

4.如果bean实现了BeanFactoryAware接口， Spring将调用setBeanFactory()方法，将 bean所在的应用引用传入进来。

5.如果bean实现了ApplicationContextAware接口，Spring将调用setApplicationContext()方法，将bean所在的应用引用传入进来。

6.如果bean实现了BeanPostProcessor 接口，Spring将调用他们的post-ProcessBeforeInitalization()方法。

7.如果bean实现了InitializingBean接口，Spring将调用他们的after-PropertiesSet()方法，类似地，如果bean使用init-method声明了初始化方法，该方法也会被调用。

8.如果bean实现了BeanPostProcessor接口，Spring将调用它们的post-ProcessAfterInitialization()方法。

9.此时， bean已经准备就绪，可以被应用程序使用了，他们将一直驻留在应用上下文中,直到该应用被销毁。

10.如果bean实现了DisposableBean接口，Spring将调用它的destory()接口方法，同样，如果bean使用destroy-method声明了销毁方法，该方法也会被调用。

11.Spring有几种配置方式?

- 基于xml
- 基于注解

- 基于Java

12.Spring中的bean有几种scope?

- singleton: 单例，每一个bean只创建一个对象实例。
- prototype，原型，每次对该bean请求调用都会生成各自的实例。
- request，请求，针对每次HTTP请求都会生成一个新的bean。表示在一次 HTTP 请求内有效。
- session，在一个http session中，一个bean定义对应一个bean实例。
- global session:在一个全局http session中，一个bean定义对应一个bean实例。

13.什么是AOP(面向切面编程)?

在软件业，AOP为Aspect Oriented Programming的缩写，意为：面向切面编程，通过预编译方式和运行期间动态代理实现程序功能的统一维护的一种技术。

14.切面有几种类型的通知？分别是?

前置通知(Before): 目标方法被调用之前调用通知功能。

后置通知(After): 目标方法完成之后调用通。

返回通知(After-returning): 目标方法成功执行之后调用通知。

异常通知(After-throwing): 目标方法抛出异常后调用通知。

环绕通知(Around): 在被通知的方法调用之前和调用之后执行自定义的行为。

15.什么是连接点 （Join point)?

连接点是在应用执行过程中能够插入切面的一个点。这个点可以是调用方法时、抛出异常时、甚至修改一个字段时。

16.什么是切点（Pointcut)?

切点的定义会匹配通知所要织入的一个或多个连接点。我们通常使用明确的类和方法名称，或是利用正则表达式定义所匹配的类和方法名称来指定这些切点。有些AOP框架允许我们创建动态的切点，可以根据运行时的决策(比如方法的参数值)来决定是否应用通知。

17.什么是切面(Aspect)?

切面是通知和切点的结合。通知和切点共同定义了切面的全部内容。

18.织入(Weaving)?

织入是把切面应用到目标对象并创建新的代理对象的过程。切面在指定的连接点被织入到目标对象中。

19.引入（Introduction）?

引入允许我们向现有的类添加新方法或属性。

20.在目标对象的生命周期里有多个点可以进行织入?

- 编译期：切面在目标类编译时被织入。AspectJ的织入编译器就是以这种方式织入切面的。
- 类加载期：切面在目标类加载到JVM时被织入。它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ 5的加载时织入(load-time weaving，LTW)就支持以这种方式织入切面。
- 运行期：切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象动态地创建一个代理对象。Spring AOP就是以这种方式织入切面的。

21.AOP动态代理策略?

- 如果目标对象实现了接口，默认采用JDK 动态代理。可以强制转为CgLib实现AOP。
- 如果没有实现接口，采用CgLib进行动态代理。

22.什么是MVC框架?

MVC全名是Model View Controller，是模型(model)- 视图(view)- 控制器(controller)的缩写，一种软件设计典范，用一种业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。

MVC被独特的发展起来用于映射传统的输入、处理和输出功能在一个逻辑的图形化用户界面的结构中。

23.什么是SpringMVC?

SpringMVC是Spring框架的一个模块。是一个基于MVC的框架。

24.SpringMVC的核心?

DispatcherServlet

25.SpringMVC的几个组件?

DispatcherServlet : 前端控制器，也叫中央控制器。相关组件都是它来调度。

HandlerMapping : 处理器映射器，根据URL路径映射到不同的Handler。

HandlerAdapter : 处理器适配器，按照HandlerAdapter的规则去执行Handler。

Handler ：处理器,由我们自己根据业务开发。

ViewResolver : 视图解析器，把逻辑视图解析成具体的视图。

View : 一个接口，它的实现支持不同的视图类型（freeMaker，JSP等）

26.SpringMVC工作流程?

1.用户请求旅程的第一站是DispatcherServlet。

2.收到请求后，DispatcherServlet调用HandlerMapping，获取对应的Handler。

3.如果有拦截器一并返回。

4.拿到Handler后，找到HandlerAdapter,通过它来访问Handler,并执行处理器。

5.执行Handler的逻辑。

6.Handler会返回一个ModelAndView对象给DispatcherServlet。

7.将获得到的ModelAndView对象返回给DispatcherServlet。

8.请求ViewResolver解析视图，根据逻辑视图名解析成真正的View。

9.返回View给DispatcherServlet。

10.DispatcherServlet对View进行渲染视图。

11.DispatcherServlet响应用户。

27.SpringMVC的优点?

1.具有Spring的特性。

2.可以支持多种视图(jsp,freemaker)等。

3.配置方便。

4.非侵入。

5.分层更清晰，利于团队开发的代码维护，以及可读性好。

Tips:Jsp目前很少有人用了。

28.单例bean是线程安全的吗?

不是。具体线程问题需要开发人员来处理。

29.Spring从哪两个角度实现自动装配?

组件扫描(component scanning):Spring会自动发现应用上下文中所创建的bean。

自动装配(autowiring):Spring自动满足bean之间的依赖。

30.自动装配有几种方式？分别是?

no - 默认设置，表示没有自动装配。

byName： 根据名称装配。

byType： 根据类型装配。

constructor： 把与Bean的构造器入参具有相同类型的其他Bean自动装配到Bean构造器的对应入参中。

autodetect： 先尝试constructor装配，失败再尝试byType方式。

default：由上级标签的default-autowire属性确定。

31.说几个声明Bean 的注解?

- @Component
- @Service
- @Repository
- @Controller

32.注入Java集合的标签?

- 允许有相同的值。
- 不允许有相同的值。
- 键和值都只能为String类型。
- < map > 键和值可以是任意类型。

33.Spring支持的ORM?

1. Hibernate
2. iBatis
3. JPA (Java Persistence API)
4. TopLink
5. JDO (Java Data Objects)
6. OJB

34.@Repository注解?

Dao 层实现类注解，扫描注册 bean。

35.@Value注解?

讲常量、配置中的变量值、等注入到变量中。

36.@Controller注解?

定义控制器类。

37.声明一个切面注解是哪个?

@Aspect

38.映射web请求的注解是?

@RequestMapping

39.@ResponseBody注解?

作用是将返回对象通过适当的转换器转成置顶格式，写进response的body区。通常用来返回json、xml等。

40.@ResponseBody + @Controller =?

@RestController

41.接收路径参数用哪个注解?

@PathVariable

42.@Cacheable注解?

用来标记缓存查询。

43.清空缓存是哪个注解?

@CacheEvict

44.@Component注解?

泛指组件，不好归类时，可以用它。

45.BeanFactory 和 ApplicationContext区别?

BeanFactory	ApplicationContext
它使用懒加载	它使用即时加载
它使用语法显式提供资源对象	它自己创建和管理资源对象
不支持国际化	支持国际化
不支持基于依赖的注解	支持基于依赖的注解

46.@Qualifier注解?

当创建多个相同类型的 bean 时，并且想要用一个属性只为它们其中的一个进行装配，在这种情况下，你可以使用 @Qualifier 注释和 @Autowired 注释通过指定哪一个真正的 bean 将会被装配来消除混乱。

47.事务的注解是?

@Transaction

48.Spring事务实现方式有?

声明式：声明式事务也有两种实现方式。

- xml 配置文件的方式。
- 注解方式（在类上添加 @Transaction 注解）。

编码式：提供编码的形式管理和维护事务。

49.什么是事务传播?

事务在嵌套方法调用中如何传递，具体如何传播，取决于事务传播行为。

50.Spring事务传播行为有哪些?

事务传播行为类型	说明
PROPAGATION_REQUIRED	支持当前事务，如果不存在，就新建一个
PROPAGATION_SUPPORTS	支持当前事务，如果不存在，就不使用事务
PROPAGATION_MANDATORY	支持当前事务，如果不存在，抛出异常
PROPAGATION_REQUIRES_NEW	如果有事务存在，挂起当前事务，创建一个新的事务
PROPAGATION_NOT_SUPPORTED	以非事务方式运行，如果有事务存在，挂起当前事务
PROPAGATION_NEVER	以非事务方式运行，如果有事务存在，抛出异常
PROPAGATION_NESTED	如果当前事务存在，则嵌套事务执行

参考:

- 《Spring in action 4》
- 《SPRING技术内幕》
- 《Spring源码深度解析》
- 《Spring5企业级开发实战》
- https://spring.io
- 百度百科

Mybatis

1.什么是Mybatis?

MyBatis 是一款优秀的支持自定义 SQL 查询、存储过程和高级映射的持久层框架，消除了 几乎所有的 JDBC 代码和参数的手动设置以及结果集的检索 。 MyBatis 可以使用 XML 或注解进 行配置和映射，MyBatis 通过将参数映射到配置的 SQL 形成最终执行的 SQL 语句 ， 最后将执行 SQL 的结果映射成 Java对象返回。

2.Hibernate优点?

Hibernate建立在POJO和数据库表模型的直接映射关系上。通过xml或注解即可和数据库表做映射。通过pojo直接可以操作数据库的数据。它提供的是全表的映射模型。

- 消除代码映射规则，被分离到xml或注解里配置。
- 无需在管理数据库连接，配置在xml中即可。
- 一个会话中，不要操作多个对象，只要操作Session对象即可。
- 关闭资源只需关闭Session即可。

3.Hibernate缺点?

- 全表映射带来的不便，比如更新需要发送所有的字段。
- 无法根据不同的条件组装不同的sql。

- 对多表关联和复杂的sql查询支持较差，需要自己写sql，返回后，需要自己将数据组成pojo。
- 不能有效支持存储过程。
- 虽然有hql,但是性能较差，大型互联网需要优化sql，而hibernate做不到。

4.Mybatis优点?

1. 小巧，学习成本低，会写sql上手就很快了。
2. 比jdbc，基本上配置好了，大部分的工作量就专注在sql的部分。
3. 方便维护管理，sql不需要在Java代码中找，sql代码可以分离出来，重用。
4. 接近jdbc,灵活，支持动态sql。
5. 支持对象与数据库orm字段关系映射。

5.Mybatis缺点?

- 由于工作量在sql上，需要 sql的熟练度高。
- 移植性差。sql语法依赖数据库。不同数据库的切换会因语法差异，会报错。

6.什么时候用Mybatis?

如果你需要一个灵活的、可以动态生成映射关系的框架。目前来说，因为适合，互联网项目用Mybatis的比例还是很高滴。

7.Mybatis的核心组件有哪些？分别是?

SqlSessionFactoryBuilder(构造器):它会根据配置信息或者代码来生成SqlSessionFactory。

SqlSessionFactory（工厂接口）:依靠工厂来生成SqlSession。

SqlSession（会话):是一个既可以发送 sql去执行返回结果，也可以获取Mapper接口。

SQL Mapper:它是新设计的组件，是由一个Java接口和XML文件（或注解）构成的。需要给出对象的SQl和映射规则。它负责发送SQL去执行，并返回结果。

8.#{}和${}的区别是什么?

${}是字符串替换，#{}是预编译处理。一般用#{}防止 sql注入问题。

9.Mybatis中9个动态标签是?

- if
- c h o o s e (when 、 oterwise)
- trim (where、 set)
- foreach
- bind

10.xml映射文件中，有哪些标签?

select | insert | updae | delete | resultMap | parameterMap | sql | include | selectKey 加上9个动态标签，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。

11.Mybatis支持注解吗？优点？缺点?

支持。

优点：对于需求简单sql逻辑简单的系统，效率较高。

缺点： 当sql变化需要重新编译代码，sql复杂时，写起来更不方便，不好维护。

12.Mybatis动态sql?

Mybatis 动态 sql 可以让我们在 Xml 映射文件内，以标签的形式编写动态 sql，完成逻辑判断和动态拼接 sql 的功能

13.Mybatis 是如何进行分页的?分页插件的原理是什么?

1）Mybatis 使用 RowBounds 对象进行分页，也可以直接编写 sql 实现分页，也可以使用 Mybatis 的分页插件。
2）分页插件的原理：实现 Mybatis 提供的接口，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。
举例：select from student，拦截 sql 后重写为：select t. from （select from student）t limit 0，10

14.如何获取自增主键?

注解：

```
@Options(useGeneratedKeys =true, keyProperty =”id”)

int insert( );
```

Xml：

```
<insert id=”insert” useGeneratedKeys=”true” keyProperty=” id”>

sql

</insert>
```

15.为什么Mapper接口没有实现类，却能被正常调用?

这是因为MyBatis在Mapper接口上使用了动态代理。

16.用注解好还是xml好?

简单的增删改查可以注解。

复杂的sql还是用xml,官方也比较推荐xml方式。

xml的方式更便于统一维护管理代码。

17.如果不想手动指定别名，如何用驼峰的形式自动映射?

mapUnderscoreToCamelCase=true

18.当实体属性名和表中字段不一致，怎么办?

一、别名: 比如 order_num

select order_num orderNum

二、通过做映射

<result property = “order_Num" column =”order_num"/>

三、如果是驼峰注解用17问的方式。

19.嵌套查询用什么标签?

association 标签的嵌套查询常用的属性如下 。

select:另一个映射查询的 id, MyBatis会额外执行这个查询获取嵌套对象的结果。

column:列名(或别名)，将主查询中列的结果作为嵌套查询的 参数，配置 方式如 column={propl=coll , prop2=col2}, propl 和 prop2 将作为嵌套查询的参数。

fetch Type:数据加载方式，可选值为 lazy 和 eager，分别为延迟加载和积极加载， 这个配置会覆盖全局的 lazyLoadingEnabled 配置 。

20.like模糊查询怎么写?

（1）'%${value}%' 不推荐

（2）CONCAT('%',#{value},'%') 推荐

(3) like '%'|| #{value} || '%'

（4）使用bind

LIKE #{pattern}

21.Mybatis支持枚举吗?

支持。有转换器EnumTypeHandler和EnumOrdinalTypeHandler

```
<typeHandlers>
    <typeHandler handler="org.apache.ibatis.type.EnumOrdinalTypeHandler"
javaType="枚举类 (com.xx.StateEnum)"/>
</typeHandlers>
```

22.SqlSessionFactoryBuilder生命周期?

利用xml或者Java代码获得资源构建SqlSessionFactoryBuilder，它可以创建多个SessionFactory,一旦构建成功SessionFactory,就可以回收它了。

23.一级缓存的结构?如何开启一级缓存？如何不使用一级缓存?

Map 。默认情况下，一级缓存是开启的。< select >标签内加属性flushCache=true。

24.二级缓存如何配置?

```
<settings>
〈 !—其他自己直— 〉
<setting name=" cacheEnabled" value=" true " />
</settings>
```

这个参数是二级缓存的全局开关，默认值是 true，初始状态 为启用状态 。 如果把这个参数设置为 false，即使有后面 的 二级缓 存配置，也不会生效 。

25.简述 Mybatis 的插件运行原理，以及如何编写一个插件?

1）Mybatis 仅可以编写针对 ParameterHandler、ResultSetHandler、StatementHandler、Executor 这 4 种接口的插件，Mybatis 通过动态代理，为需要拦截的接口生成代理对象以实现接口方法拦截功能，每当执行这 4 种接口对象的方法时，就会进入拦截方法，具体就是 InvocationHandler 的 invoke()方法，当然，只会拦截那些你指定需要拦截的方法。
2）实现 Mybatis 的 Interceptor 接口并复写 intercept()方法，然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，记住，别忘了在配置文件中配置你编写的插件。

26.二级缓存的回收策略有哪些?

eviction (收回策略)

LRU(最近最少使用的) : 移除最长时间不被使用的对象，这是默认值 。

IFO(先进先出〉: 按对象进入缓存的顺序来移除它们 。

SOFT(软引用) : 移除基于垃圾回收器状态和软引用规则的对象 。

WEAK (弱引用) : 更积极地移除基于垃圾收集器状态和弱引用规则的对象 。

27.Mybatis的Xml文件中id可以重复吗?

同一namespace下，id不可重复。不同namespace下，可以重复。

28. 和Mybatis搭配java框架中比较好用的缓存框架? 有哪些特点?

EhCache是一个纯牌的 Java进程内的缓存框架，具有快速、精干等特点。

具体来说， EhCache主要的特性如下 。

- 快速。
- 简单。
- 多种缓存策略 。
- 缓存数据有内存和磁盘两级，无须担心容量问题 。
- 缓存数据会在虚拟机重启 的过程中写入磁盘。
- 可 以通过 RMI、可插入 API 等方式进行分布式缓存。
- .具有缓存和缓存管理器的侦 昕接口。
- 支持多缓存管理器实例 以及一个实例的多个缓存区域。

参考:

- 《 MyBatis从入门到精通》
- 《深入浅出 MyBatis技术原理与实战》
- 《MyBatis技术》

Nginx

1.什么是nginx?

Nginx是一个高性能的HTTP和反向代理服务器。同时也是一个 IMAP/POP3/SMTP 代理服务器。 官方网站:http://nginx.org。

2.nginx主要特征?

处理静态文件，索引文件以及自动索引;打开文件描述符缓冲. 无缓存的反向代理加速，简单的负载均衡和容错. FastCGI，简单的负载均衡和容错.模块化的结构。包括 gzipping, byte ranges, chunked responses,以及 SSI-filter 等filter。如果由 FastCGI 或其它代理服务器处理单页中存在的多个 SSI，则这项处理可以并行 运行，而不需要相互等待。

支持 SSL 和 TLSSNI.

Nginx 它支持内核 Poll 模型，能经受高负载的考验,有报告表明能支持高达 50,000 个并发连接数。

Nginx 具有很高的稳定性。 例如当前 apache 一旦上到 200 个以上进程，web 响应速度就明显非常缓慢了。而 Nginx 采取了分阶段资源分配技术，使得它的 CPU 与内存占用率非常低。nginx 官方表示保持 10,000 个没有活动的连接，它只占 2.5M 内存，所以类似 DOS 这样的攻击对 nginx 来说基本上是毫无用处的。

Nginx 支持热部署。它的启动特别容易, 并且几乎可以做到 7*24 不间断运行，即使运 行数个月也不需要重新启动。对软件版本进行进行热升级。

Nginx 采用 master-slave 模型,能够充分利用 SMP 的优势，且能够减少工作进程在磁 盘 I/O 的阻塞延迟。当采用 select()/poll()调用时，还可以限制每个进程的连接数。

Nginx 代码质量非常高，代码很规范，手法成熟， 模块扩展也很容易。特别值得一提的是强大的 Upstream 与 Filter 链。

Nginx 采用了一些 os 提供的最新特性如对 sendfile (Linux2.2+)，accept-filter (FreeBSD4.1+)，TCP_DEFER_ACCEPT (Linux 2.4+)的支持，从而大大提高了性能。

免费开源，可以做高并发负载均衡。

3.nginx 常用命令?

启动 nginx 。
停止 nginx -s stop 或 nginx -s quit 。
重载配置 ./sbin/nginx -s reload(平滑重启) 或 service nginx reload 。
重载指定配置文件 .nginx -c /usr/local/nginx/conf/nginx.conf 。
查看 nginx 版本 nginx -v 。
检查配置文件是否正确 nginx -t 。
显示帮助信息 nginx -h 。

4.工作模式及连接数上限?

```
events {

use epoll; #epoll 是多路复用 IO(I/O Multiplexing)中的一种方 式,但是仅用于 linux2.6
以上内核,可以大大提高 nginx 的性能

worker_connections 1024;#单个后台 worker process 进程的最大并发链接数

# multi_accept on;
}
```

5.nginx负载均衡几种算法?

5种。

1.轮询模式（默认）
每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器down掉，能自动剔除。
2.权重模式
指定轮询几率，weight和访问比率成正比，用于后端服务器性能不均的情况
3.IP_hash模式 （IP散列）
每个请求按访问ip的hash结果分配，这样每个访客固定访问一个后端服务器，可以解决session的问题。
4.url_hash模式
5.fair模式
按后端服务器的响应时间来分配请求，响应时间短的优先分配。

6.nginx有几种进程模型?

分为master-worker模式和单进程模式。在master-worker模式下，有一个master进程和至少一个的worker进程，单进程模式顾名思义只有一个进程。

7.如何定义错误提示页面?

定义错误提示页面

```
error_page 500 502 503 504 /50x.html;

location = /50x.html { root /root;

}
```

8.如何精准匹配路径?

location =开头表示精准匹配

```
location = /get {
#规则 A }
```

9.路径匹配优先级?

多个 location 配置的情况下匹配顺序为
首先匹配 =，其次匹配^~, 其次是按文件中顺序的正则匹配，最后是交给 / 通用匹配。当 有匹配成功时候，停止匹配，按当前匹配规则处理请求。

10.如何把请求转发给后端应用服务器?

```
location = / {

proxy_pass http://tomcat:8080/index

}
```

11.如何根据文件类型设置过期时间?

```
location ~* \.(js|css|jpg|jpeg|gif|png|swf)$ {
if (-f $request_filename) {
expires 1h;
break;
}
}
```

12.禁止访问某个目录?

```
location ^~/path/ {
    deny all;
}
```

13.nginx负载均衡实现过程?

首先在 http 模块中配置使用 upstream 模块定义后台的 webserver 的池子，名为 proxy-web，在池子中我们可以添加多台后台 webserver，其中状态 检查、调度算法都是在池子中配置;然后在 serverr 模块中定义虚拟主机，但是这个虚拟主 机不指定自己的 web 目录站点，它将使用 location 匹配 url 然后转发到上面定义好的 web 池子中，最后根据调度策略再转发到后台 web server 上 。

14.负载均衡配置?

```
Upstream proxy_nginx {

server 192.168.0.254  weight=1max_fails=2 fail_timeout=10s ;
server 192.168.0.253 weight=2 max_fails=2fail_timeout=10s;
server192.168.0.252 backup; server192.168.0.251 down;

}

server{
 listen 80;
 server_name xiaoka.com;

location / {

 proxy_pass http:// proxy_nginx;
 proxy_set_header Host
 proxy_set_header X-Real-IP
 proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

}
```

15.设置超时时间?

```
http {
 ……..
 keepalive_timeout 60; ###设置客户端连接保持会话的超时时间，超过这个时间，服务器会关闭该连接。 tcp_nodelay on;
 \####打开 tcp_nodelay，在包含了 keepalive 参数才有效
 client_header_timeout 15; ####设置客户端请求头读取超时时间，如果超过这个时间，客户端还没有发送任何数据， Nginx 将返回“Request time out(408)”错误
 client_body_timeout 15;

\####设置客户端请求主体读取超时时间，如果超过这个时间，客户端还没有发送任何数据， Nginx 将返回“Request time out(408)”错误
 send_timeout 15; ####指定响应客户端的超时时间。这个超过仅限于两个连接活动之间的时间，如果超过这 个时间，客户端没有任何活动，Nginx 将会关闭连接。

…… }
```

16.开启压缩功能好处？坏处?

好处：压缩是可以节省带宽，提高传输效率

坏处：但是由于是在服务器上进行压缩，会消耗服务器起源

参考:

- 《Nginx从入门到精通》
- 《Nginx高性能Web服务器详解》
- 《深入理解nginx》

Redis

1.Redis是什么?

Redis是一个开放源代码（BSD许可）的内存中数据结构存储，可用作数据库，缓存和消息代理，是一个基于键值对的NoSQl数据库。

2.Redis特性?

- 速度快
- 基于键值对的数据结构服务器
- 丰富的功能、丰富的数据结构
- 简单稳定
- 客户端语言多
- 持久化
- 主从复制
- 高可以 & 分布式

3.Redis合适的应用场景?

- 缓存
- 排行榜
- 计数器
- 分布式会话
- 分布式锁
- 社交网络
- 最新列表
- 消息系统

4.除了Redis你还知道哪些NoSQL数据库?

MongoDB、MemcacheDB、Cassandra、CouchDB、Hypertable、Leveldb。

5.Redis和Memcache区别?

支持的存储类型不同，memcached只支持简单的k/v结构。redis支持更多类型的存储结构类型(详见问题6)。

memcached数据不可恢复，redis则可以把数据持久化到磁盘上。

新版本的redis直接自己构建了VM 机制 ，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。

redis当物理内存用完时，可以将很久没用到的value交换到磁盘。

6.Redis的有几种数据类型?

基础：字符串（String）、哈希（hash)、列表（list)、集合(set)、有序集合(zset)。

还有HyperLogLog、流、地理坐标等。

7.Redis有哪些高级功能?

消息队列、自动过期删除、事务、数据持久化、分布式锁、附近的人、慢查询分析、Sentinel 和集群等多项功能。

8.安装过Redis吗,简单说下步骤?

1. 下载Redis指定版本源码安装包压缩到当前目录。
2. 解压缩Redis源码安装包。
3. 建立一个redis目录软链接，指向解压包。
4. 进入redis目录
5. 编译
6. 安装

   对于使用docker的童靴来说就比较容易了。

   docker pull redis

9.redis几个比较主要的可执行文件？分别是?

可执行文件	作　　用
redis-server	启动 Redis
redis-cli	Redis 命令行客户端
redis-benchmark	Redis 基准测试工具
redis-check-aof	Redis AOF 持久化文件检测和修复工具
redis-check-dump	Redis RDB 持久化文件检测和修复工具
redis-sentinel	启动 Redis Sentinel

10.启动Redis的几种方式?

1.默认配置：

./redis-server

2.运行启动: redis-server 加上要修改配置名和值（可以是多对），没有配置的将使用默认配置。

例如: redis-server ————port 7359

3.指定配置文件启动:

./redis-server /opt/redis/redis.conf

11.Redis配置需要自己写？如何配置?

redis目录下有一个redis.conf的模板配置。所以只需要复制模板配置然后修改即可。

一般来说大部分生产环境都会用指定配置文件的方式启动redis。

12.Redis客户端命令执行的方式?

1.交互方式:

```
redis-cli -h 127.0.0.1 -p 6379
```

连接到redis后，后面执行的命令就可以通过交互方式实现了。

2.命令行方式：

```
redis-cli -h 127.0.0.1 -p 6379 get value
```

13.如何停止redis服务?

Kill -9 pid (粗暴，请不要使用,数据不仅不会持久化，还会造成缓存区等资源不能被优雅关闭)

可以用redis 的shutdown 命令，可以选择是否在关闭前持久化数据。

```
redis-cli shutdown nosave|save
```

14.如何查看当前键是否存在?

exists key

15.如何删除数据?

del key

16.redis为什么快？单线程?

- redis使用了单线程架构和I/O多路复用模型模型。
- 纯内存访问。
- 由于是单线程避免了线程上下文切换带来的资源消耗。

17.字符串最大不能超过多少?

512MB

18.redis默认分多少个数据库?

16

19.redis持久化的几种方式?

RDB、AOF、混合持久化。

20.RDB持久化?

RDB（Redis DataBase)持久化是把当前进程数据生成快照保存到硬盘的过程。

Tips:是以二进制的方式写入磁盘。

21.RDB的持久化是如何触发的?

手动触发:

save: 阻塞当前Redis服务器，直到RDB过程完成为止，如果数据比较大的话，会造成长时间的阻塞，线上不建议。

bgsave:redis进程执行 fork操作创作子进程，持久化由子进程负责，完成后自动结束，阻塞只发生在fork阶段，一半时间很短。

自动触发:

save xsecends n:

表示在x秒内，至少有n个键发生变化，就会触发RDB持久化。也就是说满足了条件就会触发持久化。

flushall :

主从同步触发

22.RDB的优点?

- rdb是一个紧凑的二进制文件，代表Redis在某个时间点上的数据快照。
- 适合于备份，全量复制的场景，对于灾难恢复非常有用。
- Redis加载RDB恢复数据的速度远快于AOF方式。

23.RDB的缺点?

- RDB没法做到实时的持久化。中途意外终止，会丢失一段时间内的数据。
- RDB需要fork()创建子进程，属于重量级操作，可能导致Redis卡顿若干秒。

24.如何禁用持久化?

一般来说生成环境不会用到，了解一下也有好处的。

```
config set save ""
```

25.AOF持久化?

AOF(append only file)为了解决rdb不能实时持久化的问题，aof来搞定。以独立的日志方式记录把每次命令记录到aof文件中。

26.如何查询AOF是否开启?

```
config get appendonly
```

27.如何开启AOF?

命令行方式： 实时生效，但重启后失效。

```
config set appendonly
```

配置文件：需要重启生效，重启后依然生效。

```
appendonly yes
```

28.AOF工作流程?

1.所有写入命令追加到aof_buf缓冲区。

2.AOF缓冲区根据对应的策略向硬盘做同步操作。

3.随着AOF文件越来越大，需要定期对AOF文件进行重写，达到压缩的目的。

4.当redis服务器重启时，可以加载AOF文件进行数据恢复。

29.为什么AOF要先把命令追加到缓存区(aof_buf)中?

Redis使用单线程响应命令，如果每次写入文件命令都直接追加到硬盘，性能就会取决于硬盘的负载。如果使用缓冲区，redis提供多种缓冲区策略，在性能和安全性方面做出平衡。

30.AOF持久化如何触发的?

自动触发：满足设置的策略和满足重写触发。

策略：(在配置文件中配置)

![image-20200427101221526](https://gitee.com/yizhibuerdai/Imagetools/raw/master/images/image-20200427101221526.png

手动触发：（执行命令）

```
bgrewriteaof
```

31.AOF优点?

- AOF提供了3种保存策略：每秒保存、跟系统策略、每次操作保存。实时性比较高，一般来说会选择每秒保存，因此意外发生时顶多失去一秒的数据。
- 文件追加写形式，所以文件很少有损坏问题，如最后意外发生少写数据，可通过redis-check-aof工具修复。
- AOF由于是文本形式，直接采用协议格式，避免二次处理开销，另外对于修改也比较灵活。

32.AOF缺点?

- AOF文件要比RDB文件大。
- AOF冷备没RDB迅速。
- 由于执行频率比较高，所以负载高时，性能没有RDB好。

33.混合持久化？优缺点?

一般来说我们的线上都会采取混合持久化。redis4.0以后添加了新的混合持久化方式。

优点：

- 在快速加载的同时，避免了丢失过更多的数据。

缺点：

- 由于混合了两种格式，所以可读性差。
- 兼容性，需要4.0以后才支持。

34.Redis的Java客户端官方推荐？实际选择?

官方推荐的有3种：Jedis、Redisson和lettuce。

一般来说用的比较多的有:Jedis | Redisson。

Jedis：更轻量、简介、不支持读写分离需要我们来实现，文档比较少。API提供了比较全面的Redis命令的支持。

Redisson：基于Netty实现，性能高，支持异步请求。提供了很多分布式相关操作服务。高级功能能比较多，文档也比较丰富，但实用上复杂度也相对高。和Jedis相比，功能较为简单，不支持字符串操作，不支持排序、事务、管道、分区等Redis特性。

35.Redis事务?

事务提供了一种将多个命令请求打包，一次性、按顺序的执行多个命令的机制。并且在事务执行期间，服务器不会中断事务而改去执行其他客户端命令请求，它会

36.Redis事务开始到结束的几个阶段?

- 开启事务
- 命令入队
- 执行事务/放弃事务

37.Redis中key的过期操作?

设置key的生存时间为n秒

```
expire key nseconds
```

设置key的生存时间为nmilliseconds

```
pxpire key milliseconds
```

设置过期时间为timestamp所指定的秒数时间戳

```
expireat key timespamp
```

设置过期时间为timestamp毫秒级时间戳

```
pexpireat key millisecondsTimestamp
```

38.Redis过期键删除策略?

定时删除：在设置的过期时间同时，创建一个定时器在键的过期时间来临时，立即执行队键的操作删除。

惰性删除：放任过期键不管，但每次从键空间中获取键时，都检查取得的键是否过期，如果过期就删除，如果没有就返回该键。

定期删除：每隔一段时间执行一次删除过期键操作，并通过先吃删除操作执行的时长和频率来减少删除操作对cpu时间的影响。

39.Pipeline是什么？为什么要它?

命令批处理技术，对命令进行组装，然后一次性执行多个命令。

可以有效的节省RTT(Round Trip Time 往返时间)。

经过测试验证：

- pipeline执行速度一般比逐条执行快。
- 客户端和服务的网络延越大，pipeline效果越明显。

40.如何获取当前最大内存？如何动态设置?

获取最大内存:

```
config get maxmemory
```

设置最大内存：

命令设置:

```
config set maxmemory 1GB
```

41.Redis内存溢出控制?

当Redis所用内存达到maxmemory上限时，会出发相应的溢出策略。

42.Redis内存溢出策略?

1.noeviction(默认策略):拒绝所有写入操作并返回客户端错误信息（error) OOM command not allowed when used memory,只响应读操作。

2. volatile-lru:根据LRU算法删除设置了超时属性（expire)的键，直到腾出足够空间为止。如果没有可删除的键对象，回退到noeviction策略。
3. allkeys-lru:根据LRU算法删除键，不管数据有没有设置超时属性， 直到腾出足够空间为止。
4. allkeys-random:随机删除所有键，直到腾出足够空间为止。
5. volatile-random:随机删除过期键，直到腾出足够空间为止。
6. volatile-tth根据键值对象的ttl属性，删除最近将要过期数据。如果没有，回退到noeviction策略。

43.Redis高可用方案?

Redis Sentinel(哨兵)能自动完成故障发现和转移。

44.Redis集群方案?

Twemproxy、Redis Cluster、Codis。

45.Redis Cluster槽范围?

0~16383

46.Redis锁实现思路?

setnx (set if not exists),如果创建成功则表示获取到锁。

setnx lock true 创建锁

del lock 释放锁

如果中途崩溃，无法释放锁?

此时需要考虑到超时时间的问题。比如 :expire lock 300

由于命令是非原子的，所以还是会死锁,如何解决?

Redis 支持 set 并设置超时时间的功能。

比如: set lock true ex 30 nx

47.什么是布隆过滤器?

是1970年由布隆提出的。它实际上是一个很长的二进制向量和一系列随机映射函数。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都比一般的算法要好的多，缺点是有一定的误识别率和删除困难。

Tips:当判断一定存在时，可能会误判，当判断不存在时，就一定不存在。

48.什么是缓存穿透？处理问题？

缓存穿透：缓存层不命中，存储层不命中。

处理方式1:缓存空对象，不过此时会占用更多内存空间，所以根据大家业务特性去设置超时时间来控制内存占用的问题。

处理方式2:布隆过滤器。

49.什么是缓存预热？

就是系统上线后，提前将相关数据加载到缓存系统，避免用户先查库，然后在缓存。

50.什么是缓存雪崩？处理问题？

缓存雪崩：由于缓存层承载着大量请求，有效的保护了存储层，但如果存储层由于某些原因不能提供服务，存储层调用暴增，造成存储层宕机。

处理：

- 保证缓存层服务高可用性。
- 对缓存系统做实时监控，报警等。
- 依赖隔离组件为后端限流并降级。
- 做好持久化，以便数据的快速恢复。

参考：

- 《Redis深度历险:核心原理和应用实践》
- 《Redis开发与运维》
- 《Redis设计与实现》
- https://redis.io/
- 百度百科

Dubbo

1.什么是Dubbo?

Dubbo是基于Java的高性能轻量级的RPC分布式服务框架，现已成为 Apache 基金会孵化项目。

官网：http://dubbo.apache.org/en-us/

2.为什么要使用Dubbo?

背景:

随着互联网的快速发展，Web应用程序的规模不断扩大，最后我们发现传统的垂直体系结构（整体式）已无法解决。分布式服务体系结构和流计算体系结构势在必行，迫切需要一个治理系统来确保体系结构的有序发展。

- 开源免费
- 一些核心业务被提取并作为独立的服务提供服务，逐渐形成一个稳定的服务中心，这样前端应用程序就可以更好地响应变化多端的市场需求
- 分布式框架能承受更大规模的流量
- 内部基于netty性能高

3.Dubbo提供了哪3个关键功能?

基于接口的远程调用

容错和负载均衡

自动服务注册和发现

4.你知道哪些机构在用Dubbo吗?

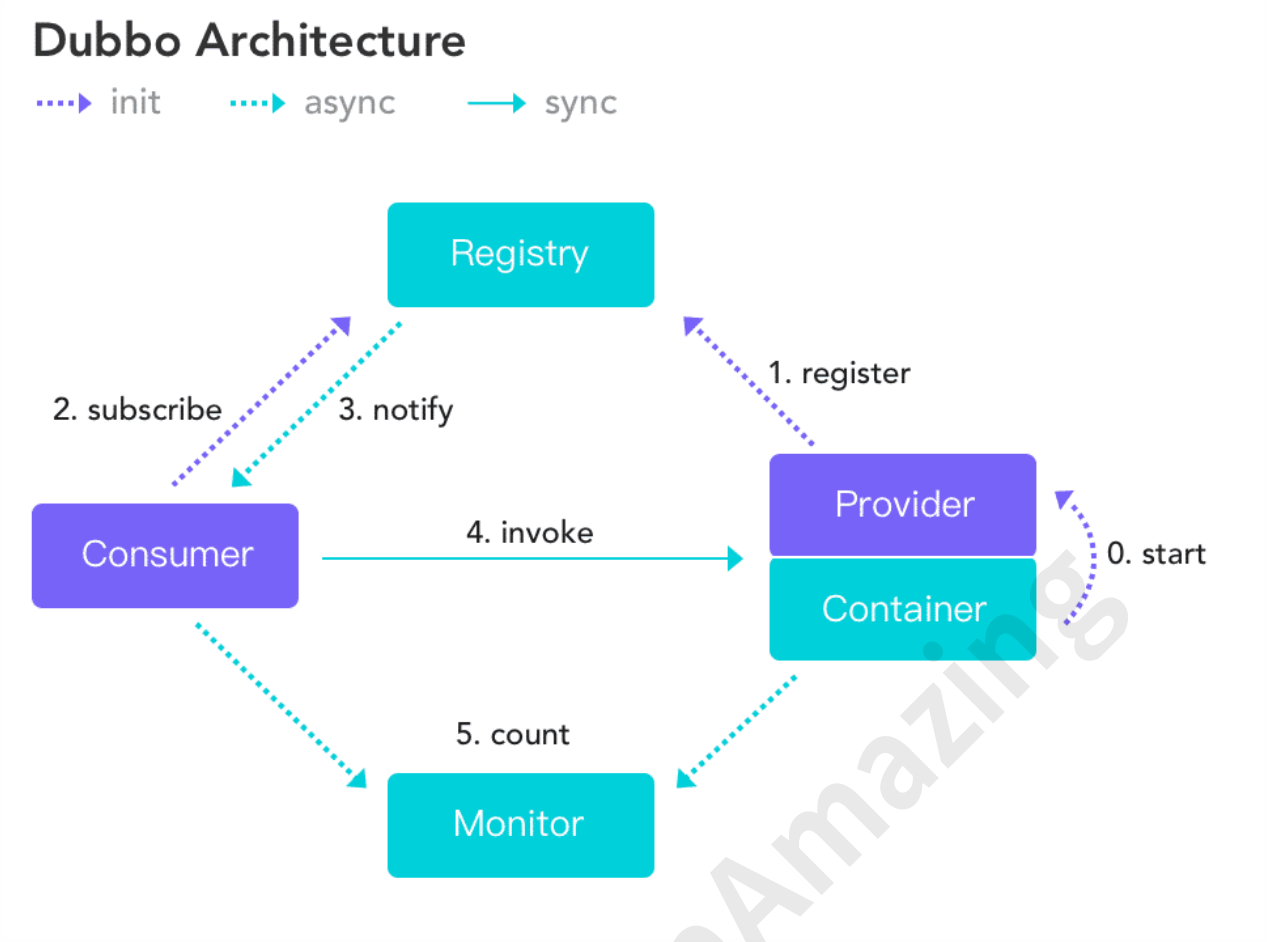
5.Dubbo服务的关键节点有哪些?

节点	角色说明
`Provider`	暴露服务的服务提供方
`Consumer`	调用远程服务的服务消费方
`Registry`	服务注册与发现的注册中心
`Monitor`	统计服务的调用次数和调用时间的监控中心
`Container`	服务运行容器

6.说一下Dubbo服务注册流程?

1. 服务容器负责启动，加载，运行服务提供者。
2. 服务提供者在启动时，向注册中心注册自己提供的服务。
3. 服务消费者在启动时，向注册中心订阅自己所需的服务。
4. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
5. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。
6. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

7.能画一下服务注册流程图吗?



8.Dubbo架构的特点?

连通性、健壮性、伸缩性、以及向未来架构的升级性。

9.对jdk的最小版本需求?

jdk1.6+

10.注册中心的选择?

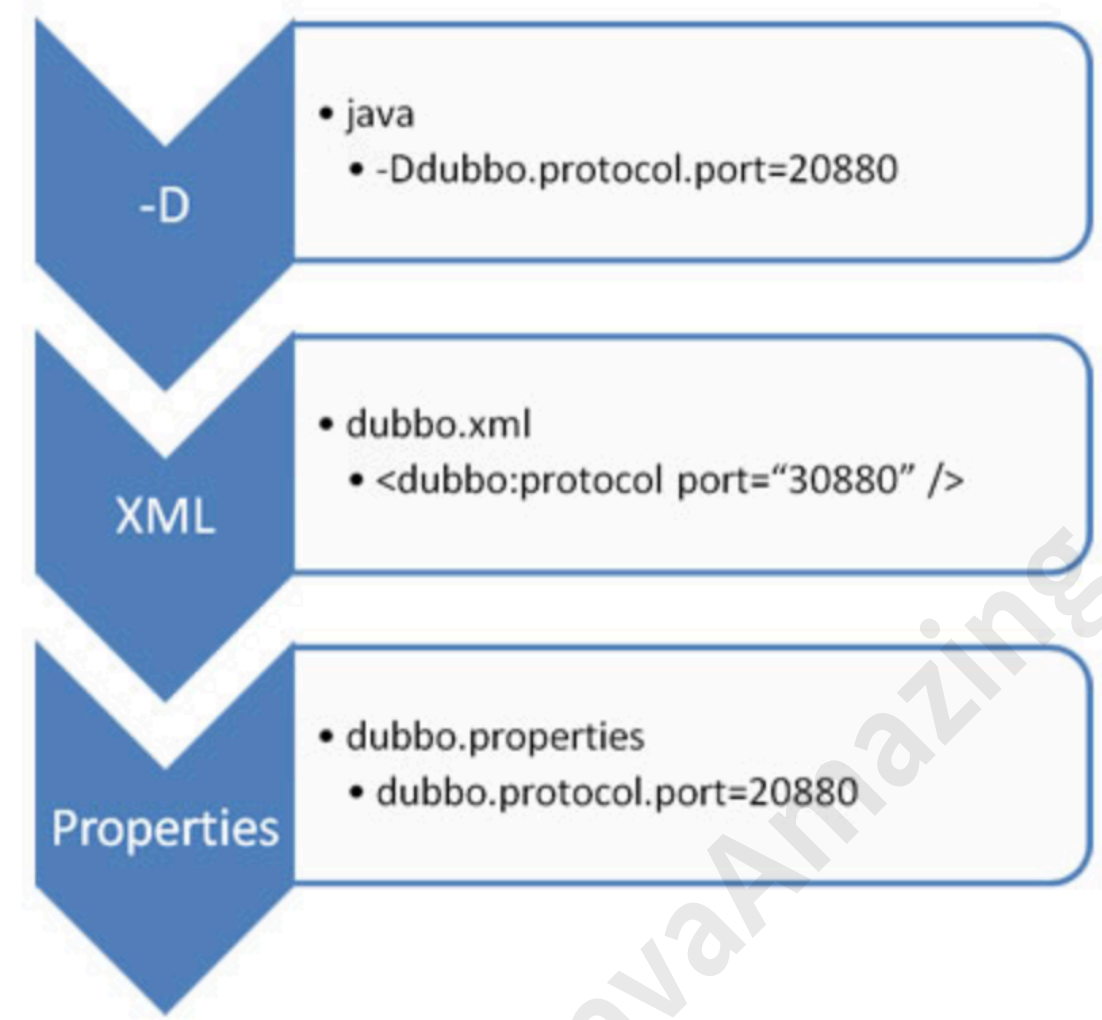
一般来说选中Zookeeper更稳定更合适。

除了Zookeeper还有Redis注册中心、Multicast注册中心、Simple注册中心。

11.Dubbo的核心配置? 用途?

标签	用途	解释
`<dubbo:service/>`	服务配置	用于暴露一个服务，定义服务的元信息，一个服务可以用多个协议暴露，一个服务也可以注册到多个注册中心
`<dubbo:reference/>` [2]	引用配置	用于创建一个远程服务代理，一个引用可以指向多个注册中心
`<dubbo:protocol/>`	协议配置	用于配置提供服务的协议信息，协议由提供方指定，消费方被动接受
`<dubbo:application/>`	应用配置	用于配置当前应用信息，不管该应用是提供者还是消费者
`<dubbo:module/>`	模块配置	用于配置当前模块信息，可选
`<dubbo:registry/>`	注册中心配置	用于配置连接注册中心相关信息
`<dubbo:monitor/>`	监控中心配置	用于配置连接监控中心相关信息，可选
`<dubbo:provider/>`	提供方配置	当 ProtocolConfig 和 ServiceConfig 某属性没有配置时，采用此缺省值，可选
`<dubbo:consumer/>`	消费方配置	当 ReferenceConfig 某属性没有配置时，采用此缺省值，可选
`<dubbo:method/>`	方法配置	用于 ServiceConfig 和 ReferenceConfig 指定方法级的配置信息
`<dubbo:argument/>`	参数配置	用于指定方法参数配置

12.配置优先级规则?



优先级从高到低：

- JVM -D参数，当你部署或者启动应用时，它可以轻易地重写配置，比如，改变dubbo协议端口；
- XML, XML中的当前配置会重写dubbo.properties中的；
- Properties，默认配置，仅仅作用于以上两者没有配置时。

13.如何用代码方式绕过注册中心点对点直连?

```
…

ReferenceConfig<XxxService> reference = new ReferenceConfig<XxxService>(); //
此实例很重，封装了与注册中心的连接以及与提供者的连接，请自行缓存，否则可能造成内存和连接泄漏
// 如果点对点直连，可以用reference.setUrl()指定目标地址，设置url后将绕过注册中心，
// 其中，协议对应provider.setProtocol()的值，端口对应provider.setPort()的值，
// 路径对应service.setPath()的值，如果未设置path，缺省path为接口名
reference.setUrl("dubbo://10.20.130.230:20880/com.xxx.XxxService");

…
```

14.Dubbo配置来源有几种？分别是?

4种

- JVM System Properties，-D参数
- Externalized Configuration，外部化配置
- ServiceConfig、ReferenceConfig等编程接口采集的配置
- 本地配置文件dubbo.properties

15.如何禁用某个服务的启动检查?

```
<dubbo:reference interface = "com.foo.BarService" check = "false" />
```

16.Dubbo 负载均衡策略？默认是?

- 随机负载平衡(默认)
- RoundRobin负载平衡
- 最小活动负载平衡
- 一致的哈希负载平衡

17.上线兼容老版本?

多版本号(version)

18.开发测试环境，想绕过注册中心如何配置?

- xml

```
 <dubbo:reference id="xxxService" interface="com.alibaba.xxx.XxxService"
url="dubbo://localhost:20890" />
```

- -D

```
java -Dcom.alibaba.xxx.XxxService=dubbo://localhost:20890
```

- .properties

```
java -Ddubbo.resolve.file=xxx.properties
```

```
com.alibaba.xxx.XxxService=dubbo://localhost:20890
```

19.集群容错几种方法?

集群容错方案	说明
Failover Cluster	失败自动切换，自动重试其它服务器（默认）
Failfast Cluster	快速失败，立即报错，只发起一次调用
Failsafe Cluster	失败安全，出现异常时，直接忽略
Failback Cluster	失败自动恢复，记录失败请求，定时重发
Forking Cluster	并行调用多个服务器，只要一个成功即返回

20.Dubbo有几种配置方式?

1. Spring
2. Java API

21.Dubbo有哪些协议？推荐?

- dubbo://(推荐)
- rmi://
- hessian://
- http://
- webservice://
- thrift://
- memcached://
- redis://
- rest://

22.Dubbo使用什么通信框架?

dubbo使用netty。

23.dubbo协议默认端口号？http协议默认端口？hessian?rmi?

- dubbo:20880
- http:80
- hessian:80
- rmi:80

24.Dubbo默认序列化框架?其他的你还知道?

- dubbo协议缺省为hessian2

- rmi协议缺省为java
- http协议缺省为json

25.一个服务有多重实现时，如何处理?

可以用group分组，服务提供方和消费放都指定同一个group。

26.Dubbo服务调用默认是阻塞的？还有其他的?

默认是同步等待结果阻塞的，同时也支持异步调用。

Dubbo 是基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对多线程开销较小，异步调用会返回一个 Future 对象。

27.Dubbo服务追踪解决方案?

- Zipkin
- Pinpoint
- SkyWalking

28.Dubbo不维护了吗？Dubbo和Dubbox有什么区别?

现在进入了Apache,由apache维护。

Dubbox是当当的扩展项目。

29.Dubbox有什么新功能?

- 支持REST风格远程调用（HTTP + JSON/XML)
- 支持基于Kryo和FST的Java高效序列化实现
- 支持基于嵌入式Tomcat的HTTP remoting体系
- 升级Spring
- 升级ZooKeeper客户端

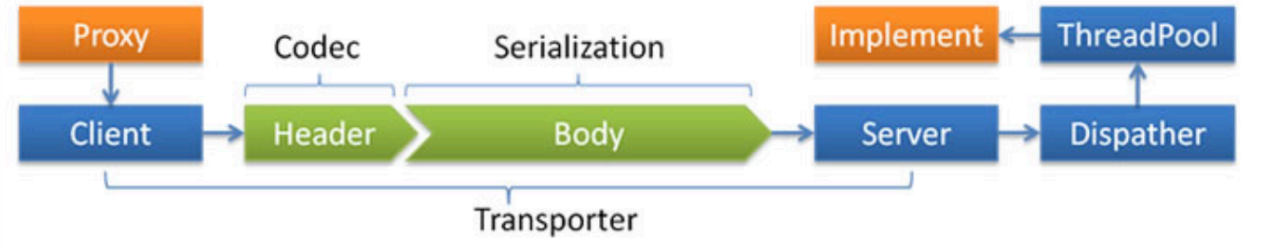
30.io线程池大小默认?

cpu个数 + 1

31.dubbo://协议适合什么样的服务调用?

采用单一长链接和NIO异步通讯，适用于小数量大并发的服务调用，以及服务消费者机器数远大于服务提供者机器数的情况。

不适合传送大数据量的服务，比如传文件，传视频等，除非请求量很低。



32.自动剔除服务什么原理?

zookeeper临时节点，会话保持原理。

33.从 2.0.5 版本开始，dubbo支持通过x命令来进行服务治理?

telnet

34.如何用命令查看服务列表?

```
telnet localhost 20880
```

进入命令行。然后执行 ls相关命令:

- ls: 显示服务列表
- ls -l: 显示服务详细信息列表
- ls XxxService: 显示服务的方法列表
- ls -l XxxService: 显示服务的方法详细信息列表

35.Dubbo框架设计是怎样的?

各层说明:

- **config 配置层**：对外配置接口，以 ServiceConfig, ReferenceConfig 为中心，可以直接初始化配置类，也可以通过 spring 解析配置生成配置类
- **proxy 服务代理层**：服务接口透明代理，生成服务的客户端 Stub 和服务器端 Skeleton, 以 ServiceProxy 为中心，扩展接口为 ProxyFactory
- **registry 注册中心层**：封装服务地址的注册与发现，以服务 URL 为中心，扩展接口为 RegistryFactory, Registry, RegistryService
- **cluster 路由层**：封装多个提供者的路由及负载均衡，并桥接注册中心，以 Invoker 为中心，扩展接口为 Cluster, Directory, Router, LoadBalance
- **monitor 监控层**：RPC 调用次数和调用时间监控，以 Statistics 为中心，扩展接口为 MonitorFactory, Monitor, MonitorService
- **protocol 远程调用层**：封装 RPC 调用，以 Invocation, Result 为中心，扩展接口为 Protocol, Invoker, Exporter
- **exchange 信息交换层**：封装请求响应模式，同步转异步，以 Request, Response 为中心，扩展接口为 Exchanger, ExchangeChannel, ExchangeClient, ExchangeServer
- **transport 网络传输层**：抽象 mina 和 netty 为统一接口，以 Message 为中心，扩展接口为 Channel, Transporter, Client, Server, Codec
- **serialize 数据序列化层**：可复用的一些工具，扩展接口为 Serialization, ObjectInput, ObjectOutput, ThreadPool

36.你读过Dubbo的源码吗?

这个问题其实面试中如果问dubbo的话，基本就会带这个问题。有时间的话，大家可以下载源码，读一读，如果大家有兴趣的话，我会出后续文章。

参考：http://dubbo.apache.org/en-us/

SpringBoot

1.什么是SpringBoot?

通过Spring Boot，可以轻松地创建独立的，基于生产级别的Spring的应用程序，您可以“运行”它们。大多数Spring Boot应用程序需要最少的Spring配置。

2.SpringBoot的特征?

- 创建独立的Spring应用程序
- 直接嵌入Tomcat，Jetty或Undertow（无需部署WAR文件）
- 提供固化的“starter”依赖项，以简化构建配置
- 尽可能自动配置Spring和3rd Party库
- 提供可用于生产的功能，例如指标，运行状况检查和外部化配置
- 完全没有代码生成，也不需要XML配置

3.如何快速构建一个SpringBoot项目?

- 通过Web界面使用。http://start.spring.io
- 通过Spring Tool Suite使用。
- 通过IntelliJ IDEA使用。
- 使用Spring Boot CLI使用。

4.SpringBoot启动类注解?它是由哪些注解组成?

@SpringBootApplication

- @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。
- @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })
- @ComponentScan:Spring组件扫描

5.什么是yaml?

YAML（/ˈjæməl/，尾音类似camel骆驼）是一个可读性高，用来表达数据序列化的格式。YAML参考了其他多种语言，包括：C语言、Python、Perl。更具有结构性。

6.SpringBoot支持配置文件的格式?

1.properties

```
java.xiaokaxiu.name = xiaoka
```

2.yml

```
java:
   xiaokaxiu:
        name: xiaoka
```

7.SpringBoot启动方式?

1. main方法
2. 命令行 java -jar 的方式
3. mvn/gradle

8.SpringBoot需要独立的容器运行?

不需要，内置了 Tomcat/Jetty。

9.SpringBoot配置途径?

1. 命令行参数
2. java:comp/env里的JNDI属性
3. JVM系统属性
4. 操作系统环境变量
5. 随机生成的带random.*前缀的属性(在设置其他属性时，可以引用它们，比如${random. long})
6. 应用程序以外的application.properties或者appliaction.yml文件
7. 打包在应用程序内的application.properties或者appliaction.yml文件
8. 通过@PropertySource标注的属性源
9. 默认属性

tips:这个列表按照优先级排序，也就是说，任何在高优先级属性源里设置的属性都会覆盖低优先级的相同属性。

10.application.properties和application.yml文件可放位置?优先级?

1. 外置，在相对于应用程序运行目录的/config子目录里。
2. 外置，在应用程序运行的目录里。
3. 内置，在config包内。
4. 内置，在Classpath根目录。

   这个列表按照优先级排序,优先级高的会覆盖优先级低的。

   当然我们可以自己指定文件的位置来加载配置文件。

```
java -jar xiaoka.jar ——spring.config.location=/home/application.yml
```

11.SpringBoot自动配置原理?

@EnableAutoConfiguration (开启自动配置)
该注解引入了AutoConfigurationImportSelector，该类中的方法会扫描所有存在META-INF/spring.factories的jar包。

12.SpringBoot热部署方式?

- spring-boot-devtools
- Spring Loaded
- Jrebel
- 模版热部署

13.bootstrap.yml 和application.yml?

bootstrap.yml 优先于application.yml

14.SpringBoot如何修改端口号?

yml中:

```
server :
  port : 8888
```

properties:

```
server.port = 8888
```

命令1:

```
java -jar xiaoka.jar —— server.port=8888
```

命令2:

```
java - Dserver.port=8888 -jar xiaoka.jar
```

15.开启SpringBoot特性的几种方式?

1. 继承spring-boot-starter-parent项目
2. 导入spring-boot-dependencies项目依赖

16.SpringBoot如何兼容Spring项目?

在启动类加:

@ImportResource(locations = {"classpath:spring.xml"})

17.SpringBoot配置监控?

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

18.获得Bean装配报告信息访问哪个端点?

/beans 端点

19.关闭应用程序访问哪个端点?

/shutdown

该端点默认是关闭的，如果开启，需要如下设置。

```
endpoints:
     shutdown:
       enabled: true
```

或者properties格式也是可以的。

20.查看发布应用信息访问哪个端点?

/info

21.针对请求访问的几个组合注解?

@PatchMapping

@PostMapping

@GetMapping

@PutMapping

@DeleteMapping

22.SpringBoot 中的starter?

可以理解成对依赖的一种合成，starter会把一个或一套功能相关依赖都包含进来，避免了自己去依赖费事，还有各种包的冲突问题。大大的提升了开发效率。

并且相关配置会有一个默认值，如果我们自己去配置，就会覆盖默认值。

23.SpringBoot集成Mybatis?

mybatis-spring-boot-starter

24.什么是SpringProfiles?

一般来说我们从开发到生产，经过开发(dev)、测试（test）、上线(prod)。不同的时刻我们会用不同的配置。Spring Profiles 允许用户根据配置文件（dev，test，prod 等）来注册 bean。它们可以让我们自己选择什么时候用什么配置。

25.不同的环境的配置文件?

可以是 application-{profile}.properties/yml ，但默认是启动主配置文件application.properties,一般来说我们的不同环境配置如下。

- `application.properties`：主配置文件
- `application-dev.properties`：开发环境配置文件
- `application-test.properties`：测试环境配置文件
- `application.prop-properties`：生产环境配置文件

26.如何激活某个环境的配置?

比如我们激活开发环境。

yml:

```yaml
spring:
  profiles:
    active: dev
```

properties:

```properties
spring.profiles.active=dev
```

命令行:

```
java -jar xiaoka-v1.0.jar ——spring.profiles.active=dev
```

27.编写测试用例的注解?

@SpringBootTest

28.SpringBoot异常处理相关注解?

@ControllerAdvice

@ExceptionHandler

29.SpringBoot 1.x 和 2.x区别?·····

1. SpringBoot 2基于Spring5和JDK8，Spring 1x用的是低版本。
2. 配置变更，参数名等。
3. SpringBoot2相关的插件最低版本很多都比原来高
4. 2.x配置中的中文可以直接读取，不用转码
5. Actuator的变化
6. CacheManager 的变化

30.SpringBoot读取配置相关注解有?

- @PropertySource
- @Value
- @Environment
- @ConfigurationProperties

参考:

- 《SpringBoot实战（第4版）》
- 《Spring Boot编程思想》
- 《深入浅出Spring Boot 2.x》
- https://spring.io/projects/spring-boot
- 百度百科

Kafka

1.什么是kafka?

Apache Kafka是由Apache开发的一种发布订阅消息系统。

2.kafka的3个关键功能?

- 发布和订阅记录流，类似于消息队列或企业消息传递系统。
- 以容错的持久方式存储记录流。
- 处理记录流。

3.kafka通常用于两大类应用?

- 建立实时流数据管道，以可靠地在系统或应用程序之间获取数据
- 构建实时流应用程序，以转换或响应数据流

4.kafka特性?

1. 消息持久化
2. 高吞吐量
3. 扩展性
4. 多客户端支持
5. Kafka Streams

6. 安全机制
7. 数据备份
8. 轻量级
9. 消息压缩

5.kafka的5个核心Api?

- Producer API
- Consumer API
- Streams API
- Connector API
- Admin API

6.什么是Broker（代理）?

Kafka集群中，一个kafka实例被称为一个代理(Broker)节点。

7.什么是Producer（生产者）?

消息的生产者被称为Producer。

Producer将消息发送到集群指定的主题中存储，同时也自定义算法决定将消息记录发送到哪个分区?

8.什么是Consumer（消费者）?

消息的消费者，从kafka集群中指定的主题读取消息。

9.什么是Topic（主题）?

主题，kafka通过不同的主题却分不同的业务类型的消息记录。

10.什么是Partition（分区）?

每一个Topic可以有一个或者多个分区(Partition)。

11.分区和代理节点的关系?

一个分区只对应一个Broker,一个Broker可以管理多个分区。

12.什么是副本(Replication)?

每个主题在创建时会要求制定它的副本数（默认1）。

13.什么是记录(Record)?

实际写入到kafka集群并且可以被消费者读取的数据。

每条记录包含一个键、值和时间戳。

14.kafka适合哪些场景?

日志收集、消息系统、活动追踪、运营指标、流式处理、时间源等。

15.kafka磁盘选用上?

SSD的性能比普通的磁盘好，这个大家都知道，实际中我们用普通磁盘即可。它使用的方式多是顺序读写操作，一定程度上规避了机械磁盘最大的劣势，即随机读写操作慢，因此SSD的没有太大优势。

16.使用RAID的优势?

- 提供冗余的磁盘存储空间
- 提供负载均衡

17.磁盘容量规划需要考虑到几个因素?

- 新增消息数
- 消息留存时间
- 平均消息大小
- 备份数
- 是否启用压缩

18.Broker使用单个？多个文件目录路径参数?

log.dirs 多个

log.dir 单个

19.一般来说选择哪个参数配置路径？好处?

log.dirs

好处:

提升读写性能，多块物理磁盘同时读写高吞吐。

故障转移。一块磁盘挂了转移到另一个上。

20.自动创建主题的相关参数是?

auto.create.topics.enable

21.解决kafka消息丢失问题?

- 不要使用 producer.send(msg)，而要使用 producer.send(msg, callback)。
- 设置 acks = all。
- 设置 retries 为一个较大的值。
- 设置 unclean.leader.election.enable = false。
- 设置 replication.factor >= 3。
- 设置 min.insync.replicas > 1。
- 确保 replication.factor > min.insync.replicas。
- 确保消息消费完成再提交。

22.如何自定分区策略?

显式地配置生产者端的参数partitioner.class

参数为你实现类的 全限定类名，一般来说实现partition方法即可。

23.kafka压缩消息可能发生的地方?

Producer 、Broker。

24.kafka消息重复问题?

做好幂等。

数据库方面可以（唯一键和主键）避免重复。

在业务上做控制。

25.你知道的kafka监控工具?

- JMXTool 工具
- Kafka Manager
- Burrow
- JMXTrans + InfluxDB + Grafana
- Confluent Control Center

参考:

- 《Kafka并不难学》
- 《kafka入门与实践》
- 极客时间：Kafka核心技术与实战
- http://kafka.apache.org/

SpringCloud

1.什么是SpringCloud?

Spring Cloud为开发人员提供了工具，以快速构建分布式系统中的一些常见模式（例如，配置管理，服务发现，断路器，智能路由，微代理，控制总线，一次性令牌，全局锁，领导选举，分布式会话，群集状态）。它们可以在任何分布式环境中正常工作，包括开发人员自己的笔记本电脑，裸机数据中心以及Cloud Foundry等托管平台。

2.什么是微服务?

所谓的微服务是SOA架构下的最终产物，该架构的设计目标是为了肢解业务，使得服务能够独立运行。
微服务设计原则：

1、各司其职 。

2、服务高可用和可扩展性。

3.SpringCloud有哪些特征?

Spring Cloud专注于为典型的用例和可扩展性机制（包括其他用例）提供良好的开箱即用体验。

- 分布式/版本化配置
- 服务注册和发现
- 路由
- 服务到服务的调用
- 负载均衡
- 断路器
- 全局锁
- 领导选举和集群状态
- 分布式消息传递

4.SpringCloud核心组件?

Eureka : 注册中心

Ribbon： 客服端负载均衡

Hystrix： 服务容错处理

Feign: 声明式REST客户端

Zuul : 服务网关

Config : 分布式配置

5.SpringCloud基于什么协议?

HTTP

6.SpringCloud和Dubbo区别?

	Dubbo	Spring Cloud
服务注册中心	Zookeeper	Spring Cloud Netflix Eureka
服务调用方式	RPC	REST API
服务监控	Dubbo-monitor	Spring Boot Admin
断路器	不完善	Spring Cloud Netflix Hystrix
服务网关	无	Spring Cloud Netflix Zuul
分布式配置	无	Spring Cloud Config
服务跟踪	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream
批量任务	无	Spring Cloud Task

7.Eureka是什么?

云端服务发现，一个基于 REST 的服务，用于定位服务，以实现云端中间层服务发现和故障转移。

8.服务治理的基础角色?

服务注册中心：提供服务注册与发现的能力。

服务提供者：提供服务的应用，会把自己提供的服务注册到注册中心。

服务消费者：服务的消费者，从注册中心获取服务列表。

9.什么是服务续约?

在注册完服务以后，服务提供者会维护一个心跳来向注册中心证明自己还活着，以此防止被“剔除服务”。

10.什么是服务下线?

当服务实例进行正常关闭时，会发送一个REST请求（我要下线了）给注册中心，收到请求后，将该服务状态设置下线（DOWN），并把这事件传播出去。

11.什么是失效剔除?

当服务非正常下线时，可能服务注册中心没有收到下线请求，注册中心会创建一个定时任务（默认60s）将没有在固定时间(默认90s)内续约的服务剔除掉。

12.什么是自我保护机制?

在运行期间，注册中心会统计心跳失败比例在15分钟之内是否低于85%,如果低于的情况，注册中心会将当前注册实例信息保护起来，不再删除这些实例信息，当网络恢复后，退出自我保护机制。

自我保护机制让服务集群更稳定、健壮。

13.Ribbon是什么?

提供云端负载均衡，有多种负载均衡策略可供选择，可配合服务发现和断路器使用。

14.Ribbon负载均衡的注解是?

@LoadBalanced

15.Ribbon负载均衡策略有哪些?

RandomRule： 随机。

RoundRobinRule : 轮询。

RetryRule : 重试。

WeightedResponseTimeRule : 权重。

ClientConfigEnabledRoundRobinRule : 一般不用，通过继承该策略，默认的choose就实现了线性轮询机制。可以基于它来做扩展。

BestAvailableRule : 通过便利负载均衡器中维护的所有服务实例，会过滤到故障的，并选择并发请求最小的一个。

PredicateBasedRule : 先过滤清单，再轮询。

AvailabilityFilteringRule： 继承了父类的先过滤清单，再轮询。调整了算法。

ZoneAvoidanceRule : 该类也是PredicateBasedRule的子类，它可以组合过滤条件。以ZoneAvoidancePredicate为主过滤条件，以AvailabilityPredicate为次过滤条件。

16.什么是服务熔断?

服务熔断的作用类似于我们家用的保险丝，当某服务出现不可用或响应超时的情况时，为了防止整个系统出现雪崩，暂时停止对该服务的调用。

17.什么是服务降级?

服务降级是当服务器压力剧增的情况下，根据当前业务情况及流量对一些服务和页面有策略的降级，以此释放服务器资源以保证核心任务的正常运行。

18.什么是Hystrix?

熔断器，容错管理工具，旨在通过熔断机制控制服务和第三方库的节点,从而对延迟和故障提供更强大的容错能力。

19.断路器Hystrix的有哪些功能?

- 通过第三方客户端访问依赖服务出现高延迟或者失败时,为系统提供保护和控制 。
- 在复杂的分布式系统中防止级联失败(服务雪崩效应) 。
- 快速失败 (Failfast) 同时能快速恢复。
- 提供失败回滚 (Fallback) 和优雅的服务降级机制。
- 提供近实时的监控、 报警和运维控制手段。

20.Hystrix将远程调用封装到?

HystrixCommand 或者 HystrixObservableCommand对象中。

21.启动熔断降级服务的注解?

@EnableHystrix

22.什么是Feign?

Feign是一种声明式、模板化的HTTP客户端。

23.Feign优点?

1.feign采用的是基于接口的注解。
2.feign整合了ribbon，具有负载均衡的能力。
3.整合了Hystrix，具有熔断的能力。

24.什么是Config?

配置管理工具包，让你可以把配置放到远程服务器，集中化管理集群配置，目前支持本地存储、Git以及Subversion。

25.Config组件中的两个角色?

Config Server : 配置中心服务端。

Config Client : 配置中心客户端。

26.什么是Zuul?

Zuul 是在云平台上提供动态路由,监控,弹性,安全等边缘服务的框架。Zuul 相当于是设备和 Netflix 流应用的 Web 网站后端所有请求的前门。

27.使用Zuul的优点?

- 方便监控。可以在微服务网管手机监控数据并将其推送到外部系统进行分析。
- 方便认证。可在网关进行统一认证，然后在讲请求转发到后端服务。
- 隐藏架构实现细节，提供统一的入口给客户端请求，减少了客户端和每个微服务的交互次数。
- 可以统一处理切面任务，避免每个微服务自己开发，提升效率。

- 高可用高伸缩性的服务，避免单点失效。

28.Zuul的核心是?

过滤器。

29.Zuul有几种过滤器类型？分别是?

4种。

pre : 可以在请求被路由之前调用。

适用于身份认证的场景，认证通过后再继续执行下面的流程。

route : 在路由请求时被调用。

适用于灰度发布场景，在将要路由的时候可以做一些自定义的逻辑。

post :在 route 和 error 过滤器之后被调用。

这种过滤器将请求路由到达具体的服务之后执行。适用于需要添加响应头，记录响应日志等应用场景。

error : 处理请求时发生错误时被调用。

在执行过程中发送错误时会进入 error 过滤器，可以用来统一记录错误信息。

30.什么是Sleuth?

日志收集工具包，封装了Dapper和log-based追踪以及Zipkin和HTrace操作，为SpringCloud应用实现了一种分布式追踪解决方案。

31.Sleuth帮助我们做了哪些工作?

- 可以方便的了解到每个采样的请求耗时，分析出哪些服务调用比较耗时。
- 对于程序未捕捉的异常，可以在集成Zipkin服务页面上看到。
- 识别调用比较频繁的服务，从而进行优化。

32.什么是Bus?

事件、消息总线，用于在集群（例如，配置变化事件）中传播状态变化，可与Spring Cloud Config联合实现热部署。

33.eureka比zookeeper的优势在?

A:高可用 C:一致性，P:分区容错性

Zookeeper保证了CP，Eureka保证了AP。

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。

34.什么是Stream?

数据流操作开发包，封装了与Redis,Rabbit、Kafka等发送接收消息。

35.更多知识?

SpringCloud这个体系东西还是挺大的，小编会不断的丰富内容以及优化。欢迎大家关注我的公众号获得最新动态《Java小咖秀》。

参考:

- 《Spring Cloud微服务实战》
- 《Spring Cloud微服务全栈技术与案例解析》
- 《Spring Cloud微服务架构开发实战》
- https://spring.io/projects/spring-cloud
- https://www.springcloud.cc/
- 百度百科

简历

几个制作简历不错的网站:

https://mp.weixin.qq.com/s/z0k922U6jwXe5VYlz33gaA

原文链接:https://www.zhihu.com/question/25002833 ThoughtWorks中国回答

大家伙让一让，这个问题让老司机先答！作为一个潜入IT圈五年之久、看过数万份简历的HR，在这个问题上还是有点发言权的。HR在筛选简历时主要从公司需求出发，重点不一，不过还是有很多“通用”的套路，为了在30秒内判断出这份简历是否值得跟进，我认为程序员写简历的正确姿势是这样的:

一、基本格调

即打开简历之后的第一印象。就好比我们看见一个人，会有一个整体的感觉，他是fashion的、小清新的还是老道的？有了第一印象之后再慢慢分解来看。

加分写法:

- 简洁明了，逻辑结构清晰。
- 字体，排版，顺畅，清晰整齐就好。
- 最好是PDF格式，兼容性强且不易乱序。

减分写法:

- 设计的过于浮夸或者过于简单的。（eg.有的简历五颜六色、非常酷炫，却半天找不到联系方式，抑

或是只有个人基本信息和公司名称）
- 写了十几页，半天打不开的，或者加载了半天，打开还乱码。

二、基本信息（姓名/性别/毕业院校/电话/邮箱/居住地/期望地）

加分写法:

- 清晰罗列出以上信息，这样HR就不用在接下来的电话沟通或面试中再去追问这些内容，建立我们接下来电话沟通对你的熟悉度。
- 再额外能加上QQ或者微信就更好了（以防有时候电话打不通哦，时不时会遇到这种情况）

减分写法:

- 大部分的基本信息没有写
- 甩给我一个Github链接，极致简洁的几句描述，需要通过你的链接来找你的联系方式。（如果不是博客写的特别好，基本是要放弃你了）

三、工作经历&项目经历

加分写法:

- 工作经历项目经历可参照万能的STAR法则来写，STAR不清楚的童鞋点这里啦
- 效力过哪些公司，我们匹配的公司？ BAT？ 知名大型互联网公司？
- 做过什么行业领域，和我们目前的行业是否匹配
- 擅长的技术语言，应用了哪些技术栈，（Java, Scala，Ruby, React, Vue, Microservice…）
- 经历的项目复杂度，及在项目中承担什么样的角色(人的变化/技术的变化/环境的变化/不同工作经历相同角色的不同点)
- 时间节点（空档期）

减分写法:

- 看了半天，不知所云，没有任何亮点，没有让人有去和你聊一聊深扒的信息。

来几个栗子

栗子1错误打开方式:

- XX（全栈工程师）2013.06 — 至今
- 参与需求分析及实现方案设计。
- 设计数据库表结构，实现后台功能及web页面展示。
- 产品线上部署及运维。
- ay 配置管理工程师 2010.03 — 2013.03
- 负责公司产品性能测试，及线上数据分析
- 负责公司配置管理，环境维护等工作

点评：看不出来他做的什么事情，没有逻辑性，甚至不知道他做的什么技术语言。

栗子2正确打开方式:

西安XXX公司 Java工程师 — 2016.2月-2017.2月

1、MOGU推荐架构数据与缓存层设计开发

- MOGU是一款时尚资讯app,负责推荐页面资讯feed流的展示及用户历史的展示

- 负责数据层,处理前端逻辑整个开发工作,分布式rpc服务搭建
- 负责进行压测监测、缓存处理,对接又进行改进优化,主用redis缓存

2、基于JAVA的电商爬虫开发

- 使用java搭建爬虫server平台,进行配置和开发,进行网页改版监测功能开发
- 爬取淘宝时尚品牌与其他电商网站商品品牌与详情等
- 通过频率、ip池、匿名代理等应对一些网站的反爬

3、同图搜索Solr服务开发

基于算法组的同图策略,使用solr做java接又实现rpc服务搭建,进行索引构建和solr实现

北京XXX

java大数据工程师— 2013.4月-2015.12月

1、负责实时流消息处理应用系统构建和实现

- 在调研了kafka的优势和我们的具体需求之后,用kafka作为消费者,保证高吞吐处理消息,并持久化消息的同时供其它服务使用,进行了系统的设计和搭建使用。 本地日志保证消息不丢失,并通过记录游标滑动重复读取数据。
- 使用storm 负责搭建消息处理架构,并完成基于业务的消息落地,提供后续的数据 统计分析实时和离线任务,诸如pv、uv等数据,为运营做决策
- 网站用户行为埋点和基于js的日志收集器开发,定义接又和前端部门配合。主用go 2、hadoop集群搭建和数据分析处理

2、基于CDH的集群搭建工作,后期进行维护

编写MapReduce程序,能将复杂工作逻辑化,尽最大能力发挥大数据应用的特点, 对程序高要求,监控自己程序运行情况,使用内存合理,注重增量和全量运算的利弊

3、调度系统设计与实现 基于quartz2搭建调度平台,带徒弟实现相关功能并定期review代码

4、数据库调优 负责主从搭建,并掌握主从搭建的利弊,了解业界mycat原理,有数据库优化经验,能 正确并擅长使用索引,对锁有深刻的认识

5、网站开发 java web网站业务开发,并能很好的使用缓存技术,对重构有实际的经验,并对面向对 象开发有全面的实战经验。了解java数据结构的使用场景,虽然对于大并发没有太大的 发挥余地,但是掌握了数据结构,对于并发和阻塞等有自己的见解。

点评：非常清晰的告诉简历阅读者自己做了什么事情，负责了什么样的事情，用了什么技术栈，且逻辑连贯。

四、工作期望&个人评价

加分写法：

- 对自己有一个全方位的一个描述总结，让别人更好的解读你。或者在此处，高亮你的优点特长有哪些。
- 即使不写个人评价，也一定记得写上工作期望。

减分写法：

完全看不出个性特点，写和没写没什么区别。 来几个栗子

栗子1 错误打开方式

为人性格,诚实谦虚，勤奋，能吃苦耐劳，有耐心，有团队意识，能和同学和谐相处，能虚心接受别人的建议的人。

责任心强，善于沟通，具有良好的团队合作精神；专业扎实，具有较强的钻研精神和学习能力；性格比较乐观外向，喜欢打羽毛球。

栗子2正确打开方式

- 我对自己的定位: 主攻前端,同时在其他方面打打辅助。我不希望过于依赖别人,即使没有后端没有设计没有产品经理,我依然想要把这个产品做到完美。毕竟全栈才能最高效地解决问题。
- 我对工作的态度: 第一,要高效完成自己的本职工作。第二,要在完成的基础上寻找完美。第三,要在完美的基础上,与其他同事 互相交流学习,互相提升。工作是一种生活方式,不是一份养家糊口的差事。
- 我怎样克服困难: 不用百度是第一原则,在遇到技术问题时我往往会去Google、Stack over flow上寻找答案。但通常很多问题 并不一定已经被人解决,所以熟练地阅读源码、在手册、规范甚至 REPL的环境自己做实验才是最终解决问题的办法。相信事实的结果,自己动手去做。
- 怎样保持自己的视野:我一直认为软件开发中视野极其重要,除了在 Twitter 上关注业界大牛,Github Trending 也是每周必刷。 另外 Podcast、Hacker News、Reddit 以及TechRadar 也是重要的一手资料。保持开阔视野才能找到更酷的解决方案。
- 我的优势: 热爱技术、自学能力强,有良好的自我认知。全面的技能树与开阔的视野,良好的心态、情商与沟通能力。
- 我的劣势: 非科班出身没有科班同学对算法的熟练掌握,但我决定死磕技术,弥补不足。

栗子3正确打开方式

- 极客、热爱技术、热爱开源
- Ruby on Rails：精通
- Agile/Lean：精通
- ReactJS：掌握
- Docker：掌握
- AWS：掌握

五、 是否有博客，个人技术栈点等

1. 看到有这项的HR两眼已经放光了，加分加分项，说明你真正的热爱技术，善于学习总结，乐于分享，且有投入自己的业余时间到软件事业中。
2. 我喜欢的书籍:《重构》《卓有成效的程序员》《代码整洁之道》等
3. 我喜欢的社区: 图灵社区，知乎，博客园，Stack Over flow，Google Developer Group等
4. 我的博客链接、个人作品链接如下:

- https://github.com/github
- http://www.oschina.net/
- https://www.cnblogs.com/
- https://itunes.apple.com/app/battle-of-crab/id1121917063?l=en&mt=8

六、简历内容真实性

老司机提醒你，你简历的任意一个细节将会是后面面试中的呈堂证供。

基本就这些了，希望对大家能有帮助，看起简历来几十秒，码字还是个体力活。

写在最后: 希望大家都能拿到自己心仪的offer。