java基础

面向对象

什么是面向对象?

对比面向过程，是两种不同的处理问题的角度

面向过程更注重事情的每一个步骤及顺序，面向对象更注重事情有哪些参与者（对象）、及各自需要做什么

比如：洗衣机洗衣服

面向过程会将任务拆解成一系列的步骤（函数），1、打开洗衣机----->2、放衣服----->3、放洗衣粉----->4、清洗----->5、烘干

面向对象会拆出人和洗衣机两个对象：

人：打开洗衣机　放衣服　 放洗衣粉

洗衣机：清洗　 烘干

从以上例子能看出，面向过程比较直接高效，而面向对象更易于复用、扩展和维护

面向对象

封装：封装的意义，在于明确标识出允许外部使用的所有成员函数和数据项

内部细节对外部调用透明，外部调用无需修改或者关心内部实现

1、javabean的属性私有，提供getset对外访问，因为属性的赋值或者获取逻辑只能由javabean本身决定。而不能由外部胡乱修改

```
private String name;
public void setName(String name){
    this.name = "tuling_"+name;
}
该name有自己的命名规则，明显不能由外部直接赋值
```

2、orm框架

操作数据库，我们不需要关心链接是如何建立的、sql是如何执行的，只需要引入mybatis，调方法即可

继承：继承基类的方法，并做出自己的改变和/或扩展

子类共性的方法或者属性直接使用父类的，而不需要自己再定义，只需扩展自己个性化的

多态：基于对象所属类的不同，外部对同一个方法的调用，实际执行的逻辑不同。

继承，方法重写，父类引用指向子类对象

```
父类类型  变量名  =  new  子类对象 ;
变量名.方法名();
```

无法调用子类特有的功能

JDK JRE JVM

JDK：

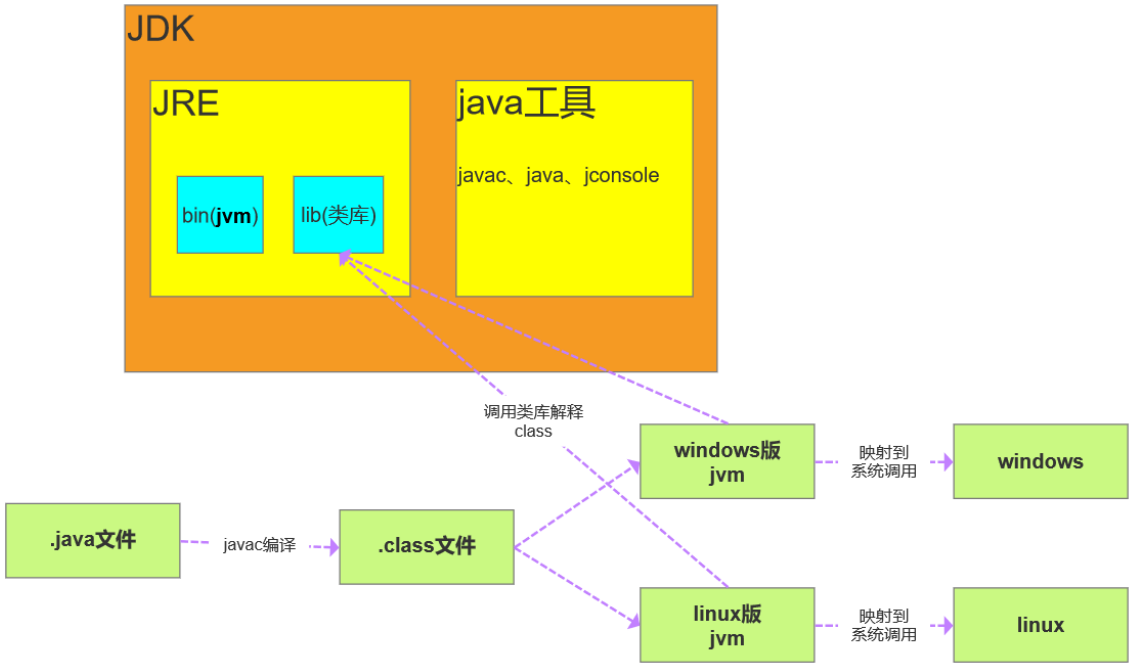
Java Develpment Kit java 开发工具

JRE：

Java Runtime Environment java运行时环境

JVM：

java Virtual Machine java 虚拟机



==和equals比较

==对比的是栈中的值，基本数据类型是变量值，引用类型是堆中内存对象的地址

equals：object中默认也是采用==比较，通常会重写

Object

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

String

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

上述代码可以看出，String类中被复写的equals()方法其实是比较两个字符串的内容。

```
public class StringDemo {
      public static void main(String args[]) {
          String str1 = "Hello";
          String str2 = new String("Hello");
          String str3 = str2; // 引用传递
          System.out.println(str1 == str2); // false
          System.out.println(str1 == str3); // false
          System.out.println(str2 == str3); // true
          System.out.println(str1.equals(str2)); // true
          System.out.println(str1.equals(str3)); // true
          System.out.println(str2.equals(str3)); // true
     }
 }
```

hashCode与equals

hashCode介绍：

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个int整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。hashCode() 定义在JDK的Object.java中，Java中的任何类都包含有hashCode() 函数。
散列表存储的是键值对(key-value)，它的特点是：能根据“键”快速的检索出对应的“值”。这其中就利用到了散列码！（可以快速找到所需要的对象）

为什么要有hashCode：

以“HashSet如何检查重复”为例子来说明为什么要有hashCode：
对象加入HashSet时，HashSet会先计算对象的hashcode值来判断对象加入的位置，看该位置是否有值，如果没有、HashSet会假设对象没有重复出现。但是如果发现有值，这时会调用equals（）方法来检查两个对象是否真的相同。如果两者相同，HashSet就不会让其加入操作成功。如果不同的话，就会重新散列到其他位置。这样就大大减少了equals的次数，相应就大大提高了执行速度。

- 如果两个对象相等，则hashcode一定也是相同的
- 两个对象相等,对两个对象分别调用equals方法都返回true
- 两个对象有相同的hashcode值，它们也不一定是相等的
- 因此，equals方法被覆盖过，则hashCode方法也必须被覆盖
- hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写hashCode()，则该class的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）

final

最终的

- 修饰类：表示类不可被继承
- 修饰方法：表示方法不可被子类覆盖，但是可以重载
- 修饰变量：表示变量一旦被赋值就不可以更改它的值。

（1）修饰成员变量

- 如果final修饰的是类变量，只能在静态初始化块中指定初始值或者声明该类变量时指定初始值。
- 如果final修饰的是成员变量，可以在非静态初始化块、声明该变量或者构造器中执行初始值。

（2）修饰局部变量

系统不会为局部变量进行初始化，局部变量必须由程序员显示初始化。因此使用final修饰局部变量时，即可以在定义时指定默认值（后面的代码不能对变量再赋值），也可以不指定默认值，而在后面的代码中对final变量赋初值（仅一次）

```
public class FinalVar {
    final static int a = 0;//再声明的时候就需要赋值 或者静态代码块赋值
    /**
    static{
        a = 0;
    }
    */
    final int b = 0;//再声明的时候就需要赋值 或者代码块中赋值    或者构造器赋值
```

```
    /*{
        b = 0;
    }*/
    public static void main(String[] args) {
        final int localA;   //局部变量只声明没有初始化，不会报错,与final无关。
        localA = 0;//在使用之前一定要赋值
        //localA = 1;  但是不允许第二次赋值
    }
}
```

(3) 修饰基本类型数据和引用类型数据

- 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。**但是引用的值是可变的。**

```
public class FinalReferenceTest{
    public static void main(){
        final int[] iArr={1,2,3,4};
        iArr[2]=-3;//合法
        iArr=null;//非法，对iArr不能重新赋值

        final Person p = new Person(25);
        p.setAge(24);//合法
        p=null;//非法
    }
}
```

为什么局部内部类和匿名内部类只能访问局部final变量?

编译之后会生成两个class文件，Test.class Test1.class

```
public class Test {
    public static void main(String[] args)  {
    }
    //局部final变量a,b
    public void test(final int b) {//jdk8在这里做了优化，不用写,语法糖，但实际上也是有的，也不能修改
        final int a = 10;
        //匿名内部类
        new Thread(){
            public void run() {
                System.out.println(a);
                System.out.println(b);
            };
        }.start();
    }
}

class OutClass {
        private int age = 12;

        public void outPrint(final int x) {
                class InClass {
                        public void InPrint() {
                                System.out.println(x);
                                System.out.println(age);
```

```
                        }
                }
                new InClass().InPrint();
        }
}
```

首先需要知道的一点是: 内部类和外部类是处于同一个级别的，内部类不会因为定义在方法中就会随着方法的执行完毕就被销毁。

这里就会产生问题：当外部类的方法结束时，局部变量就会被销毁了，但是内部类对象可能还存在(只有没有人再引用它时，才会死亡)。这里就出现了一个矛盾：内部类对象访问了一个不存在的变量。为了解决这个问题，就将局部变量复制了一份作为内部类的成员变量，这样当局部变量死亡后，内部类仍可以访问它，实际访问的是局部变量的"copy"。这样就好像延长了局部变量的生命周期

将局部变量复制为内部类的成员变量时，必须保证这两个变量是一样的，也就是如果我们在内部类中修改了成员变量，方法中的局部变量也得跟着改变，怎么解决问题呢?

就将局部变量设置为final，对它初始化后，我就不让你再去修改这个变量，就保证了内部类的成员变量和方法的局部变量的一致性。这实际上也是一种妥协。使得局部变量与内部类内建立的拷贝保持一致。

String、StringBuffer、StringBuilder

String是final修饰的，不可变，每次操作都会产生新的String对象

StringBuffer和StringBuilder都是在原对象上操作

StringBuffer是线程安全的，StringBuilder线程不安全的

StringBuffer方法都是synchronized修饰的

性能：StringBuilder > StringBuffer > String

场景：经常需要改变字符串内容时使用后面两个

优先使用StringBuilder，多线程使用共享变量时使用StringBuffer

重载和重写的区别

重载： 发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同，发生在编译时。
重写： 发生在父子类中，方法名、参数列表必须相同，返回值范围小于等于父类，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类；如果父类方法访问修饰符为private则子类就不能重写该方法。

```
public int add(int a,String b)
public String add(int a,String b)
//编译报错
```

接口和抽象类的区别

- 抽象类可以存在普通成员函数，而接口中只能存在public abstract 方法。
- 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是public static final类型的。
- 抽象类只能继承一个，接口可以实现多个。

接口的设计目的，是对类的行为进行约束（更准确的说是一种"有"约束，因为接口不能规定类不可以有什么行为），也就是提供一种机制，可以强制要求不同的类具有相同的行为。它只约束了行为的有无，但不对如何实现行为进行限制。

而抽象类的设计目的，是代码复用。当不同的类具有某些相同的行为(记为行为集合A)，且其中一部分行为的实现方式一致时（A的非真子集，记为B），可以让这些类都派生于一个抽象类。在这个抽象类中实现了B，避免让所有的子类来实现B，这就达到了代码复用的目的。而A减B的部分，留给各个子类自己实现。正是因为A-B在这里没有实现，所以抽象类不允许实例化出来（否则当调用到A-B时，无法执行）。

抽象类是对类本质的抽象，表达的是 is a 的关系，比如：`BMW` is a `Car`。抽象类包含并实现子类的通用特性，将子类存在差异化的特性进行抽象，交由子类去实现。

而接口是对行为的抽象，表达的是 like a 的关系。比如：`Bird` like a `Aircraft`（像飞行器一样可以飞），但其本质上 is a `Bird`。接口的核心是定义行为，即实现类可以做什么，至于实现类主体是谁、是如何实现的，接口并不关心。

使用场景：当你关注一个事物的本质的时候，用抽象类；当你关注一个操作的时候，用接口。

抽象类的功能要远超过接口，但是，定义抽象类的代价高。因为高级语言来说（从实际设计上来说也是）每个类只能继承一个类。在这个类中，你必须继承或编写出其所有子类的所有共性。虽然接口在功能上会弱化许多，但是它只是针对一个动作的描述。而且你可以在一个类中同时实现多个接口。在设计阶段会降低难度

List和Set的区别

- List：有序，按对象进入的顺序保存对象，可重复，允许多个Null元素对象，可以使用Iterator取出所有元素，在逐一遍历，还可以使用get(int index)获取指定下标的元素
- Set：无序，不可重复，最多允许有一个Null元素对象，取元素时只能用Iterator接口取得所有元素，在逐一遍历各个元素

ArrayList和LinkedList区别

ArrayList：基于动态数组，连续内存存储，适合下标访问（随机访问），扩容机制：因为数组长度固定，超出长度存数据时需要新建数组，然后将老数组的数据拷贝到新数组，如果不是尾部插入数据还会涉及到元素的移动（往后复制一份，插入新元素），使用尾插法并指定初始容量可以极大提升性能、甚至超过linkedList（需要创建大量的node对象）

LinkedList：基于链表，可以存储在分散的内存中，适合做数据插入及删除操作，不适合查询：需要逐一遍历

遍历LinkedList必须使用iterator不能使用for循环，因为每次for循环体内通过get(i)取得某一元素时都需要对list重新进行遍历，性能消耗极大。
另外不要试图使用indexOf等返回元素索引，并利用其进行遍历，使用indexlOf对list进行了遍历，当结果为空时会遍历整个列表。

HashMap和HashTable有什么区别？其底层实现是什么?

区别：

（1）HashMap方法没有synchronized修饰，线程非安全，HashTable线程安全；

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

2.底层实现：数组+链表实现

jdk8开始链表高度到8、数组长度超过64，链表转变为红黑树，元素以内部类Node节点存在

- 计算key的hash值，二次hash然后对数组长度取模，对应到数组下标，
- 如果没有产生hash冲突(下标位置没有元素)，则直接创建Node存入数组，
- 如果产生hash冲突，先进行equal比较，相同则取代该元素，不同，则判断链表高度插入链表，链表高度达到8，并且数组长度到64则转变为红黑树，长度低于6则将红黑树转回链表
- key为null，存在下标0的位置

数组扩容

ConcurrentHashMap原理，jdk7和jdk8版本的区别

jdk7：

数据结构：ReentrantLock+Segment+HashEntry，一个Segment中包含一个HashEntry数组，每个HashEntry又是一个链表结构

元素查询：二次hash，第一次Hash定位到Segment，第二次Hash定位到元素所在的链表的头部

锁：Segment分段锁 Segment继承了ReentrantLock，锁定操作的Segment，其他的Segment不受影响，并发度为segment个数，可以通过构造函数指定，数组扩容不会影响其他的segment

get方法无需加锁，volatile保证

jdk8：

数据结构：synchronized+CAS+Node+红黑树，Node的val和next都用volatile修饰，保证可见性

查找，替换，赋值操作都使用CAS

锁：锁链表的head节点，不影响其他元素的读写，锁粒度更细，效率更高，扩容时，阻塞所有的读写操作、并发扩容

读操作无锁：
Node的val和next使用volatile修饰，读写线程对该变量互相可见
数组用volatile修饰，保证扩容时被读线程感知

什么是字节码？采用字节码的好处是什么？

java中的编译器和解释器：

Java中引入了虚拟机的概念，即在机器和编译程序之间加入了一层抽象的虚拟的机器。这台虚拟的机器在任何平台上都提供给编译程序一个的共同的接口。
编译程序只需要面向虚拟机，生成虚拟机能够理解的代码，然后由解释器来将虚拟机代码转换为特定系统的机器码执行。在Java中，这种供虚拟机理解的代码叫做 字节码（即扩展名为 .class的文件），它不面向任何特定的处理器，只面向虚拟机。
每一种平台的解释器是不同的，但是实现的虚拟机是相同的。Java源程序经过编译器编译后变成字节码，字节码由虚拟机解释执行，虚拟机将每一条要执行的字节码送给解释器，解释器将其翻译成特定机器上的机器码，然后在特定的机器上运行。这也就是解释了Java的编译与解释并存的特点。
Java源代码---->编译器---->jvm可执行的Java字节码(即虚拟指令)---->jvm---->jvm中解释器----->机器可执行的二进制机器码---->程序运行。
采用字节码的好处：

Java语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。所以Java程序运行时比较高效，而且，由于字节码并不专对一种特定的机器，因此，Java程序无须重新编译便可在多种不同的计算机上运行。

Java中的异常体系

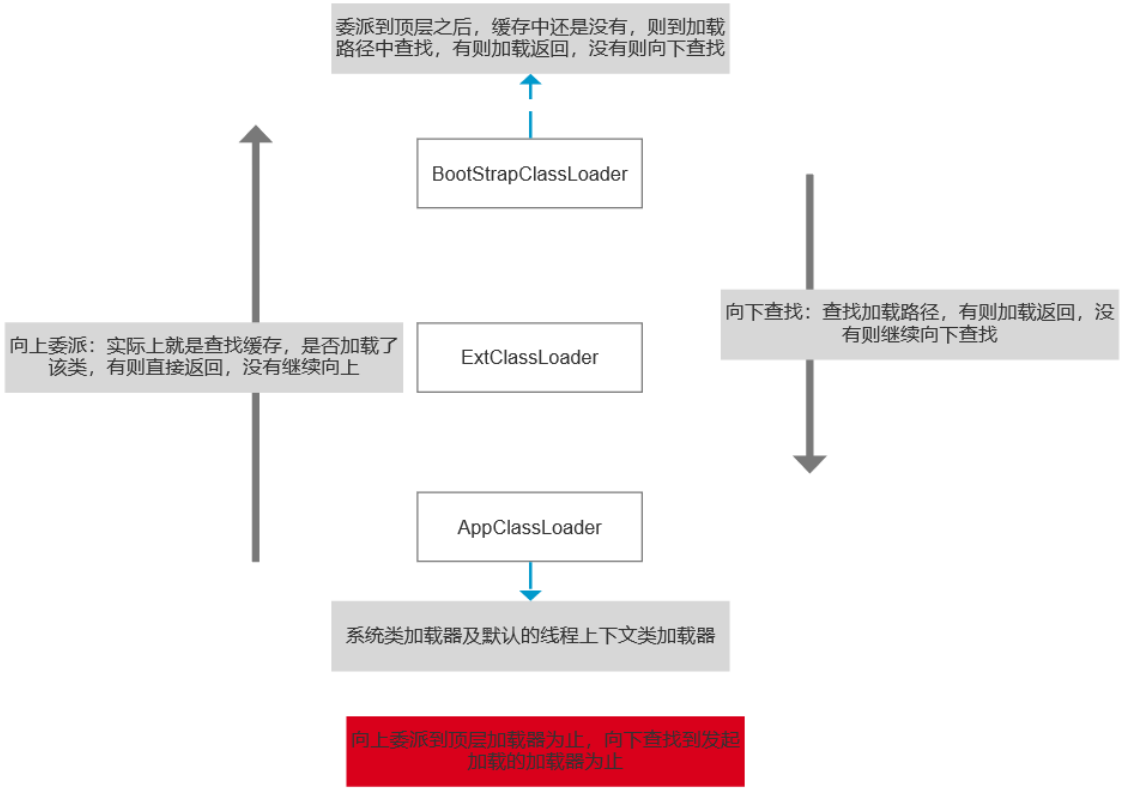
Java中的所有异常都来自顶级父类Throwable。
Throwable下有两个子类Exception和Error。
Error是程序无法处理的错误，一旦出现这个错误，则程序将被迫停止运行。
Exception不会导致程序停止，又分为两个部分RunTimeException运行时异常和CheckedException检查异常。
RunTimeException常常发生在程序运行过程中，会导致程序当前线程执行失败。CheckedException常常发生在程序编译过程中，会导致程序编译不通过。

Java类加载器

JDK自带有三个类加载器：bootstrap ClassLoader、ExtClassLoader、AppClassLoader。
BootStrapClassLoader是ExtClassLoader的父类加载器，默认负责加载%JAVA_HOME%lib下的jar包和class文件。
ExtClassLoader是AppClassLoader的父类加载器，负责加载%JAVA_HOME%/lib/ext文件夹下的jar包和class类。
AppClassLoader是自定义类加载器的父类，负责加载classpath下的类文件。系统类加载器，线程上下文加载器

继承ClassLoader实现自定义类加载器

双亲委托模型



双亲委派模型的好处：

- 主要是为了安全性，避免用户自己编写的类动态替换 Java的一些核心类，比如 String。
- 同时也避免了类的重复加载，因为 JVM中区分不同类，不仅仅是根据类名，相同的 class文件被不同的 ClassLoader加载就是不同的两个类

GC如何判断对象可以被回收

- 引用计数法：每个对象有一个引用计数属性，新增一个引用时计数加1，引用释放时计数减1，计数为0时可以回收，
- 可达性分析法：从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是不可用的，那么虚拟机就判断是可回收对象。

> 引用计数法，可能会出现A 引用了 B，B 又引用了 A，这时候就算他们都不再使用了，但因为相互引用 计数器=1 永远无法被回收。

GC Roots的对象有：

- 虚拟机栈(栈帧中的本地变量表）中引用的对象
- 方法区中类静态属性引用的对象
- 方法区中常量引用的对象
- 本地方法栈中JNI(即一般说的Native方法)引用的对象

可达性算法中的不可达对象并不是立即死亡的，对象拥有一次自我拯救的机会。对象被系统宣告死亡至少要经历两次标记过程：第一次是经过可达性分析发现没有与GC Roots相连接的引用链，第二次是在由虚拟机自动建立的Finalizer队列中判断是否需要执行finalize()方法。

当对象变成(GC Roots)不可达时，GC会判断该对象是否覆盖了finalize方法，若未覆盖，则直接将其回收。否则，若对象未执行过finalize方法，将其放入F-Queue队列，由一低优先级线程执行该队列中对象的finalize方法。执行finalize方法完毕后，GC会再次判断该对象是否可达，若不可达，则进行回收，否则，对象“复活”

每个对象只能触发一次finalize()方法

由于finalize()方法运行代价高昂，不确定性大，无法保证各个对象的调用顺序，不推荐大家使用，建议遗忘它。

线程、并发相关

线程的生命周期？线程有几种状态

1.线程通常有五种状态，创建，就绪，运行、阻塞和死亡状态。

2.阻塞的情况又分为三种：

(1)、等待阻塞：运行的线程执行wait方法，该线程会释放占用的所有资源，JVM会把该线程放入“等待池”中。进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用notify或notifyAll方法才能被唤醒，wait是object类的方法

(2)、同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入“锁池”中。

(3)、其他阻塞：运行的线程执行sleep或join方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep状态超时、join等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。sleep是Thread类的方法

1.新建状态（New）：新创建了一个线程对象。

2.就绪状态（Runnable）：线程对象创建后，其他线程调用了该对象的start方法。该状态的线程位于可运行线程池中，变得可运行，等待获取CPU的使用权。

3.运行状态（Running）：就绪状态的线程获取了CPU，执行程序代码。

4.阻塞状态（Blocked）：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

5.死亡状态（Dead）：线程执行完了或者因异常退出了run方法，该线程结束生命周期。

sleep()、wait()、join()、yield()的区别

1.锁池

所有需要竞争同步锁的线程都会放在锁池当中，比如当前对象的锁已经被其中一个线程得到，则其他线程需要在这个锁池进行等待，当前面的线程释放同步锁后锁池中的线程去竞争同步锁，当某个线程得到后会进入就绪队列进行等待cpu资源分配。

2.等待池

当我们调用wait（）方法后，线程会放到等待池当中，等待池的线程是不会去竞争同步锁。只有调用了notify（）或notifyAll()后等待池的线程才会开始去竞争锁，notify（）是随机从等待池选出一个线程放到锁池，而notifyAll()是将等待池的所有线程放到锁池当中

1、sleep 是 Thread 类的静态本地方法，wait 则是 Object 类的本地方法。

2、sleep方法不会释放lock，但是wait会释放，而且会加入到等待队列中。

> sleep就是把cpu的执行资格和执行权释放出去，不再运行此线程，当定时时间结束再取回cpu资源，参与cpu的调度，获取到cpu资源后就可以继续运行了。而如果sleep时该线程有锁，那么sleep不会释放这个锁，而是把锁带着进入了冻结状态，也就是说其他需要这个锁的线程根本不可能获取到这个锁。也就是说无法执行程序。如果在睡眠期间其他线程调用了这个线程的interrupt方法，那么这个线程也会抛出interruptexception异常返回，这点和wait是一样的。

3、sleep方法不依赖于同步器synchronized，但是wait需要依赖synchronized关键字。

4、sleep不需要被唤醒（休眠之后推出阻塞），但是wait需要（不指定时间需要被别人中断）。

5、sleep 一般用于当前线程休眠，或者轮循暂停操作，wait 则多用于多线程之间的通信。

6、sleep 会让出 CPU 执行时间且强制上下文切换，而 wait 则不一定，wait 后可能还是有机会重新竞争到锁继续执行的。

yield（）执行后线程直接进入就绪状态，马上释放了cpu的执行权，但是依然保留了cpu的执行资格，所以有可能cpu下次进行线程调度还会让这个线程获取到执行权继续执行

join（）执行后线程进入阻塞状态，例如在线程B中调用线程A的join（），那线程B会进入到阻塞队列，直到线程A结束或中断线程

```
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
            System.out.println("22222222");
        }
    });
    t1.start();
    t1.join();
    // 这行代码必须要等t1全部执行完毕，才会执行
    System.out.println("1111");
}

22222222
1111
```

对线程安全的理解

不是线程安全、应该是内存安全，堆是共享内存，可以被所有线程访问

> 当多个线程访问一个对象时，如果不用进行额外的同步控制或其他的协调操作，调用这个对象的行为都可以获得正确的结果，我们就说这个对象是线程安全的

堆是进程和线程共有的空间，分全局堆和局部堆。全局堆就是所有没有分配的空间，局部堆就是用户分配的空间。堆在操作系统对进程初始化的时候分配，运行过程中也可以向系统要额外的堆，但是用完了要还给操作系统，要不然就是内存泄漏。

> 在Java中，堆是Java虚拟机所管理的内存中最大的一块，是所有线程共享的一块内存区域，在虚拟机启动时创建。堆所存在的内存区域的唯一目的就是存放对象实例，几乎所有的对象实例以及数组都在这里分配内存。

栈是每个线程独有的，保存其运行状态和局部自动变量的。栈在线程开始的时候初始化，每个线程的栈互相独立，因此，栈是线程安全的。操作系统在切换线程的时候会自动切换栈。栈空间不需要在高级语言里面显式的分配和释放。

目前主流操作系统都是多任务的，即多个进程同时运行。为了保证安全，每个进程只能访问分配给自己的内存空间，而不能访问别的进程的，这是由操作系统保障的。

在每个进程的内存空间中都会有一块特殊的公共区域，通常称为堆（内存）。进程内的所有线程都可以访问到该区域，这就是造成问题的潜在原因。

Thread、Runable的区别

Thread和Runnable的实质是继承关系，没有可比性。无论使用Runnable还是Thread，都会new Thread，然后执行run方法。用法上，如果有复杂的线程操作需求，那就选择继承Thread，如果只是简单的执行一个任务，那就实现runnable。

```
//会卖出多一倍的票
public class Test {
    public static void main(String[] args) {
        // TODO Auto-generated method stub

        new MyThread().start();
        new MyThread().start();

    }


     static class MyThread extends Thread{
        private int ticket = 5;
        public void run(){
            while(true){
                System.out.println("Thread ticket = " + ticket--);
                if(ticket < 0){
                    break;
                }
            }
        }
    }
}
```

```
//正常卖出
public class Test2 {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MyThread2 mt=new MyThread2();
        new Thread(mt).start();
        new Thread(mt).start();


    }
    static class MyThread2 implements Runnable{
        private int ticket = 5;
        public void run(){
            while(true){
                System.out.println("Runnable ticket = " + ticket--);
                if(ticket < 0){
                    break;
                }
            }
        }
    }
}
```

原因是：MyThread创建了两个实例，自然会卖出两倍，属于用法错误

对守护线程的理解

守护线程：为所有非守护线程提供服务的线程；任何一个守护线程都是整个JVM中所有非守护线程的保姆；

守护线程类似于整个进程的一个默默无闻的小喽喽；它的生死无关重要，它却依赖整个进程而运行；哪天其他线程结束了，没有要执行的了，程序就结束了，理都没理守护线程，就把它中断了；

注意： 由于守护线程的终止是自身无法控制的，因此千万不要把IO、File等重要操作逻辑分配给它；因为它不靠谱；

守护线程的作用是什么?
举例， GC垃圾回收线程：就是一个经典的守护线程，当我们的程序中不再有任何运行的Thread,程序就不会再产生垃圾，垃圾回收器也就无事可做，所以当垃圾回收线程是JVM上仅剩的线程时，垃圾回收线程会自动离开。它始终在低级别的状态中运行，用于实时监控和管理系统中的可回收资源。
应用场景：（1）来为其它线程提供服务支持的情况；（2） 或者在任何情况下，程序结束时，这个线程必须正常且立刻关闭，就可以作为守护线程来使用；反之，如果一个正在执行某个操作的线程必须要正确地关闭掉否则就会出现不好的后果的话，那么这个线程就不能是守护线程，而是用户线程。通常都是些关键的事务，比方说，数据库录入或者更新，这些操作都是不能中断的。

thread.setDaemon(true)必须在thread.start()之前设置，否则会跑出一个IllegalThreadStateException异常。你不能把正在运行的常规线程设置为守护线程。

在Daemon线程中产生的新线程也是Daemon的。

守护线程不能用于去访问固有资源，比如读写操作或者计算逻辑。因为它会在任何时候甚至在一个操作的中间发生中断。

Java自带的多线程框架，比如ExecutorService，会将守护线程转换为用户线程，所以如果要使用后台线程就不能用Java的线程池。

ThreadLocal的原理和使用场景

每一个`Thread`对象均含有一个`ThreadLocalMap`类型的成员变量`threadLocals`，它存储本线程中所有ThreadLocal对象及其对应的值

`ThreadLocalMap`由一个个`Entry`对象构成

`Entry`继承自`WeakReference<ThreadLocal<?>>`，一个`Entry`由`ThreadLocal`对象和`Object`构成。由此可见，`Entry`的key是ThreadLocal对象，并且是一个弱引用。当没指向key的强引用后，该key就会被垃圾收集器回收

当执行set方法时，ThreadLocal首先会获取当前线程对象，然后获取当前线程的ThreadLocalMap对象。再以当前ThreadLocal对象为key，将值存储进ThreadLocalMap对象中。

get方法执行过程类似。ThreadLocal首先会获取当前线程对象，然后获取当前线程的ThreadLocalMap对象。再以当前ThreadLocal对象为key，获取对应的value。

由于每一条线程均含有各自**私有的**ThreadLocalMap容器，这些容器相互独立互不影响，因此不会存在线程安全性问题，从而也无需使用同步机制来保证多条线程访问容器的互斥性。

使用场景：

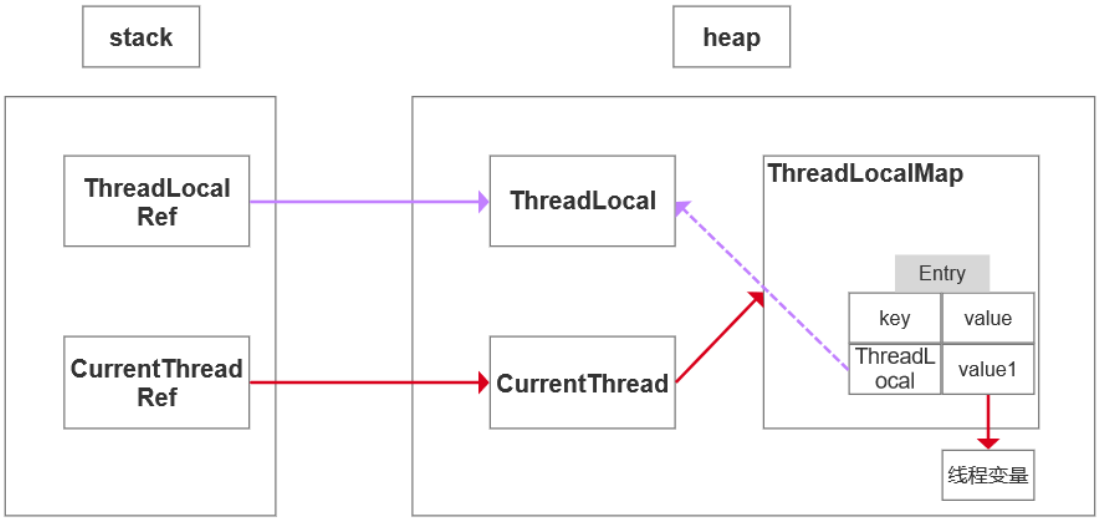
1、在进行对象跨层传递的时候，使用ThreadLocal可以避免多次传递，打破层次间的约束。

2、线程间数据隔离

3、进行事务操作，用于存储线程事务信息。

4、数据库连接，Session会话管理。

> Spring框架在事务开始时会给当前线程绑定一个Jdbc Connection,在整个事务过程都是使用该线程绑定的connection来执行数据库操作，实现了事务的隔离性。Spring框架里面就是用的ThreadLocal来实现这种隔离



ThreadLocal内存泄露原因，如何避免

内存泄露为程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光，

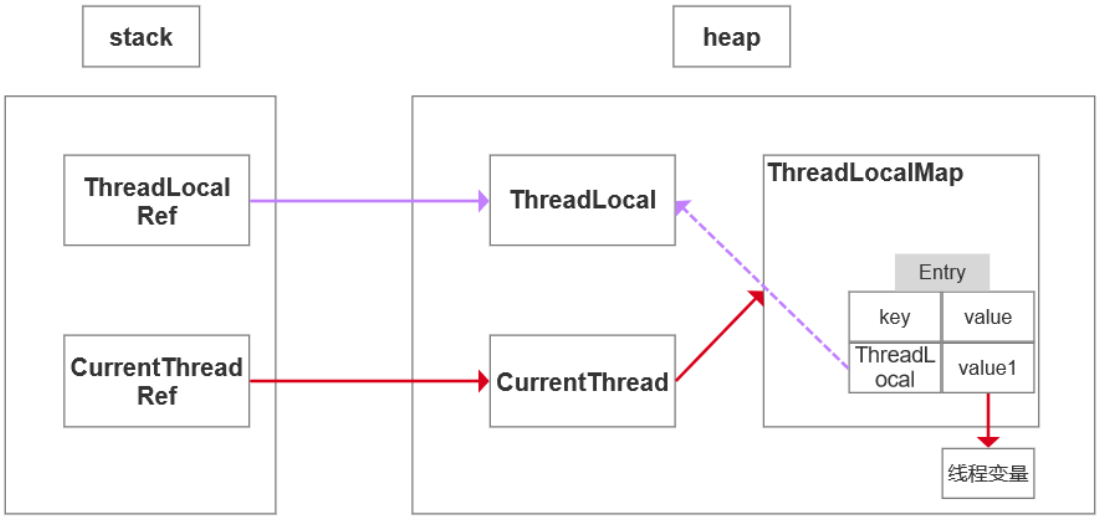
不再会被使用的对象或者变量占用的内存不能被回收，就是内存泄露。

强引用：使用最普遍的引用(new)，一个对象具有强引用，不会被垃圾回收器回收。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不回收这种对象。

如果想取消强引用和某个对象之间的关联，可以显式地将引用赋值为null，这样可以使JVM在合适的时间就会回收该对象。

弱引用：JVM进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。在java中，用java.lang.ref.WeakReference类来表示。可以在缓存中使用弱引用。

ThreadLocal的实现原理，每一个Thread维护一个ThreadLocalMap，key为使用**弱引用**的ThreadLocal实例，value为线程变量的副本



hreadLocalMap使用ThreadLocal的弱引用作为key，如果一个ThreadLocal不存在外部**强引用**时，Key(ThreadLocal)势必会被GC回收，这样就会导致ThreadLocalMap中key为null， 而value还存在着强引用，只有thead线程退出以后,value的强引用链条才会断掉，但如果当前线程再迟迟不结束的话，这些key为null的Entry的value就会一直存在一条强引用链（红色链条）

key 使用强引用

当hreadLocalMap的key为强引用回收ThreadLocal时，因为ThreadLocalMap还持有ThreadLocal的强引用，如果没有手动删除，ThreadLocal不会被回收，导致Entry内存泄漏。

key 使用弱引用

当ThreadLocalMap的key为弱引用回收ThreadLocal时，由于ThreadLocalMap持有ThreadLocal的弱引用，即使没有手动删除，ThreadLocal也会被回收。当key为null，在下一次ThreadLocalMap调用set(),get()，remove()方法的时候会被清除value值。

因此，ThreadLocal内存泄漏的根源是：由于ThreadLocalMap的生命周期跟Thread一样长，如果没有手动删除对应key就会导致内存泄漏，而不是因为弱引用。

ThreadLocal正确的使用方法

- 每次使用完ThreadLocal都调用它的remove()方法清除数据
- 将ThreadLocal变量定义成private static，这样就一直存在ThreadLocal的强引用，也就能保证任何时候都能通过ThreadLocal的弱引用访问到Entry的value值，进而清除掉 。

并发、并行、串行的区别

串行在时间上不可能发生重叠，前一个任务没搞定，下一个任务就只能等着

并行在时间上是重叠的，两个任务在**同一时刻互不干扰**的同时执行。

并发允许两个任务彼此干扰。统一时间点、只有一个任务运行，交替执行

并发的三大特性

- 原子性

原子性是指在一个操作中cpu不可以在中途暂停然后再调度，即不被中断操作，要不全部执行完成，要不都不执行。就好比转账，从账户A向账户B转1000元，那么必然包括2个操作：从账户A减去1000元，往账户B加上1000元。2个操作必须全部完成。

```
private long count = 0;

public void calc() {
    count++;
}
```

- 1：将 count 从主存读到工作内存中的副本中
- 2：+1的运算
- 3：将结果写入工作内存
- 4：将工作内存的值刷回主存(什么时候刷入由操作系统决定，不确定的)

那程序中原子性指的是最小的操作单元，比如自增操作，它本身其实并不是原子性操作，分了3步的，包括读取变量的原始值、进行加1操作、写入工作内存。所以在多线程中，有可能一个线程还没自增完，可能才执行到第二部，另一个线程就已经读取了值，导致结果错误。那如果我们能保证自增操作是一个原子性的操作，那么就能保证其他线程读取到的一定是自增后的数据。

关键字：synchronized

- 可见性

当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

若两个线程在不同的cpu，那么线程1改变了i的值还没刷新到主存，线程2又使用了i，那么这个i值肯定还是之前的，线程1对变量的修改线程没看到这就是可见性问题。

```
//线程1
boolean stop = false;
while(!stop){
    doSomething();
}

//线程2
stop = true;
```

如果线程2改变了stop的值，线程1一定会停止吗？不一定。当线程2更改了stop变量的值之后，但是还没来得及写入主存当中，线程2转去做其他事情了，那么线程1由于不知道线程2对stop变量的更改，因此还会一直循环下去。

关键字：volatile、synchronized、final

- 有序性

虚拟机在进行代码编译时，对于那些改变顺序之后不会对最终结果造成影响的代码，虚拟机不一定会按照我们写的代码的顺序来执行，有可能将他们重排序。实际上，对于有些代码进行重排序之后，虽然对变量的值没有造成影响，但有可能会出现线程安全问题。

```
int a = 0;
bool flag = false;

public void write() {
    a = 2;              //1
    flag = true;        //2
}

public void multiply() {
    if (flag) {         //3
        int ret = a * a;//4
    }

}
```

write方法里的1和2做了重排序，线程1先对flag赋值为true，随后执行到线程2，ret直接计算出结果，再到线程1，这时候a才赋值为2,很明显迟了一步

关键字：volatile、synchronized

volatile本身就包含了禁止指令重排序的语义，而synchronized关键字是由“一个变量在同一时刻只允许一条线程对其进行lock操作”这条规则明确的。

synchronized关键字同时满足以上三种特性，但是volatile关键字不满足原子性。

在某些情况下，volatile的同步机制的性能确实要优于锁(使用synchronized关键字或java.util.concurrent包里面的锁)，因为volatile的总开销要比锁低。

我们判断使用volatile还是加锁的唯一依据就是volatile的语义能否满足使用的场景(原子性)

volatile

1. 保证被volatile修饰的共享变量对所有线程总是可见的，也就是当一个线程修改了一个被volatile修饰共享变量的值，新值总是可以被其他线程立即得知。

   ```
   //线程1
   boolean stop = false;
   while(!stop){
       doSomething();
   }

   //线程2
   stop = true;
   ```

   如果线程2改变了stop的值，线程1一定会停止吗？不一定。当线程2更改了stop变量的值之后，但是还没来得及写入主存当中，线程2转去做其他事情了，那么线程1由于不知道线程2对stop变量的更改，因此还会一直循环下去。

2. 禁止指令重排序优化。

```
int a = 0;
bool flag = false;

public void write() {
    a = 2;              //1
    flag = true;        //2
}

public void multiply() {
    if (flag) {         //3
        int ret = a * a;//4
    }

}
```

write方法里的1和2做了重排序，线程1先对flag赋值为true，随后执行到线程2，ret直接计算出结果，再到线程1，这时候a才赋值为2,很明显迟了一步。

但是用volatile修饰之后就变得不一样了

第一：使用volatile关键字会强制将修改的值立即写入主存；

第二：使用volatile关键字的话，当线程2进行修改时，会导致线程1的工作内存中缓存变量stop的缓存行无效（反映到硬件层的话，就是CPU的L1或者L2缓存中对应的缓存行无效）；

第三：由于线程1的工作内存中缓存变量stop的缓存行无效，所以线程1再次读取变量stop的值时会去主存读取。

`inc++;` 其实是两个步骤，先加加，然后再赋值。不是原子性操作，所以volatile不能保证线程安全。

为什么用线程池？解释下线程池参数?

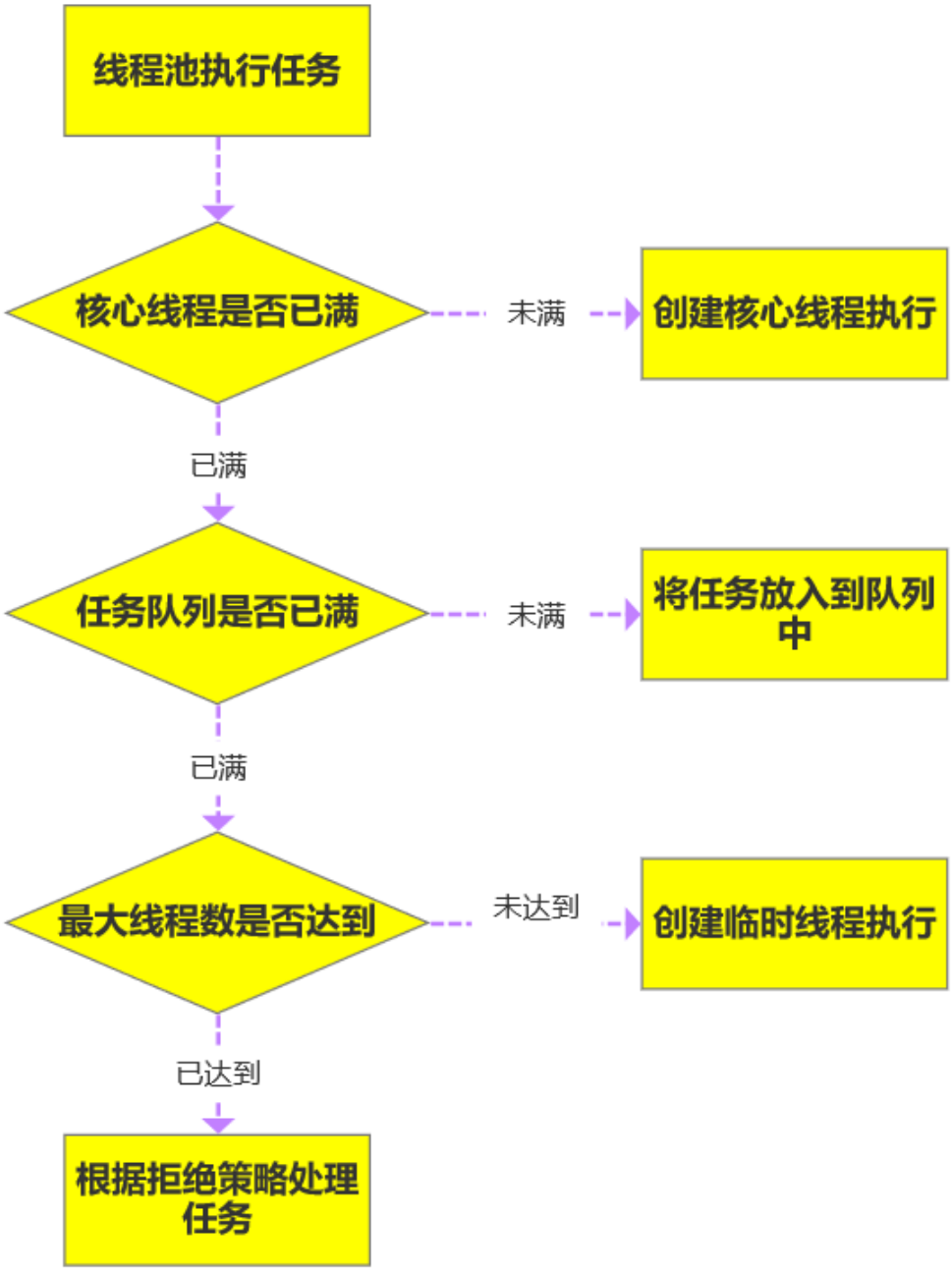
1、降低资源消耗；提高线程利用率，降低创建和销毁线程的消耗。

2、提高响应速度；任务来了，直接有线程可用可执行，而不是先创建线程，再执行。

3、提高线程的可管理性；线程是稀缺资源，使用线程池可以统一分配调优监控。

- `corePoolSize` 代表核心线程数，也就是正常情况下创建工作的线程数，这些线程创建后并不会消除，而是一种常驻线程
- `maxinumPoolSize` 代表的是最大线程数，它与核心线程数相对应，表示最大允许被创建的线程数，比如当前任务较多，将核心线程数都用完了，还无法满足需求时，此时就会创建新的线程，但是线程池内线程总数不会超过最大线程数
- `keepAliveTime`、`unit` 表示超出核心线程数之外的线程的空闲存活时间，也就是核心线程不会消除，但是超出核心线程数的部分线程如果空闲一定的时间则会被消除,我们可以通过 `setKeepAliveTime` 来设置空闲时间
- `workQueue` 用来存放待执行的任务，假设我们现在核心线程都已被使用，还有任务进来则全部放入队列，直到整个队列被放满但任务还再持续进入则会开始创建新的线程
- `ThreadFactory` 实际上是一个线程工厂，用来生产线程执行任务。我们可以选择使用默认的创建工厂，产生的线程都在同一个组内，拥有相同的优先级，且都不是守护线程。当然我们也可以选择自定义线程工厂，一般我们会根据业务来制定不同的线程工厂
- `Handler` 任务拒绝策略，有两种情况，第一种是当我们调用 `shutdown` 等方法关闭线程池后，这时候即使线程池内部还有没执行完的任务正在执行，但是由于线程池已经关闭，我们再继续想线程池提交任务就会遭到拒绝。另一种情况就是当达到最大线程数，线程池已经没有能力继续处理新提交的任务时，这是也就拒绝

简述线程池处理流程



线程池中阻塞队列的作用？为什么是先添加列队而不是先创建最大线程？

1、一般的队列只能保证作为一个有限长度的缓冲区，如果超出了缓冲长度，就无法保留当前的任务了，阻塞队列通过阻塞可以保留住当前想要继续入队的任务。

阻塞队列可以保证任务队列中没有任务时阻塞获取任务的线程，使得线程进入wait状态，释放cpu资源。

阻塞队列自带阻塞和唤醒的功能，不需要额外处理，无任务执行时,线程池利用阻塞队列的take方法挂起，从而维持核心线程的存活、不至于一直占用cpu资源

2、在创建新线程的时候，是要获取全局锁的，这个时候其它的就得阻塞，影响了整体效率。

就好比一个企业里面有10个（core）正式工的名额，最多招10个正式工，要是任务超过正式工人数（task > core）的情况下，工厂领导（线程池）不是首先扩招工人，还是这10人，但是任务可以稍微积压一下，即先放到队列去（代价低）。10个正式工慢慢干，迟早会干完的，要是任务还在继续增加，超过正式工的加班忍耐极限了（队列满了），就的招外包帮忙了（注意是临时工）要是正式工加上外包还是不能完成任务，那新来的任务就会被领导拒绝了（线程池的拒绝策略）。

线程池中线程复用原理

线程池将线程和任务进行解耦，线程是线程，任务是任务，摆脱了之前通过 Thread 创建线程时的一个线程必须对应一个任务的限制。

在线程池中，同一个线程可以从阻塞队列中不断获取新任务来执行，其核心原理在于线程池对 Thread 进行了封装，并不是每次执行任务都会调用 Thread.start() 来创建新线程，而是让每个线程去执行一个“循环任务”，在这个“循环任务”中不停检查是否有任务需要被执行，如果有则直接执行，也就是调用任务中的 run 方法，将 run 方法当成一个普通的方法执行，通过这种方式只使用固定的线程就将所有任务的 run 方法串联起来。

spring

如何实现一个IOC容器

1、配置文件配置包扫描路径

2、递归包扫描获取.class文件

3、反射、确定需要交给IOC管理的类

4、对需要注入的类进行依赖注入

- 配置文件中指定需要扫描的包路径
- 定义一些注解，分别表示访问控制层、业务服务层、数据持久层、依赖注入注解、获取配置文件注解
- 从配置文件中获取需要扫描的包路径，获取到当前路径下的文件信息及文件夹信息，我们将当前路径下所有以.class结尾的文件添加到一个Set集合中进行存储
- 遍历这个set集合，获取在类上有指定注解的类，并将其交给IOC容器，定义一个安全的Map用来存储这些对象
- 遍历这个IOC容器，获取到每一个类的实例，判断里面是有有依赖其他的类的实例，然后进行递归注入

spring是什么?

轻量级的开源的J2EE框架。它是一个容器框架，用来装javabean（java对象），中间层框架（万能胶）可以起一个连接作用，比如说把Struts和hibernate粘合在一起运用，可以让我们的企业开发更快、更简洁

Spring是一个轻量级的控制反转（IoC)和面向切面（AOP）的容器框架

--从大小与开销两方面而言Spring都是轻量级的。

--通过控制反转(IoC)的技术达到松耦合的目的

--提供了面向切面编程的丰富支持，允许通过分离应用的业务逻辑与系统级服务进行内聚性的开发

--包含并管理应用对象(Bean)的配置和生命周期，这个意义上是一个容器。

--将简单的组件配置、组合成为复杂的应用，这个意义上是一个框架。

谈谈你对AOP的理解

系统是由许多不同的组件所组成的，每一个组件各负责一块特定功能。除了实现自身核心功能之外，这些组件还经常承担着额外的职责。例如日志、事务管理和安全这样的核心服务经常融入到自身具有核心业务逻辑的组件中去。这些系统服务经常被称为横切关注点，因为它们会跨越系统的多个组件。

当我们需要为分散的对象引入公共行为的时候，OOP则显得无能为力。也就是说，OOP允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能。

日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。

在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。

AOP：将程序中的交叉业务逻辑（比如安全，日志，事务等），封装成一个切面，然后注入到目标对象（具体业务逻辑）中去。AOP可以对某个对象或某些对象的功能进行增强，比如对象中的方法进行增强，可以在执行某个方法之前额外的做一些事情，在某个方法执行之后额外的做一些事情

谈谈你对IOC的理解

容器概念、控制反转、依赖注入

ioc容器：实际上就是个map（key，value），里面存的是各种对象（在xml里配置的bean节点、@repository、@service、@controller、@component），在项目启动的时候会读取配置文件里面的bean节点，根据全限定类名使用反射创建对象放到map里、扫描到打上上述注解的类还是通过反射创建对象放到map里。

这个时候map里就有各种对象了，接下来我们在代码里需要用到里面的对象时，再通过DI注入（autowired、resource等注解，xml里bean节点内的ref属性，项目启动的时候会读取xml节点ref属性根据id注入，也会扫描这些注解，根据类型或id注入；id就是对象名）。

控制反转：

没有引入IOC容器之前，对象A依赖于对象B，那么对象A在初始化或者运行到某一点的时候，自己必须主动去创建对象B或者使用已经创建的对象B。无论是创建还是使用对象B，控制权都在自己手上。

引入IOC容器之后，对象A与对象B之间失去了直接联系，当对象A运行到需要对象B的时候，IOC容器会主动创建一个对象B注入到对象A需要的地方。

通过前后的对比，不难看出来：对象A获得依赖对象B的过程,由主动行为变为了被动行为，控制权颠倒过来了，这就是“控制反转”这个名称的由来。

全部对象的控制权全部上缴给“第三方”IOC容器，所以，IOC容器成了整个系统的关键核心，它起到了一种类似“粘合剂”的作用，把系统中的所有对象粘合在一起发挥作用，如果没有这个“粘合剂”，对象与对象之间会彼此失去联系，这就是有人把IOC容器比喻成“粘合剂”的由来。

依赖注入：

“获得依赖对象的过程被反转了”。控制被反转之后，获得依赖对象的过程由自身管理变为了由IOC容器主动注入。依赖注入是实现IOC的方法，就是由IOC容器在运行期间，动态地将某种依赖关系注入到对象之中。

BeanFactory和ApplicationContext有什么区别?

ApplicationContext是BeanFactory的子接口

ApplicationContext提供了更完整的功能：

①继承MessageSource，因此支持国际化。

②统一的资源文件访问方式。

③提供在监听器中注册bean的事件。

④同时加载多个配置文件。

⑤载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层。

- BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

- ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean ,确保当你需要的时候，你就不用等待，因为它们已经创建好了。
- 相对于基本的BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。
- BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader。
- BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册。

描述一下Spring Bean的生命周期?

1、解析类得到BeanDefinition

2、如果有多个构造方法，则要推断构造方法

3、确定好构造方法后，进行实例化得到一个对象

4、对对象中的加了@Autowired注解的属性进行属性填充

5、回调Aware方法，比如BeanNameAware，BeanFactoryAware

6、调用BeanPostProcessor的初始化前的方法

7、调用初始化方法

8、调用BeanPostProcessor的初始化后的方法，在这里会进行AOP

9、如果当前创建的bean是单例的则会把bean放入单例池

10、使用bean

11、Spring容器关闭时调用DisposableBean中destory()方法

解释下Spring支持的几种bean的作用域。

- singleton：默认，每个容器中只有一个bean的实例，单例的模式由BeanFactory自身来维护。该对象的生命周期是与Spring IOC容器一致的（但在第一次被注入时才会创建）。
- prototype：为每一个bean请求提供一个实例。在每次注入时都会创建一个新的对象
- request：bean被定义为在每个HTTP请求中创建一个单例对象，也就是说在单个请求中都会复用这一个单例对象。
- session：与request范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。
- application：bean被定义为在ServletContext的生命周期中复用一个单例对象。
- websocket：bean被定义为在websocket的生命周期中复用一个单例对象。

global-session：全局作用域，global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet共用全局的存储变量的话，那么这全局变量需要存储在global-session中。全局作用域与Servlet中的session作用域效果相同。

Spring框架中的单例Bean是线程安全的么?

Spring中的Bean默认是单例模式的，框架并没有对bean进行多线程的封装处理。

如果Bean是有状态的 那就需要开发人员自己来进行线程安全的保证，最简单的办法就是改变bean的作用域 把 "singleton"改为"protopyte' 这样每次请求Bean就相当于是 new Bean() 这样就可以保证线程的安全了。

- 有状态就是有数据存储功能
- 无状态就是不会保存数据　　controller、service和dao层本身并不是线程安全的，只是如果只是调用里面的方法，而且多线程调用一个实例的方法，会在内存中复制变量，这是自己的线程的工作内存，是安全的。

Dao会操作数据库Connection，Connection是带有状态的，比如说数据库事务，Spring的事务管理器使用Threadlocal为不同线程维护了一套独立的connection副本，保证线程之间不会互相影响（Spring是如何保证事务获取同一个Connection的）

不要在bean中声明任何有状态的实例变量或类变量，如果必须如此，那么就使用ThreadLocal把变量变为线程私有的，如果bean的实例变量或类变量需要在多个线程之间共享，那么就只能使用synchronized、lock、CAS等这些实现线程同步的方法了。

Spring 框架中都用到了哪些设计模式?

简单工厂：由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类。

> Spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得Bean对象，但是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。

工厂方法：

> 实现了FactoryBean接口的bean是一类叫做factory的bean。其特点是，spring会在使用getBean()调用获得该bean时，会自动调用该bean的getObject()方法，所以返回的不是factory这个bean，而是这个bean.getOjbect()方法的返回值。

单例模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点

spring对单例的实现： spring中的单例模式完成了后半句话，即提供了全局的访问点BeanFactory。但没有从构造器级别去控制单例，这是因为spring管理的是任意的java对象。

适配器模式：

Spring定义了一个适配接口，使得每一种Controller有一种对应的适配器实现类，让适配器代替controller执行相应的方法。这样在扩展Controller时，只需要增加一个适配器类就完成了SpringMVC的扩展了。

装饰器模式：动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator模式相比生成子类更为灵活。

Spring中用到的包装器模式在类名上有两种表现：一种是类名中含有Wrapper，另一种是类名中含有Decorator。

动态代理：

切面在应用运行的时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象创建动态的创建一个代理对象。SpringAOP就是以这种方式织入切面的。

织入：把切面应用到目标对象并创建新的代理对象的过程。

观察者模式：

spring的事件驱动模型使用的是 观察者模式 ，Spring中Observer模式常用的地方是listener的实现。

策略模式：

Spring框架的资源访问Resource接口。该接口提供了更强的资源访问能力，Spring 框架本身大量使用了 Resource 接口来访问底层资源。

模板方法：父类定义了骨架（调用哪些方法及顺序），某些特定方法由子类实现。

最大的好处：代码复用，减少重复代码。除了子类要实现的特定方法，其他方法及方法调用顺序都在父类中预先写好了。

refresh方法

Spring事务的实现方式和原理以及隔离级别?

在使用Spring框架时，可以有两种使用事务的方式，一种是编程式的，一种是申明式的，@Transactional注解就是申明式的。

首先，事务这个概念是数据库层面的，Spring只是基于数据库中的事务进行了扩展，以及提供了一些能让程序员更加方便操作事务的方式。

比如我们可以通过在某个方法上增加@Transactional注解，就可以开启事务，这个方法中所有的sql都会在一个事务中执行，统一成功或失败。

在一个方法上加了@Transactional注解后，Spring会基于这个类生成一个代理对象，会将这个代理对象作为bean，当在使用这个代理对象的方法时，如果这个方法上存在@Transactional注解，那么代理逻辑会先把事务的自动提交设置为false，然后再去执行原本的业务逻辑方法，如果执行业务逻辑方法没有出现异常，那么代理逻辑中就会将事务进行提交，如果执行业务逻辑方法出现了异常，那么则会将事务进行回滚。

当然，针对哪些异常回滚事务是可以配置的，可以利用@Transactional注解中的rollbackFor属性进行配置，默认情况下会对RuntimeException和Error进行回滚。

spring事务隔离级别就是数据库的隔离级别：外加一个默认级别

- read uncommitted（未提交读）
- read committed（提交读、不可重复读）
- repeatable read（可重复读）
- serializable（可串行化）

> 数据库的配置隔离级别是Read Commited,而Spring配置的隔离级别是Repeatable Read，请问这时隔离级别是以哪一个为准？
> 以Spring配置的为准，如果spring设置的隔离级别数据库不支持，效果取决于数据库

spring事务传播机制

多个事务方法相互调用时,事务如何在这些方法间传播

> 方法A是一个事务的方法，方法A执行过程中调用了方法B，那么方法B有无事务以及方法B对事务的要求不同都会对方法A的事务具体执行造成影响，同时方法A的事务对方法B的事务执行也有影响，这种影响具体是什么就由两个方法所定义的事务传播类型所决定。

REQUIRED(Spring默认的事务传播类型)：如果当前没有事务，则自己新建一个事务，如果当前存在事务，则加入这个事务

SUPPORTS：当前存在事务，则加入当前事务，如果当前没有事务，就以非事务方法执行

MANDATORY：当前存在事务，则加入当前事务，如果当前事务不存在，则抛出异常。

REQUIRES_NEW：创建一个新事务，如果存在当前事务，则挂起该事务。

NOT_SUPPORTED：以非事务方式执行,如果当前存在事务，则挂起当前事务

NEVER：不使用事务，如果当前事务存在，则抛出异常

NESTED：如果当前事务存在，则在嵌套事务中执行，否则REQUIRED的操作一样（开启一个事务）

> 和REQUIRES_NEW的区别
> REQUIRES_NEW是新建一个事务并且新开启的这个事务与原有事务无关，而NESTED则是当前存在事务时（我们把当前事务称之为父事务）会开启一个嵌套事务（称之为一个子事务）。 在NESTED情况下父事务回滚时，子事务也会回滚，而在REQUIRES_NEW情况下，原有事务回滚，不会影响新开启的事务。
>
> 和REQUIRED的区别
> REQUIRED情况下，调用方存在事务时，则被调用方和调用方使用同一事务，那么被调用方出现异常时，由于共用一个事务，所以无论调用方是否catch其异常，事务都会回滚 而在NESTED情况下，被调用方发生异常时，调用方可以catch其异常，这样只有子事务回滚，父事务不受影响

spring事务什么时候会失效?

spring事务的原理是AOP，进行了切面增强，那么失效的根本原因是这个AOP不起作用了！常见情况有如下几种

1、发生自调用，类里面使用this调用本类的方法（this通常省略），此时这个this对象不是代理类，而是UserService对象本身!
解决方法很简单，让那个this变成UserService的代理类即可!

2、方法不是public的

> `@Transactional` 只能用于 `public` 的方法上，否则事务不会失效，如果要用在非 `public` 方法上，可以开启 `AspectJ` 代理模式。

3、数据库不支持事务

4、没有被spring管理

5、异常被吃掉，事务不会回滚(或者抛出的异常没有被定义，默认为RuntimeException)

什么是bean的自动装配，有哪些方式?

开启自动装配，只需要在xml配置文件中定义“autowire”属性。

```
<bean id="cutomer" class="com.xxx.xxx.Customer" autowire="" />
```

autowire属性有五种装配的方式：

- no – 缺省情况下，自动配置是通过“ref”属性手动设定 。

  ```
  手动装配：以value或ref的方式明确指定属性值都是手动装配。
  需要通过‘ref’属性来连接bean。
  ```

- byName-根据bean的属性名称进行自动装配。

  ```
  Cutomer的属性名称是person，Spring会将bean id为person的bean通过setter方法进行自动装配。
  <bean id="cutomer" class="com.xxx.xxx.Cutomer" autowire="byName"/>
  <bean id="person" class="com.xxx.xxx.Person"/>
  ```

- byType-根据bean的类型进行自动装配。

  ```
  Cutomer的属性person的类型为Person，Spirng会将Person类型通过setter方法进行自动装配。
  <bean id="cutomer" class="com.xxx.xxx.Cutomer" autowire="byType"/>
  <bean id="person" class="com.xxx.xxx.Person"/>
  ```

- constructor-类似byType，不过是应用于构造器的参数。如果一个bean与构造器参数的类型形同，则进行自动装配，否则导致异常。

  ```
  Cutomer构造函数的参数person的类型为Person，Spirng会将Person类型通过构造方法进行自动装配。
  <bean id="cutomer" class="com.xxx.xxx.Cutomer" autowire="construtor"/>
  <bean id="person" class="com.xxx.xxx.Person"/>
  ```

- autodetect-如果有默认的构造器，则通过constructor方式进行自动装配，否则使用byType方式进行自动装配。

  ```
  如果有默认的构造器，则通过constructor方式进行自动装配，否则使用byType方式进行自动装配。
  ```

@Autowired自动装配bean，可以在字段、setter方法、构造函数上使用。

springmvc、springBoot

Spring Boot、Spring MVC 和 Spring 有什么区别

spring是一个IOC容器，用来管理Bean，使用依赖注入实现控制反转，可以很方便的整合各种框架，提供AOP机制弥补OOP的代码重复问题、更方便将不同类不同方法中的共同处理抽取成切面、自动注入给方法执行，比如日志、异常等

springmvc是spring对web框架的一个解决方案，提供了一个总的前端控制器Servlet，用来接收请求，然后定义了一套路由策略（url到handle的映射）及适配执行handle，将handle结果使用视图解析技术生成视图展现给前端

springboot是spring提供的一个快速开发工具包，让程序员能更方便、更快速的开发spring+springmvc应用，简化了配置（约定了默认配置），整合了一系列的解决方案（starter机制）、redis、mongodb、es，可以开箱即用

SpringMVC 工作流程

1）用户发送请求至前端控制器 DispatcherServlet。

2）DispatcherServlet 收到请求调用 HandlerMapping 处理器映射器。

3）处理器映射器找到具体的处理器(可以根据 xml 配置、注解进行查找)，生成处理器及处理器拦截器(如果有则生成)一并返回给 DispatcherServlet。

4）DispatcherServlet 调用 HandlerAdapter 处理器适配器。

5）HandlerAdapter 经过适配调用具体的处理器(Controller，也叫后端控制器)

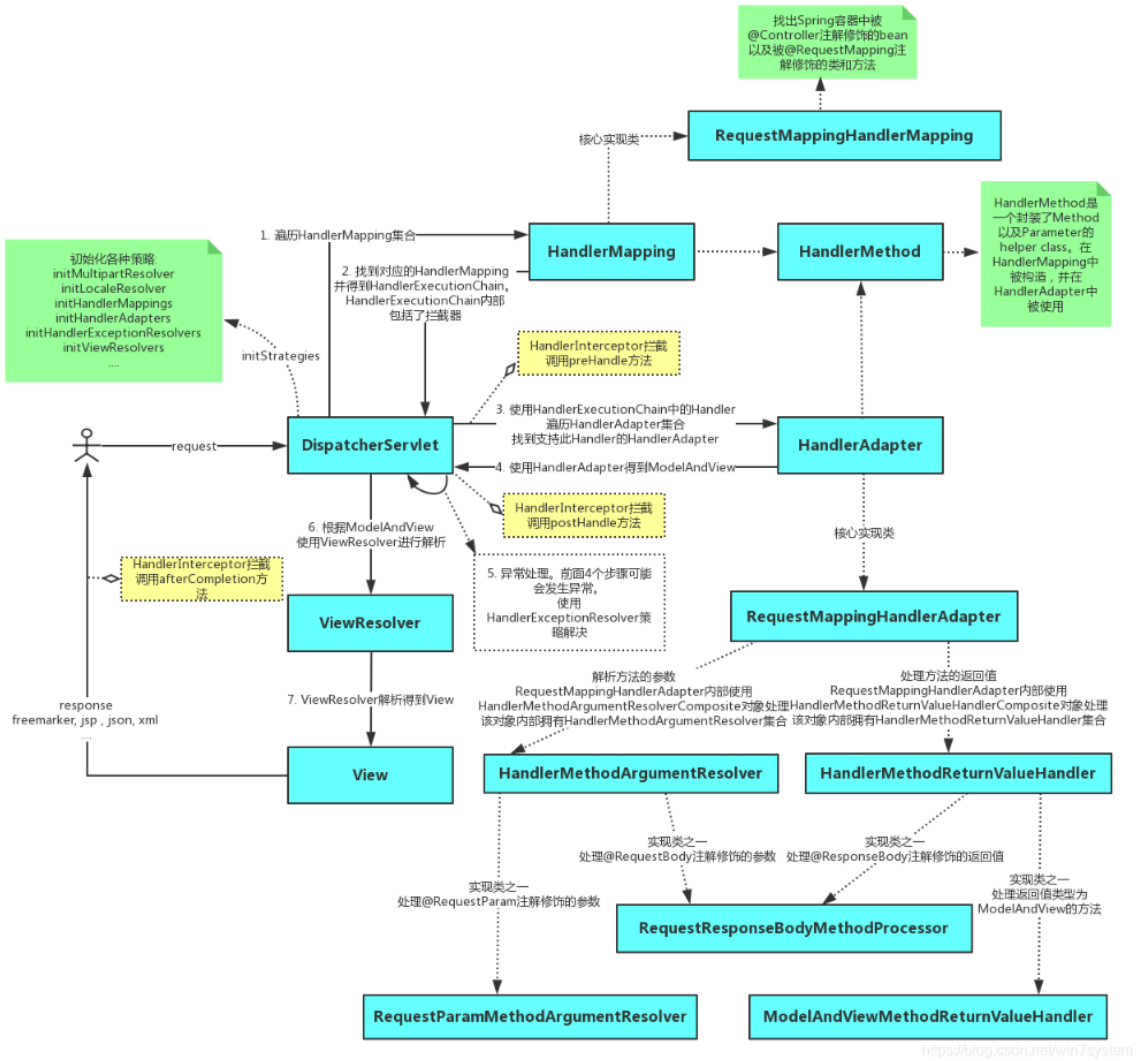
6）Controller 执行完成返回 ModelAndView。

7）HandlerAdapter 将 controller 执行结果 ModelAndView 返回给 DispatcherServlet。8）DispatcherServlet 将 ModelAndView 传给 ViewReslover 视图解析器。

9）ViewReslover 解析后返回具体 View。

10）DispatcherServlet 根据 View 进行渲染视图（即将模型数据填充至视图中）。

11）DispatcherServlet 响应用户。



Spring MVC的主要组件?

Handler：也就是处理器。它直接应对着MVC中的C也就是Controller层，它的具体表现形式有很多，可以是类，也可以是方法。在Controller层中@RequestMapping标注的所有方法都可以看成是一个Handler，只要可以实际处理请求就可以是Handler

1、HandlerMapping
initHandlerMappings(context)，处理器映射器，根据用户请求的资源uri来查找Handler的。在SpringMVC中会有很多请求，每个请求都需要一个Handler处理，具体接收到一个请求之后使用哪个Handler进行，这就是HandlerMapping需要做的事。

2、HandlerAdapter
initHandlerAdapters(context)，适配器。因为SpringMVC中的Handler可以是任意的形式，只要能处理请求就ok，但是Servlet需要的处理方法的结构却是固定的，都是以request和response为参数的方法。如何让固定的Servlet处理方法调用灵活的Handler来进行处理呢？这就是HandlerAdapter要做的事情。
Handler是用来干活的工具；HandlerMapping用于根据需要干的活找到相应的工具；HandlerAdapter是使用工具干活的人。

3、HandlerExceptionResolver
initHandlerExceptionResolvers(context)， 其它组件都是用来干活的。在干活的过程中难免会出现问题，出问题后怎么办呢？这就需要有一个专门的角色对异常情况进行处理，在SpringMVC中就是HandlerExceptionResolver。具体来说，此组件的作用是根据异常设置ModelAndView，之后再交给render方法进行渲染。

4、ViewResolver
initViewResolvers(context)，ViewResolver用来将String类型的视图名和Locale解析为View类型的视图。View是用来渲染页面的，也就是将程序返回的参数填入模板里，生成html（也可能是其它类型）文件。这里就有两个关键问题：使用哪个模板？用什么技术（规则）填入参数？这其实是ViewResolver主要要做的工作，ViewResolver需要找到渲染所用的模板和所用的技术（也就是视图的类型）进行渲染，具体的渲染过程则交由不同的视图自己完成。

5、RequestToViewNameTranslator
initRequestToViewNameTranslator(context)，ViewResolver是根据ViewName查找View，但有的Handler处理完后并没有设置View也没有设置ViewName，这时就需要从request获取ViewName了，如何从request中获取ViewName就是RequestToViewNameTranslator要做的事情了。RequestToViewNameTranslator在Spring MVC容器里只可以配置一个，所以所有request到ViewName的转换规则都要在一个Translator里面全部实现。

6、LocaleResolver
initLocaleResolver(context)， 解析视图需要两个参数：一是视图名，另一个是Locale。视图名是处理器返回的，Locale是从哪里来的？这就是LocaleResolver要做的事情。LocaleResolver用于从request解析出Locale，Locale就是zh-cn之类，表示一个区域，有了这个就可以对不同区域的用户显示不同的结果。SpringMVC主要有两个地方用到了Locale：一是ViewResolver视图解析的时候；二是用到国际化资源或者主题的时候。

7、ThemeResolver
initThemeResolver(context)，用于解析主题。SpringMVC中一个主题对应一个properties文件，里面存放着跟当前主题相关的所有资源、如图片、css样式等。SpringMVC的主题也支持国际化，同一个主题不同区域也可以显示不同的风格。SpringMVC中跟主题相关的类有 ThemeResolver、ThemeSource和Theme。主题是通过一系列资源来具体体现的，要得到一个主题的资源，首先要得到资源的名称，这是ThemeResolver的工作。然后通过主题名称找到对应的主题（可以理解为一个配置）文件，这是ThemeSource的工作。最后从主题中获取资源就可以了。

8、MultipartResolver
initMultipartResolver(context)，用于处理上传请求。处理方法是将普通的request包装成MultipartHttpServletRequest，后者可以直接调用getFile方法获取File，如果上传多个文件，还可以调用getFileMap得到FileName->File结构的Map。此组件中一共有三个方法，作用分别是判断是不是上传请求，将request包装成MultipartHttpServletRequest、处理完后清理上传过程中产生的临时资源。

9、FlashMapManager
initFlashMapManager(context)，用来管理FlashMap的，FlashMap主要用在redirect中传递参数。

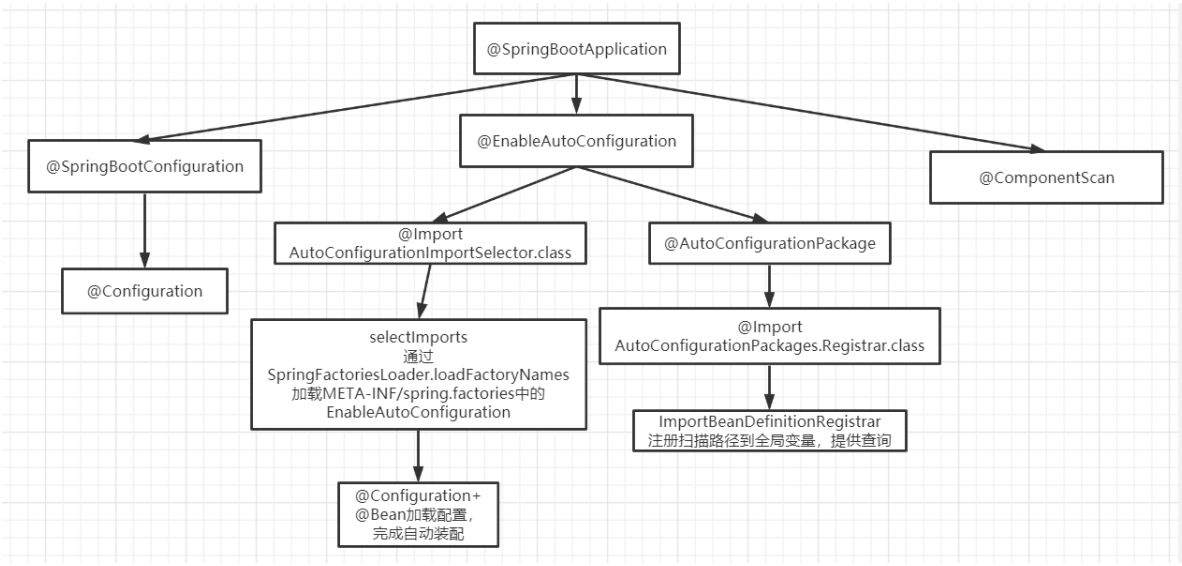
Spring Boot 自动配置原理?

@Import + @Configuration + Spring spi

自动配置类由各个starter提供，使用@Configuration + @Bean定义配置类，放到META-INF/spring.factories下

使用Spring spi扫描META-INF/spring.factories下的配置类

使用@Import导入自动配置类



如何理解 Spring Boot 中的 Starter

使用spring + springmvc使用，如果需要引入mybatis等框架，需要到xml中定义mybatis需要的bean

starter就是定义一个starter的jar包，写一个@Configuration配置类、将这些bean定义在里面，然后在starter包的META-INF/spring.factories中写入该配置类，springboot会按照约定来加载该配置类

开发人员只需要将相应的starter包依赖进应用，进行相应的属性配置（使用默认配置时，不需要配置），就可以直接进行代码开发，使用对应的功能了，比如mybatis-spring-boot--starter，spring-boot-starter-redis

什么是嵌入式服务器？为什么要使用嵌入式服务器?

节省了下载安装tomcat，应用也不需要再打war包，然后放到webapp目录下再运行

只需要一个安装了 Java 的虚拟机，就可以直接在上面部署应用程序了

springboot已经内置了tomcat.jar，运行main方法时会去启动tomcat，并利用tomcat的spi机制加载springmvc

Mybatis

mybatis的优缺点

优点：

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

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

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

4、能够与 Spring 很好的集成；

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

缺点：

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

2、SQL 语句依赖于数据库， 导致数据库移植性差， 不能随意更换数据库。

MyBatis 与Hibernate 有哪些不同?

SQL 和 ORM 的争论，永远都不会终止

开发速度的对比：
Hibernate的真正掌握要比Mybatis难些。Mybatis框架相对简单很容易上手，但也相对简陋些。
比起两者的开发速度，不仅仅要考虑到两者的特性及性能，更要根据项目需求去考虑究竟哪一个更适合项目开发，比如：一个项目中用到的复杂查询基本没有，就是简单的增删改查，这样选择hibernate效率就很快了，因为基本的sql语句已经被封装好了，根本不需要你去写sql语句，这就节省了大量的时间，但是对于一个大型项目，复杂语句较多，这样再去选择hibernate就不是一个太好的选择，选择mybatis就会加快许多，而且语句的管理也比较方便。

开发工作量的对比：
Hibernate和MyBatis都有相应的代码生成工具。可以生成简单基本的DAO层方法。针对高级查询，Mybatis需要手动编写SQL语句，以及ResultMap。而Hibernate有良好的映射机制，开发者无需关心SQL的生成与结果映射，可以更专注于业务流程

sql优化方面：
Hibernate的查询会将表中的所有字段查询出来，这一点会有性能消耗。Hibernate也可以自己写SQL来指定需要查询的字段，但这样就破坏了Hibernate开发的简洁性。而Mybatis的SQL是手动编写的，所以可以按需求指定查询的字段。
Hibernate HQL语句的调优需要将SQL打印出来，而Hibernate的SQL被很多人嫌弃因为太丑了。
MyBatis的SQL是自己手动写的所以调整方便。但Hibernate具有自己的日志统计。Mybatis本身不带日志统计，使用Log4j进行日志记录。

对象管理的对比：
Hibernate 是完整的对象/关系映射解决方案，它提供了对象状态管理（state management）的功能，使开发者不再需要理会底层数据库系统的细节。也就是说，相对于常见的 JDBC/SQL 持久层方案中需要管理 SQL 语句，Hibernate采用了更自然的面向对象的视角来持久化 Java 应用中的数据。
换句话说，使用 Hibernate 的开发者应该总是关注对象的状态（state），不必考虑 SQL 语句的执行。这部分细节已经由 Hibernate 掌管妥当，只有开发者在进行系统性能调优的时候才需要进行了解。而MyBatis在这一块没有文档说明，用户需要对对象自己进行详细的管理。

缓存机制对比：

相同点：都可以实现自己的缓存或使用其他第三方缓存方案，创建适配器来完全覆盖缓存行为。

不同点：Hibernate的二级缓存配置在SessionFactory生成的配置文件中进行详细配置，然后再在具体的表-对象映射中配置是哪种缓存。

MyBatis的二级缓存配置都是在每个具体的表-对象映射中进行详细配置，这样针对不同的表可以自定义不同的缓存机制。并且Mybatis可以在命名空间中共享相同的缓存配置和实例，通过Cache-ref来实现。

两者比较：因为Hibernate对查询对象有着良好的管理机制，用户无需关心SQL。所以在使用二级缓存时如果出现脏数据，系统会报出错误并提示。

而MyBatis在这一方面，使用二级缓存时需要特别小心。如果不能完全确定数据更新操作的波及范围，避免Cache的盲目使用。否则，脏数据的出现会给系统的正常运行带来很大的隐患。

Hibernate功能强大，数据库无关性好，O/R映射能力强，如果你对Hibernate相当精通，而且对Hibernate进行了适当的封装，那么你的项目整个持久层代码会相当简单，需要写的代码很少，开发速度很快，非常爽。

Hibernate的缺点就是学习门槛不低，要精通门槛更高，而且怎么设计O/R映射，在性能和对象模型之间如何权衡取得平衡，以及怎样用好Hibernate方面需要你的经验和能力都很强才行。

iBATIS入门简单，即学即用，提供了数据库查询的自动对象绑定功能，而且延续了很好的SQL使用经验，对于没有那么高的对象模型要求的项目来说，相当完美。

iBATIS的缺点就是框架还是比较简陋，功能尚有缺失，虽然简化了数据绑定代码，但是整个底层数据库查询实际还是要自己写的，工作量也比较大，而且不太容易适应快速数据库修改。

#{}和${}的区别是什么？

#{}是预编译处理、是占位符， ${}是字符串替换、是拼接符。

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

Mybatis 在处理${}时， 就是把${}替换成变量的值，调用 Statement 来赋值；

#{} 的变量替换是在DBMS 中、变量替换后，#{} 对应的变量自动加上单引号

${} 的变量替换是在 DBMS 外、变量替换后，${} 对应的变量不会加上单引号

使用#{}可以有效的防止 SQL 注入， 提高系统安全性。

简述 Mybatis 的插件运行原理，如何编写一个插件。

答： Mybatis 只支持针对 ParameterHandler、ResultSetHandler、StatementHandler、Executor 这 4 种接口的插件， Mybatis 使用 JDK 的动态代理， 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， 拦截那些你指定需要拦截的方法。

编写插件： 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， 在配置文件中配置编写的插件。

```
@Intercepts({@Signature(type = StatementHandler.class, method = "query", args =
{Statement.class, ResultHandler.class}),
        @Signature(type = StatementHandler.class, method = "update", args =
{Statement.class}),
        @Signature(type = StatementHandler.class, method = "batch", args = {
Statement.class })})
@Component


invocation.proceed()执行具体的业务逻辑
```

Mysql

索引的基本原理

索引用来快速地寻找那些具有特定值的记录。如果没有索引，一般来说执行查询时遍历整张表。

索引的原理：就是把无序的数据变成有序的查询

1. 把创建了索引的列的内容进行排序
2. 对排序结果生成倒排表
3. 在倒排表内容上拼上数据地址链
4. 在查询的时候，先拿到倒排表内容，再取出数据地址链，从而拿到具体数据

mysql聚簇和非聚簇索引的区别

都是B+树的数据结构

- 聚簇索引：将数据存储与索引放到了一块、并且是按照一定的顺序组织的，找到索引也就找到了数据，数据的物理存放顺序与索引顺序是一致的，即：只要索引是相邻的，那么对应的数据一定也是相邻地存放在磁盘上的
- 非聚簇索引：叶子节点不存储数据、存储的是数据行地址，也就是说根据索引查找到数据行的位置再取磁盘查找数据，这个就有点类似一本树的目录，比如我们要找第三章第一节，那我们先在这个目录里面找，找到对应的页码后再去对应的页码看文章。

> 优势：
> 1、查询通过聚簇索引可以直接获取数据，相比非聚簇索引需要第二次查询（非覆盖索引的情况下）效率要高
> 2、聚簇索引对于范围查询的效率很高，因为其数据是按照大小排列的
> 3、聚簇索引适合用在排序的场合，非聚簇索引不适合
>
> 劣势：
> 1、维护索引很昂贵，特别是插入新行或者主键被更新导至要分页(page split)的时候。建议在大量插入新行后，选在负载较低的时间段，通过OPTIMIZE TABLE优化表，因为必须被移动的行数据可能造成碎片。使用独享表空间可以弱化碎片
> 2、表因为使用UUId（随机ID）作为主键，使数据存储稀疏，这就会出现聚簇索引有可能有比全表扫面更慢，所以建议使用int的auto_increment作为主键
> 3、如果主键比较大的话，那辅助索引将会变的更大，因为辅助索引的叶子存储的是主键值；过长的主键值，会导致非叶子节点占用占用更多的物理空间

InnoDB中一定有主键，主键一定是聚簇索引，不手动设置、则会使用unique索引，没有unique索引，则会使用数据库内部的一个行的隐藏id来当作主键索引。在聚簇索引之上创建的索引称之为辅助索引，辅助索引访问数据总是需要二次查找，非聚簇索引都是辅助索引，像复合索引、前缀索引、唯一索引，辅助索引叶子节点存储的不再是行的物理位置，而是主键值

MyISM使用的是非聚簇索引，没有聚簇索引，非聚簇索引的两棵B+树看上去没什么不同，节点的结构完全一致只是存储的内容不同而已，主键索引B+树的节点存储了主键，辅助键索引B+树存储了辅助键。表数据存储在独立的地方，这两颗B+树的叶子节点都使用一个地址指向真正的表数据，对于表数据来说，这两个键没有任何差别。由于索引树是独立的，通过辅助键检索无需访问主键的索引树。

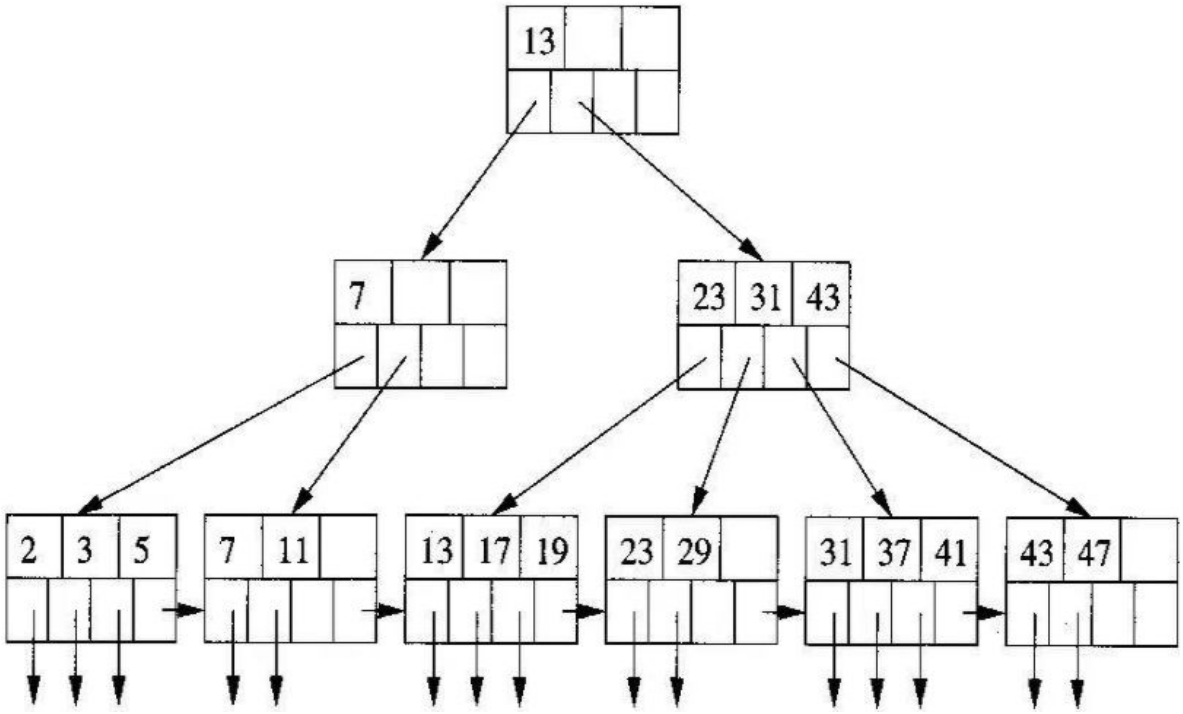
如果涉及到大数据量的排序、全表扫描、count之类的操作的话，还是MyISAM占优势些，因为索引所占空间小，这些操作是需要在内存中完成的。

mysql索引的数据结构，各自优劣

索引的数据结构和具体存储引擎的实现有关，在MySQL中使用较多的索引有Hash索引，B+树索引等，InnoDB存储引擎的默认索引实现为：B+树索引。对于哈希索引来说，底层的数据结构就是哈希表，因此在绝大多数需求为单条记录查询的时候，可以选择哈希索引，查询性能最快；其余大部分场景，建议选择BTree索引。

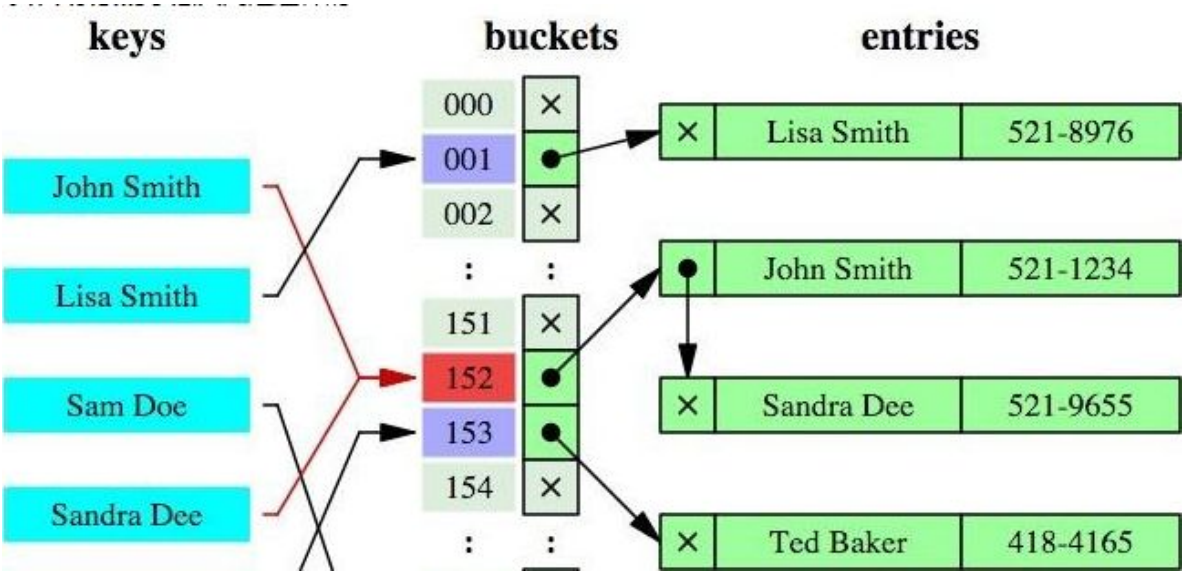
B+树：

B+树是一个平衡的多叉树，从根节点到每个叶子节点的高度差值不超过1，而且同层级的节点间有指针相互链接。在B+树上的常规检索，从根节点到叶子节点的搜索效率基本相当，不会出现大幅波动，而且基于索引的顺序扫描时，也可以利用双向指针快速左右移动，效率非常高。因此，B+树索引被广泛应用于数据库、文件系统等场景。



哈希索引：

哈希索引就是采用一定的哈希算法，把键值换算成新的哈希值，检索时不需要类似B+树那样从根节点到叶子节点逐级查找，只需一次哈希算法即可立刻定位到相应的位置，速度非常快



如果是等值查询，那么哈希索引明显有绝对优势，因为只需要经过一次算法即可找到相应的键值；前提是键值都是唯一的。如果键值不是唯一的，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应的数据；

如果是范围查询检索，这时候哈希索引就毫无用武之地了，因为原先是有序的键值，经过哈希算法后，有可能变成不连续的了，就没办法再利用索引完成范围查询检索；

哈希索引也没办法利用索引完成排序，以及like ‘xxx%’ 这样的部分模糊查询（这种部分模糊查询，其实本质上也是范围查询）；

哈希索引也不支持多列联合索引的最左匹配规则；

B+树索引的关键字检索效率比较平均，不像B树那样波动幅度大，在有大量重复键值情况下，哈希索引的效率也是极低的，因为存在哈希碰撞问题。

索引设计的原则?

查询更快、占用空间更小

1. 适合索引的列是出现在where子句中的列，或者连接子句中指定的列
2. 基数较小的表，索引效果较差，没有必要在此列建立索引
3. 使用短索引，如果对长字符串列进行索引，应该指定一个前缀长度，这样能够节省大量索引空间，如果搜索词超过索引前缀长度，则使用索引排除不匹配的行，然后检查其余行是否可能匹配。
4. 不要过度索引。索引需要额外的磁盘空间，并降低写操作的性能。在修改表内容的时候，索引会进行更新甚至重构，索引列越多，这个时间就会越长。所以只保持需要的索引有利于查询即可。
5. 定义有外键的数据列一定要建立索引。
6. 更新频繁字段不适合创建索引
7. 若是不能有效区分数据的列不适合做索引列(如性别，男女未知，最多也就三种，区分度实在太低)

8. 尽量的扩展索引，不要新建索引。比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可。
9. 对于那些查询中很少涉及的列，重复值比较多的列不要建立索引。
10. 对于定义为text、image和bit的数据类型的列不要建立索引。

什么是最左前缀原则？什么是最左匹配原则

锁的类型有哪些

基于锁的属性分类：共享锁、排他锁。

基于锁的粒度分类：行级锁(INNODB)、表级锁(INNODB、MYISAM)、页级锁(BDB引擎)、记录锁、间隙锁、临键锁。

基于锁的状态分类：意向共享锁、意向排它锁。

- 共享锁(Share Lock)

  > 共享锁又称读锁，简称S锁；当一个事务为数据加上读锁之后，其他事务只能对该数据加读锁，而不能对数据加写锁，直到所有的读锁释放之后其他事务才能对其进行加持写锁。共享锁的特性主要是为了支持并发的读取数据，读取数据的时候不支持修改，避免出现重复读的问题。

- 排他锁（eXclusive Lock）

  > 排他锁又称写锁，简称X锁；当一个事务为数据加上写锁时，其他请求将不能再为数据加任何锁，直到该锁释放之后，其他事务才能对数据进行加锁。排他锁的目的是在数据修改时候，不允许其他人同时修改，也不允许其他人读取。避免了出现脏数据和脏读的问题。

- 表锁

  > 表锁是指上锁的时候锁住的是整个表，当下一个事务访问该表的时候，必须等前一个事务释放了锁才能进行对表进行访问；
  >
  > 特点： 粒度大，加锁简单，容易冲突；

- 行锁

> 行锁是指上锁的时候锁住的是表的某一行或多行记录，其他事务访问同一张表时，只有被锁住的记录不能访问，其他的记录可正常访问；
>
> 特点：粒度小，加锁比表锁麻烦，不容易冲突，相比表锁支持的并发要高；

- 记录锁(Record Lock)

> 记录锁也属于行锁中的一种，只不过记录锁的范围只是表中的某一条记录，记录锁是说事务在加锁后锁住的只是表的某一条记录。
> 精准条件命中，并且命中的条件字段是唯一索引
> 加了记录锁之后数据可以避免数据在查询的时候被修改的重复读问题，也避免了在修改的事务未提交前被其他事务读取的脏读问题。

- 页锁

> 页级锁是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。
> 特点：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般

- 间隙锁(Gap Lock)

> 属于行锁中的一种，间隙锁是在事务加锁后其锁住的是表记录的某一个区间，当表的相邻ID之间出现空隙则会形成一个区间，遵循左开右闭原则。
> 范围查询并且查询未命中记录，查询条件必须命中索引、间隙锁只会出现在REPEATABLE_READ（重复读)的事务级别中。
> 触发条件：防止幻读问题，事务并发的时候，如果没有间隙锁，就会发生如下图的问题，在同一个事务里，A事务的两次查询出的结果会不一样。
>
> 比如表里面的数据ID 为 1,4,5,7,10 ,那么会形成以下几个间隙区间，-n-1区间，1-4区间，7-10区间，10-n区间 （-n代表负无穷大，n代表正无穷大）

- 临建锁(Next-Key Lock)

> 也属于行锁的一种，并且它是INNODB的行锁默认算法，总结来说它就是记录锁和间隙锁的组合，临键锁会把查询出来的记录锁住，同时也会把该范围查询内的所有间隙空间也会锁住，再之它会把相邻的下一个区间也会锁住
> 触发条件：范围查询并命中，查询命中了索引。
> 结合记录锁和间隙锁的特性，临键锁避免了在范围查询时出现脏读、重复读、幻读问题。加了临键锁之后，在范围区间内数据不允许被修改和插
> 入。

如果当事务A加锁成功之后就设置一个状态告诉后面的人，已经有人对表里的行加了一个排他锁了，你们不能对整个表加共享锁或排它锁了，那么后面需要对整个表加锁的人只需要获取这个状态就知道自己是不是可以对表加锁，避免了对整个索引树的每个节点扫描是否加锁，而这个状态就是意向锁。

- 意向共享锁

> 当一个事务试图对整个表进行加共享锁之前，首先需要获得这个表的意向共享锁。

- 意向排他锁

> 当一个事务试图对整个表进行加排它锁之前，首先需要获得这个表的意向排它锁。

InnoDB存储引擎的锁的算法

- Record lock：单个行记录上的锁
- Gap lock：间隙锁，锁定一个范围，不包括记录本身
- Next-key lock：record+gap 锁定一个范围，包含记录本身

相关知识点：

1. innodb对于行的查询使用next-key lock
2. Next-locking keying为了解决Phantom Problem幻读问题
3. 当查询的索引含有唯一属性时，将next-key lock降级为record key
4. Gap锁设计的目的是为了阻止多个事务将记录插入到同一范围内，而这会导致幻读问题的产生
5. 有两种方式显式关闭gap锁：（除了外键约束和唯一性检查外，其余情况仅使用record lock） A. 将事务隔离级别设置为RC B. 将参数innodb_locks_unsafe_for_binlog设置为1

关心过业务系统里面的sql耗时吗？统计过慢查询吗？对慢查询都怎么优化过？

在业务系统中，除了使用主键进行的查询，其他的都会在测试库上测试其耗时，慢查询的统计主要由运维在做，会定期将业务中的慢查询反馈给我们。

慢查询的优化首先要搞明白慢的原因是什么？是查询条件没有命中索引？是load了不需要的数据列？还是数据量太大？

所以优化也是针对这三个方向来的，

- 首先分析语句，看看是否load了额外的数据，可能是查询了多余的行并且抛弃掉了，可能是加载了许多结果中并不需要的列，对语句进行分析以及重写。
- 分析语句的执行计划，然后获得其使用索引的情况，之后修改语句或者修改索引，使得语句可以尽可能的命中索引。
- 如果对语句的优化已经无法进行，可以考虑表中的数据量是否太大，如果是的话可以进行横向或者纵向的分表。

事务的基本特性和隔离级别

事务基本特性ACID分别是：

原子性指的是一个事务中的操作要么全部成功，要么全部失败。

一致性指的是数据库总是从一个一致性的状态转换到另外一个一致性的状态。比如A转账给B100块钱，假设A只有90块，支付之前我们数据库里的数据都是符合约束的,但是如果事务执行成功了,我们的数据库数据就破坏约束了,因此事务不能成功,这里我们说事务提供了一致性的保证

隔离性指的是一个事务的修改在最终提交前，对其他事务是不可见的。

持久性指的是一旦事务提交，所做的修改就会永久保存到数据库中。

隔离性有4个隔离级别，分别是：

- read uncommit 读未提交，可能会读到其他事务未提交的数据，也叫做脏读。

  用户本来应该读取到id=1的用户age应该是10，结果读取到了其他事务还没有提交的事务，结果读取结果age=20，这就是脏读。

- read commit 读已提交，两次读取结果不一致，叫做不可重复读。

  不可重复读解决了脏读的问题，他只会读取已经提交的事务。

  用户开启事务读取id=1用户，查询到age=10，再次读取发现结果=20，在同一个事务里同一个查询读取到不同的结果叫做不可重复读。

- repeatable read 可重复复读，这是mysql的默认级别，就是每次读取结果都一样，但是有可能产生幻读。
- serializable 串行，一般是不会使用的，他会给每一行读取的数据加锁，会导致大量超时和锁竞争的问题。

脏读(Drity Read)：某个事务已更新一份数据，另一个事务在此时读取了同一份数据，由于某些原因，前一个RollBack了操作，则后一个事务所读取的数据就会是不正确的。

不可重复读(Non-repeatable read):在一个事务的两次查询之中数据不一致，这可能是两次查询过程中间插入了一个事务更新的原有的数据。

幻读(Phantom Read):在一个事务的两次查询中数据笔数不一致，例如有一个事务查询了几列(Row)数据，而另一个事务却在此时插入了新的几列数据，先前的事务在接下来的查询中，就会发现有几列数据是它先前所没有的。

ACID靠什么保证的?

A原子性由undo log日志保证，它记录了需要回滚的日志信息，事务回滚时撤销已经执行成功的sql

C一致性由其他三大特性保证、程序代码要保证业务上的一致性

I隔离性由MVCC来保证

D持久性由内存+redo log来保证，mysql修改数据同时在内存和redo log记录这次操作，宕机的时候可以从redo log恢复

```
InnoDB redo log 写盘，InnoDB 事务进入 prepare 状态。
如果前面 prepare 成功，binlog 写盘，再继续将事务日志持久化到 binlog，如果持久化成功，那么
InnoDB 事务则进入 commit 状态(在 redo log 里面写一个 commit 记录)
```

redolog的刷盘会在系统空闲时进行

什么是MVCC

多版本并发控制：读取数据时通过一种类似快照的方式将数据保存下来，这样读锁就和写锁不冲突了，不同的事务session会看到自己特定版本的数据，版本链

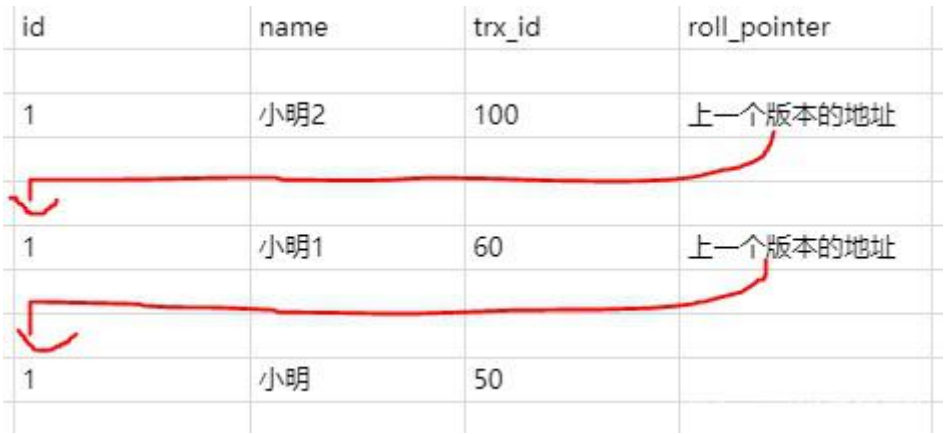
MVCC只在 READ COMMITTED 和 REPEATABLE READ 两个隔离级别下工作。其他两个隔离级别够和MVCC不兼容, 因为 READ UNCOMMITTED 总是读取最新的数据行, 而不是符合当前事务版本的数据行。而 SERIALIZABLE 则会对所有读取的行都加锁。

聚簇索引记录中有两个必要的隐藏列：

trx_id：用来存储每次对某条聚簇索引记录进行修改的时候的事务id。

roll_pointer：每次对哪条聚簇索引记录有修改的时候，都会把老版本写入undo日志中。这个roll_pointer就是存了一个指针，它指向这条聚簇索引记录的上一个版本的位置，通过它来获得上一个版本的记录信息。(注意插入操作的undo日志没有这个属性，因为它没有老版本)

已提交读和可重复读的区别就在于它们生成ReadView的策略不同。



开始事务时创建readview，readView维护当前活动的事务id，即未提交的事务id，排序生成一个数组

访问数据，获取数据中的事务id（获取的是事务id最大的记录），对比readview：

如果在readview的左边（比readview都小），可以访问（在左边意味着该事务已经提交）

如果在readview的右边（比readview都大）或者就在readview中，不可以访问，获取roll_pointer，取上一版本重新对比（在右边意味着，该事务在readview生成之后出现，在readview中意味着该事务还未提交）

已提交读隔离级别下的事务在每次查询的开始都会生成一个独立的ReadView,而可重复读隔离级别则在第一次读的时候生成一个ReadView，之后的读都复用之前的ReadView。

这就是Mysql的MVCC,通过版本链，实现多版本，可并发读-写，写-读。通过ReadView生成策略的不同实现不同的隔离级别。

分表后非sharding_key的查询怎么处理，分表后的排序?

1. 可以做一个mapping表，比如这时候商家要查询订单列表怎么办呢？不带user_id查询的话你总不能扫全表吧？所以我们可以做一个映射关系表，保存商家和用户的关系，查询的时候先通过商家查询到用户列表，再通过user_id去查询。
2. 宽表，对数据实时性要求不是很高的场景，比如查询订单列表，可以把订单表同步到离线（实时）数仓，再基于数仓去做成一张宽表，再基于其他如es提供查询服务。
3. 数据量不是很大的话，比如后台的一些查询之类的，也可以通过多线程扫表，然后再聚合结果的方式来做。或者异步的形式也是可以的。

union

排序字段是唯一索引：

- 首先第一页的查询：将各表的结果集进行合并，然后再次排序
- 第二页及以后的查询，需要传入上一页排序字段的最后一个值，及排序方式。
- 根据排序方式，及这个值进行查询。如排序字段date，上一页最后值为3，排序方式降序。查询的时候sql为select ... from table where date < 3 order by date desc limit 0,10。这样再将几个表的结果合并排序即可。

mysql主从同步原理

mysql主从同步的过程：

Mysql的主从复制中主要有三个线程：`master（binlog dump thread）、slave（I/O thread 、SQL thread）`，Master一条线程和Slave中的两条线程。

- 主节点 binlog，主从复制的基础是主库记录数据库的所有变更记录到 binlog。binlog 是数据库服务器启动的那一刻起，保存所有修改数据库结构或内容的一个文件。
- 主节点 log dump 线程，当 binlog 有变动时，log dump 线程读取其内容并发送给从节点。
- 从节点 I/O线程接收 binlog 内容，并将其写入到 relay log 文件中。
- 从节点的SQL 线程读取 relay log 文件内容对数据更新进行重放，最终保证主从数据库的一致性。

注：主从节点使用 binglog 文件 + position 偏移量来定位主从同步的位置，从节点会保存其已接收到的偏移量，如果从节点发生宕机重启，则会自动从 position 的位置发起同步。

由于mysql默认的复制方式是异步的，主库把日志发送给从库后不关心从库是否已经处理，这样会产生一个问题就是假设主库挂了，从库处理失败了，这时候从库升为主库后，日志就丢失了。由此产生两个概念。

全同步复制

主库写入binlog后强制同步日志到从库，所有的从库都执行完成后才返回给客户端，但是很显然这个方式的话性能会受到严重影响。

半同步复制

和全同步不同的是，半同步复制的逻辑是这样，从库写入日志成功后返回ACK确认给主库，主库收到至少一个从库的确认就认为写操作完成。

简述MyISAM和InnoDB的区别

MyISAM：

不支持事务，但是每次查询都是原子的；

支持表级锁，即每次操作是对整个表加锁；

存储表的总行数；

一个MYISAM表有三个文件：索引文件、表结构文件、数据文件；

采用非聚集索引，索引文件的数据域存储指向数据文件的指针。辅索引与主索引基本一致，但是辅索引不用保证唯一性。

InnoDb：

支持ACID的事务，支持事务的四种隔离级别；

支持行级锁及外键约束：因此可以支持写并发；

不存储总行数；

一个InnoDb引擎存储在一个文件空间（共享表空间，表大小不受操作系统控制，一个表可能分布在多个文件里），也有可能为多个（设置为独立表空，表大小受操作系统文件大小限制，一般为2G），受操作系统文件大小的限制；

主键索引采用聚集索引（索引的数据域存储数据文件本身），辅索引的数据域存储主键的值；因此从辅索引查找数据，需要先通过辅索引找到主键值，再访问辅索引；最好使用自增主键，防止插入数据时，为维持B+树结构，文件的大调整。

简述mysql中索引类型及对数据库的性能的影响

普通索引：允许被索引的数据列包含重复的值。

唯一索引：可以保证数据记录的唯一性。

主键：是一种特殊的唯一索引，在一张表中只能定义一个主键索引，主键用于唯一标识一条记录，使用关键字 PRIMARY KEY 来创建。

联合索引：索引可以覆盖多个数据列，如像INDEX(columnA, columnB)索引。

全文索引：通过建立`倒排索引`,可以极大的提升检索效率,解决判断字段是否包含的问题，是目前搜索引擎使用的一种关键技术。可以通过ALTER TABLE table_name ADD FULLTEXT (column);创建全文索引

索引可以极大的提高数据的查询速度。

通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

但是会降低插入、删除、更新表的速度，因为在执行这些写操作时，还要操作索引文件

索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大，如果非聚集索引很多，一旦聚集索引改变，那么所有非聚集索引都会跟着变。

mysql执行计划怎么看

执行计划就是sql的执行查询的顺序，以及如何使用索引查询，返回的结果集的行数

EXPLAIN SELECT * from A where X=? and Y=?

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra

1。id ：是一个有顺序的编号，是查询的顺序号，有几个 select 就显示几行。id的顺序是按 select 出现的顺序增长的。id列的值越大执行优先级越高越先执行，id列的值相同则从上往下执行，id列的值为NULL最后执行。

2。selectType 表示查询中每个select子句的类型

- SIMPLE： 表示此查询不包含 UNION 查询或子查询
- PRIMARY： 表示此查询是最外层的查询（包含子查询）
- SUBQUERY： 子查询中的第一个 SELECT
- UNION： 表示此查询是 UNION 的第二或随后的查询
- DEPENDENT UNION： UNION 中的第二个或后面的查询语句, 取决于外面的查询
- UNION RESULT, UNION 的结果
- DEPENDENT SUBQUERY: 子查询中的第一个 SELECT, 取决于外面的查询. 即子查询依赖于外层查询的结果.
- DERIVED：衍生，表示导出表的SELECT（FROM子句的子查询）

3.table：表示该语句查询的表

4.type：优化sql的重要字段，也是我们判断sql性能和优化程度重要指标。他的取值类型范围：

- const：通过索引一次命中，匹配一行数据
- system: 表中只有一行记录，相当于系统表；
- eq_ref：唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配
- ref: 非唯一性索引扫描,返回匹配某个值的所有
- range: 只检索给定范围的行，使用一个索引来选择行，一般用于between、<、>；
- index: 只遍历索引树；
- ALL: 表示全表扫描，这个类型的查询是性能最差的查询之一。 那么基本就是随着表的数量增多，执行效率越慢。

执行效率：

ALL < index < range< ref < eq_ref < const < system。最好是避免ALL和index

5.possible_keys：它表示Mysql在执行该sql语句的时候，可能用到的索引信息，仅仅是可能，实际不一定会用到。

6.key：此字段是 mysql 在当前查询时所真正使用到的索引。 他是possible_keys的子集

7.key_len：表示查询优化器使用了索引的字节数，这个字段可以评估组合索引是否完全被使用，这也是我们优化sql时，评估索引的重要指标

9.rows：mysql 查询优化器根据统计信息，估算该sql返回结果集需要扫描读取的行数，这个值相关重要，索引优化之后，扫描读取的行数越多，说明索引设置不对，或者字段传入的类型之类的问题，说明要优化空间越大

10.filtered：返回结果的行占需要读到的行(rows列的值)的百分比，就是百分比越高，说明需要查询到数据越准确， 百分比越小，说明查询到的数据量大，而结果集很少

11.extra

- using filesort ： 表示 mysql 对结果集进行外部排序，不能通过索引顺序达到排序效果。一般有 using filesort都建议优化去掉，因为这样的查询 cpu 资源消耗大，延时大。
- using index：覆盖索引扫描，表示查询在索引树中就可查找所需数据，不用扫描表数据文件，往往说明性能不错。

- using temporary：查询有使用临时表, 一般出现于排序， 分组和多表 join 的情况， 查询效率不高，建议优化。
- using where ： sql使用了where过滤,效率较高。

redis

RDB 和 AOF 机制

RDB：Redis DataBase

在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。

优点：

1、整个Redis数据库将只包含一个文件 dump.rdb，方便持久化。

2、容灾性好，方便备份。

3、性能最大化，fork 子进程来完成写操作，让主进程继续处理命令，所以是 IO 最大化。使用单独子进程来进行持久化，主进程不会进行任何 IO 操作，保证了 redis 的高性能

4.相对于数据集大时，比 AOF 的启动效率更高。

缺点：

1、数据安全性低。RDB 是间隔一段时间进行持久化，如果持久化之间 redis 发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候)

2、由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

AOF：Append Only File

以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录，可以打开文件看到详细的操作记录

- **优点：**

1、数据安全，Redis中提供了3中同步策略，即每秒同步、每修改同步和不同步。事实上，每秒同步也是异步完成的，其效率也是非常高的，所差的是一旦系统出现宕机现象，那么这一秒钟之内修改的数据将会丢失。而每修改同步，我们可以将其视为同步持久化，即每次发生的数据变化都会被立即记录到磁盘中。。

2、通过 append 模式写文件，即使中途服务器宕机也不会破坏已经存在的内容，可以通过 redis-check-aof 工具解决数据一致性问题。

3、AOF 机制的 rewrite 模式。定期对AOF文件进行重写，以达到压缩的目的

- **缺点：**

1、AOF 文件比 RDB 文件大，且恢复速度慢。

2、数据集大的时候，比 rdb 启动效率低。

3、运行效率没有RDB高

AOF文件比RDB更新频率高，优先使用AOF还原数据。

AOF比RDB更安全也更大

RDB性能比AOF好

如果两个都配了优先加载AOF

Redis的过期键的删除策略

Redis是key-value数据库，我们可以设置Redis中缓存的key的过期时间。Redis的过期策略就是指当Redis中缓存的key过期了，Redis如何处理。

- **惰性过期**：只有当访问一个key时，才会判断该key是否已过期，过期则清除。该策略可以最大化地节省CPU资源，却对内存非常不友好。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。
- **定期过期**：每隔一定的时间，会扫描一定数量的数据库的expires字典中一定数量的key，并清除其中已过期的key。该策略是一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。

(expires字典会保存所有设置了过期时间的key的过期时间数据，其中，key是指向键空间中的某个键的指针，value是该键的毫秒精度的UNIX时间戳表示的过期时间。键空间是指该Redis集群中保存的所有键。)

Redis中同时使用了惰性过期和定期过期两种过期策略。

Redis线程模型、单线程快的原因

Redis基于Reactor模式开发了网络事件处理器，这个处理器叫做文件事件处理器 file event handler。这个文件事件处理器，它是单线程的，所以 Redis 才叫做单线程的模型，它采用IO多路复用机制来同时监听多个Socket，根据Socket上的事件类型来选择对应的事件处理器来处理这个事件。可以实现高性能的网络通信模型，又可以跟内部其他单线程的模块进行对接，保证了 Redis 内部的线程模型的简单性。

文件事件处理器的结构包含4个部分：多个Socket、IO多路复用程序、文件事件分派器以及事件处理器（命令请求处理器、命令回复处理器、连接应答处理器等）。
多个 Socket 可能并发的产生不同的操作，每个操作对应不同的文件事件，但是IO多路复用程序会监听多个 Socket，会将 Socket 放入一个队列中排队，每次从队列中取出一个 Socket 给事件分派器，事件分派器把 Socket 给对应的事件处理器。
然后一个 Socket 的事件处理完之后，IO多路复用程序才会将队列中的下一个 Socket 给事件分派器。文件事件分派器会根据每个 Socket 当前产生的事件，来选择对应的事件处理器来处理。

单线程快的原因：

1）纯内存操作

2）核心是基于非阻塞的IO多路复用机制

3）单线程反而避免了多线程的频繁上下文切换带来的性能问题

简述Redis事务实现

1、事务开始

*MULTI*命令的执行，标识着一个事务的开始。*MULTI*命令会将客户端状态的`flags`属性中打开`REDIS_MULTI`标识来完成的。

2、命令入队

当一个客户端切换到事务状态之后，服务器会根据这个客户端发送来的命令来执行不同的操作。如果客户端发送的命令为*MULTI*、*EXEC*、*WATCH*、*DISCARD*中的一个，立即执行这个命令，否则将命令放入一个事务队列里面，然后向客户端返回`QUEUED`回复

- 如果客户端发送的命令为 EXEC、DISCARD、WATCH、MULTI 四个命令的其中一个，那么服务器立即执行这个命令。
- 如果客户端发送的是四个命令以外的其他命令，那么服务器并不立即执行这个命令。

  首先检查此命令的格式是否正确，如果不正确，服务器会在客户端状态（redisClient）的 flags 属性关闭 REDIS_MULTI 标识，并且返回错误信息给客户端。

  如果正确，将这个命令放入一个事务队列里面，然后向客户端返回 QUEUED 回复

事务队列是按照FIFO的方式保存入队的命令

3、事务执行

客户端发送 EXEC 命令，服务器执行 EXEC 命令逻辑。

- 如果客户端状态的 flags 属性不包含 REDIS_MULTI 标识，或者包含 REDIS_DIRTY_CAS 或者 REDIS_DIRTY_EXEC 标识，那么就直接取消事务的执行。
- 否则客户端处于事务状态（flags 有 REDIS_MULTI 标识），服务器会遍历客户端的事务队列，然后执行事务队列中的所有命令，最后将返回结果全部返回给客户端；

redis 不支持事务回滚机制，但是它会检查每一个事务中的命令是否错误。

Redis 事务不支持检查那些程序员自己逻辑错误。例如对 String 类型的数据库键执行对 HashMap 类型的操作!

- WATCH 命令是一个乐观锁，可以为 Redis 事务提供 check-and-set （CAS）行为。可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行，监控一直持续到EXEC命令。
- MULTI命令用于开启一个事务，它总是返回OK。MULTI执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个队列中，当EXEC命令被调用时，所有队列中的命令才会被执行。
- EXEC：执行所有事务块内的命令。返回事务块内所有命令的返回值，按命令执行的先后顺序排列。当操作被打断时，返回空值 nil 。
- 通过调用DISCARD，客户端可以清空事务队列，并放弃执行事务， 并且客户端会从事务状态中退出。
- UNWATCH命令可以取消watch对所有key的监控。

redis集群方案

主从

哨兵模式：

sentinel，哨兵是 redis 集群中非常重要的一个组件，主要有以下功能：

- 集群监控：负责监控 redis master 和 slave 进程是否正常工作。
- 消息通知：如果某个 redis 实例有故障，那么哨兵负责发送消息作为报警通知给管理员。
- 故障转移：如果 master node 挂掉了，会自动转移到 slave node 上。
- 配置中心：如果故障转移发生了，通知 client 客户端新的 master 地址。

哨兵用于实现 redis 集群的高可用，本身也是分布式的，作为一个哨兵集群去运行，互相协同工作。

- 故障转移时，判断一个 master node 是否宕机了，需要大部分的哨兵都同意才行，涉及到了分布式选举
- 即使部分哨兵节点挂掉了，哨兵集群还是能正常工作的
- 哨兵通常需要 3 个实例，来保证自己的健壮性。
- 哨兵 + redis 主从的部署架构，是不保证数据零丢失的，只能保证 redis 集群的高可用性。
- 对于哨兵 + redis 主从这种复杂的部署架构，尽量在测试环境和生产环境，都进行充足的测试和演练。

Redis Cluster是一种服务端Sharding技术，3.0版本开始正式提供。采用slot(槽)的概念，一共分成16384个槽。将请求发送到任意节点，接收到请求的节点会将查询请求发送到正确的节点上执行

方案说明

- 通过哈希的方式，将数据分片，每个节点均分存储一定哈希槽(哈希值)区间的数据，默认分配了16384 个槽位

- 每份数据分片会存储在多个互为主从的多节点上
- 数据写入先写主节点，再同步到从节点(支持配置为阻塞同步)
- 同一分片多个节点间的数据不保持强一致性
- 读取数据时，当客户端操作的key没有分配在该节点上时，redis会返回转向指令，指向正确的节点
- 扩容时需要需要把旧节点的数据迁移一部分到新节点

在 redis cluster 架构下，每个 redis 要放开两个端口号，比如一个是 6379，另外一个就是 加1w 的端口号，比如 16379。

16379 端口号是用来进行节点间通信的，也就是 cluster bus 的通信，用来进行故障检测、配置更新、故障转移授权。cluster bus 用了另外一种二进制的协议，gossip 协议，用于节点间进行高效的数据交换，占用更少的网络带宽和处理时间。

优点

- 无中心架构，支持动态扩容，对业务透明
- 具备Sentinel的监控和自动Failover(故障转移)能力
- 客户端不需要连接集群所有节点，连接集群中任何一个可用节点即可
- 高性能，客户端直连redis服务，免去了proxy代理的损耗

缺点

- 运维也很复杂，数据迁移需要人工干预
- 只能使用0号数据库
- 不支持批量操作(pipeline管道操作)
- 分布式逻辑和存储模块耦合等

Redis Sharding是Redis Cluster出来之前，业界普遍使用的多Redis实例集群方法。其主要思想是采用哈希算法将Redis数据的key进行散列，通过hash函数，特定的key会映射到特定的Redis节点上。Java redis客户端驱动jedis，支持Redis Sharding功能，即ShardedJedis以及结合缓存池的ShardedJedisPool

优点

优势在于非常简单，服务端的Redis实例彼此独立，相互无关联，每个Redis实例像单服务器一样运行，非常容易线性扩展，系统的灵活性很强

缺点

由于sharding处理放到客户端，规模进一步扩大时给运维带来挑战。

客户端sharding不支持动态增删节点。服务端Redis实例群拓扑结构有变化时，每个客户端都需要更新调整。连接不能共享，当应用规模增大时，资源浪费制约优化

redis 主从复制的核心原理

通过执行slaveof命令或设置slaveof选项，让一个服务器去复制另一个服务器的数据。主数据库可以进行读写操作，当写操作导致数据变化时会自动将数据同步给从数据库。而从数据库一般是只读的，并接受主数据库同步过来的数据。一个主数据库可以拥有多个从数据库，而一个从数据库只能拥有一个主数据库。

全量复制：

（1）主节点通过bgsave命令fork子进程进行RDB持久化，该过程是非常消耗CPU、内存(页表复制)、硬盘IO的

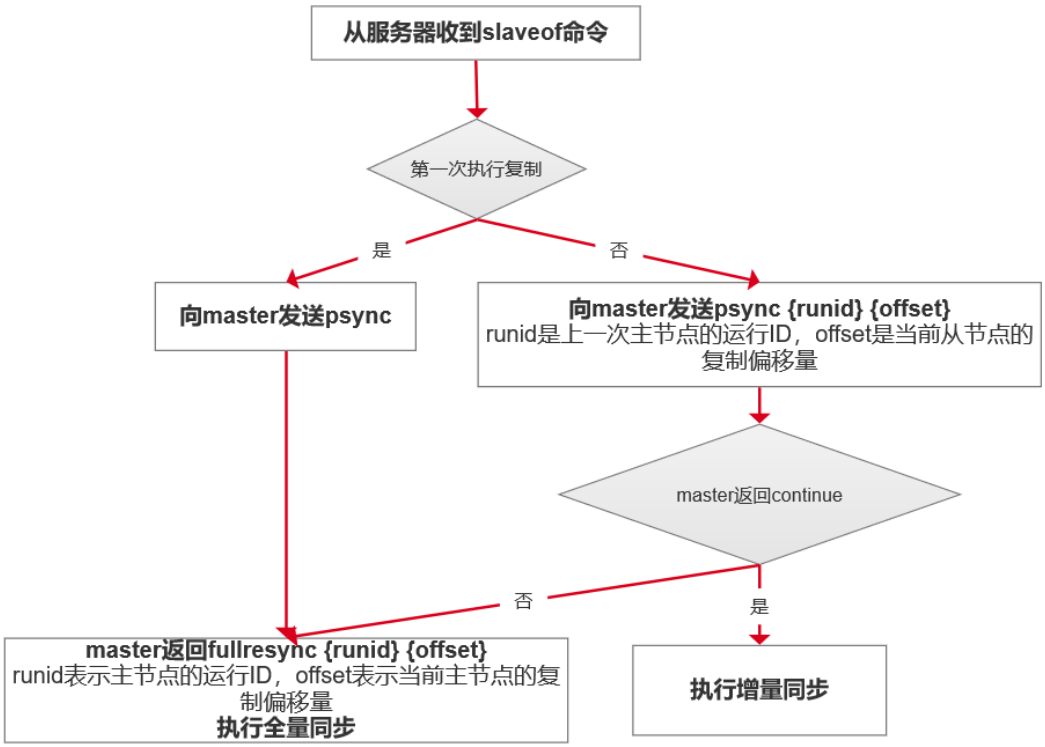
（2）主节点通过网络将RDB文件发送给从节点，对主从节点的带宽都会带来很大的消耗

（3）从节点清空老数据、载入新RDB文件的过程是阻塞的，无法响应客户端的命令；如果从节点执行bgrewriteaof，也会带来额外的消耗

部分复制：

1. 复制偏移量：执行复制的双方，主从节点，分别会维护一个复制偏移量offset
2. 复制积压缓冲区：主节点内部维护了一个固定长度的、先进先出(FIFO)队列 作为复制积压缓冲区，当主从节点offset的差距过大超过缓冲区长度时，将无法执行部分复制，只能执行全量复制。
3. 服务器运行ID(runid)：每个Redis节点，都有其运行ID，运行ID由节点在启动时自动生成，主节点会将自己的运行ID发送给从节点，从节点会将主节点的运行ID存起来。 从节点Redis断开重连的时候，就是根据运行ID来判断同步的进度：
    - 如果从节点保存的runid与主节点现在的runid相同，说明主从节点之前同步过，主节点会继续尝试使用部分复制(到底能不能部分复制还要看offset和复制积压缓冲区的情况)；
    - 如果从节点保存的runid与主节点现在的runid不同，说明从节点在断线前同步的Redis节点并不是当前的主节点，只能进行全量复制。

过程原理：



缓存雪崩、缓存穿透、缓存击穿

缓存雪崩是指缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决方案：

- 缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。
- 给每一个缓存数据增加相应的缓存标记，记录缓存是否失效，如果缓存标记失效，则更新数据缓存。
- 缓存预热
- 互斥锁

缓存穿透是指缓存和数据库中都没有的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决方案：

- 接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截；
- 从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击
- 采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力。和缓存雪崩不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

解决方案

- 设置热点数据永远不过期。
- 加互斥锁

分布式/微服务

CAP理论，BASE理论

Consistency (一致性)：

即更新操作成功并返回客户端后，所有节点在同一时间的数据完全一致。

对于客户端来说，一致性指的是并发访问时更新过的数据如何获取的问题。

从服务端来看，则是更新如何复制分布到整个系统，以保证数据最终一致。

Availability (可用性):

即服务一直可用，而且是正常响应时间。系统能够很好的为用户服务，不出现用户操作失败或者访问超时等用户体验不好的情况。

Partition Tolerance (分区容错性):

即分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务。分区容错性要求能够使应用虽然是一个分布式系统，而看上去却好像是在一个可以运转正常的整体。比如现在的分布式系统中有某一个或者几个机器宕掉了，其他剩下的机器还能够正常运转满足系统需求，对于用户而言并没有什么体验上的影响。

CP和AP：分区容错是必须保证的，当发生网络分区的时候，如果要继续服务，那么强一致性和可用性只能 2 选 1

BASE是Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）

BASE理论是对CAP中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结，是基于CAP定理逐步演化而来的。BASE理论的核心思想是：即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。

基本可用：

- 响应时间上的损失: 正常情况下，处理用户请求需要 0.5s 返回结果，但是由于系统出现故障，处理用户请求的时间变为 3 s。
- 系统功能上的损失：正常情况下，用户可以使用系统的全部功能，但是由于系统访问量突然剧增，系统的部分非核心功能无法使用。

软状态：数据同步允许一定的延迟

最终一致性：系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态，不要求实时

负载均衡算法、类型

1、轮询法

将请求按顺序轮流地分配到后端服务器上，它均衡地对待后端的每一台服务器，而不关心服务器实际的连接数和当前的系统负载。

2、随机法

通过系统的随机算法，根据后端服务器的列表大小值来随机选取其中的一台服务器进行访问。由概率统计理论可以得知，随着客户端调用服务端的次数增多，

其实际效果越来越接近于平均分配调用量到后端的每一台服务器，也就是轮询的结果。

3、源地址哈希法

源地址哈希的思想是根据获取客户端的IP地址，通过哈希函数计算得到的一个数值，用该数值对服务器列表的大小进行取模运算，得到的结果便是客服端要访问服务器的序号。采用源地址哈希法进行负载均衡，同一IP地址的客户端，当后端服务器列表不变时，它每次都会映射到同一台后端服务器进行访问。

4、加权轮询法

不同的后端服务器可能机器的配置和当前系统的负载并不相同，因此它们的抗压能力也不相同。给配置高、负载低的机器配置更高的权重，让其处理更多的请；而配置低、负载高的机器，给其分配较低的权重，降低其系统负载，加权轮询能很好地处理这一问题，并将请求顺序且按照权重分配到后端。

5、加权随机法

与加权轮询法一样，加权随机法也根据后端机器的配置，系统的负载分配不同的权重。不同的是，它是按照权重随机请求后端服务器，而非顺序。

6、最小连接数法

最小连接数算法比较灵活和智能，由于后端服务器的配置不尽相同，对于请求的处理有快有慢，它是根据后端服务器当前的连接情况，动态地选取其中当前

积压连接数最少的一台服务器来处理当前的请求，尽可能地提高后端服务的利用效率，将负责合理地分流到每一台服务器。

类型：

DNS 方式实现负载均衡

硬件负载均衡：F5 和 A10

软件负载均衡：

`Nginx`、`HAproxy`、`LVS`。其中的区别：

- `Nginx`：七层负载均衡，支持 HTTP、E-mail 协议，同时也支持 4 层负载均衡；
- `HAproxy`：支持七层规则的，性能也很不错。OpenStack 默认使用的负载均衡软件就是 HAproxy；
- `LVS`：运行在内核态，性能是软件负载均衡中最高的，严格来说工作在三层，所以更通用一些，适用各种应用服务。

分布式架构下，Session 共享有什么方案

1、采用无状态服务，抛弃session

2、存入cookie（有安全风险）

3、服务器之间进行 Session 同步，这样可以保证每个服务器上都有全部的 Session 信息，不过当服务器数量比较多的时候，同步是会有延迟甚至同步失败；

4、 IP 绑定策略

使用 Nginx （或其他复杂均衡软硬件）中的 IP 绑定策略，同一个 IP 只能在指定的同一个机器访问，但是这样做失去了负载均衡的意义，当挂掉一台服务器的时候，会影响一批用户的使用，风险很大；

5、使用 Redis 存储

把 Session 放到 Redis 中存储，虽然架构上变得复杂，并且需要多访问一次 Redis ，但是这种方案带来的好处也是很大的：

- 实现了 Session 共享；
- 可以水平扩展（增加 Redis 服务器）；
- 服务器重启 Session 不丢失（不过也要注意 Session 在 Redis 中的刷新/失效机制）；
- 不仅可以跨服务器 Session 共享，甚至可以跨平台（例如网页端和 APP 端）。

简述你对RPC、RMI的理解

RPC：在本地调用远程的函数，远程过程调用，可以跨语言实现　httpClient

RMI：远程方法调用，java中用于实现RPC的一种机制，RPC的java版本，是J2EE的网络调用机制，跨JVM调用对象的方法，面向对象的思维方式

直接或间接实现接口　java.rmi.Remote　成为存在于服务器端的远程对象，供客户端访问并提供一定的服务

远程对象必须实现java.rmi.server.UniCastRemoteObject类，这样才能保证客户端访问获得远程对象时，该远程对象将会把自身的一个拷贝以Socket的形式传输给客户端，此时客户端所获得的这个拷贝称为“存根”，而服务器端本身已存在的远程对象则称之为“骨架”。其实此时的存根是客户端的一个代理，用于与服务器端的通信，而骨架也可认为是服务器端的一个代理，用于接收客户端的请求之后调用远程方法来响应客户端的请求。

```
public interface IService extends Remote {
    String service(String content) throws RemoteException;
}

public class ServiceImpl extends UnicastRemoteObject implements IService {

    private String name;

    public ServiceImpl(String name) throws RemoteException {
        this.name = name;
    }
    @Override
    public String service(String content) {
        return "server >> " + content;
    }
```

```
}

public class Server {
    public static void main(String[] args) {
        try {
            IService service02 = new ServiceImpl("service02");
            Context namingContext = new InitialContext();
            namingContext.rebind("rmi://127.0.0.1/service02", service02);
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("000000! ");
    }
}

public class Client {
    public static void main(String[] args) {
        String url = "rmi://127.0.0.1/";
        try {
            Context namingContext = new InitialContext();
            IService service02 = (IService) namingContext.lookup(url +
"service02");
            Class stubClass = service02.getClass();
            System.out.println(service02 + " is " + stubClass.getName());
            //com.sun.proxy.$Proxy0

            Class[] interfaces = stubClass.getInterfaces();
            for (Class c : interfaces) {
                System.out.println("implement" + c.getName() + " interface");
            }
            System.out.println(service02.service("hello"));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

分布式id生成方案

- uuid

1，当前日期和时间　　时间戳
2，时钟序列。　　计数器
3，全局唯一的IEEE机器识别号，如果有网卡，从网卡MAC地址获得，没有网卡以其他方式获得。

优点：代码简单，性能好（本地生成，没有网络消耗），保证唯一（相对而言，重复概率极低可以忽略）
缺点：
　　每次生成的ID都是无序的，而且不是全数字，且无法保证趋势递增。
　　UUID生成的是字符串，字符串存储性能差，查询效率慢，写的时候由于不能产生顺序的append操作，需要进　　行insert操作，导致频繁的页分裂，这种操作在记录占用空间比较大的情况下，性能下降比较大，还会增加读　　取磁盘次数
　　UUID长度过长，不适用于存储，耗费数据库性能。
　　ID无一定业务含义，可读性差。
　　有信息安全问题，有可能泄露mac地址

- 数据库自增序列

单机模式：
优点：
　　实现简单，依靠数据库即可，成本小。
　　ID数字化，单调自增，满足数据库存储和查询性能。
　　具有一定的业务可读性。（结合业务code）
缺点：
　　强依赖DB，存在单点问题，如果数据库宕机，则业务不可用。
　　DB生成ID性能有限，单点数据库压力大，无法扛高并发场景。
　　信息安全问题，比如暴露订单量，url查询改一下id查到别人的订单

数据库高可用：多主模式做负载，基于序列的起始值和步长设置，不同的初始值，相同的步长，步长大于节点数
优点：
　　解决了ID生成的单点问题，同时平衡了负载。
缺点：
　　系统扩容困难：系统定义好步长之后，增加机器之后调整步长困难。
　　数据库压力大：每次获取一个ID都必须读写一次数据库。
　　主从同步的时候：电商下单->支付insert master db select数据 ，因为数据同步延迟导致查不到这个数　　据。加cache(不是最好的解决方式)数据要求比较严谨的话查master主库。

- Leaf-segment

采用每次获取一个ID区间段的方式来解决，区间段用完之后再去数据库获取新的号段，这样一来可以大大减轻数据库的压力
核心字段：biz_tag，max_id，step
biz_tag用来区分业务，max_id表示该biz_tag目前所被分配的ID号段的最大值，step表示每次分配的号段长度，原来每次获取ID都要访问数据库，现在只需要把Step设置的足够合理如1000，那么现在可以在1000个ID用完之后再去访问数据库
优点：
　　扩张灵活，性能强能够撑起大部分业务场景。
　　ID号码是趋势递增的，满足数据库存储和查询性能要求。
　　可用性高，即使ID生成服务器不可用，也能够使得业务在短时间内可用，为排查问题争取时间。
缺点：
　　可能存在多个节点同时请求ID区间的情况，依赖DB

双buffer：将获取一个号段的方式优化成获取两个号段，在一个号段用完之后不用立马去更新号段，还有一个缓存号段备用，这样能够有效解决这种冲突问题，而且采用双buffer的方式，在当前号段消耗了10%的时候就去检查下一个号段有没有准备好，如果没有准备好就去更新下一个号段，当当前号段用完了就切换到下一个已经缓存好的号段去使用，同时在下一个号段消耗到10%的时候，又去检测下一个号段有没有准备好，如此往复。
优点：
基于JVM存储双buffer的号段，减少了数据库查询，减少了网络依赖，效率更高。

缺点：
segment号段长度是固定的，业务量大时可能会频繁更新号段，因为原本分配的号段会一下用完
如果号段长度设置的过长，但凡缓存中有号段没有消耗完，其他节点重新获取的号段与之前相比可能跨度会很 大。动态调整Step

- 基于redis、mongodb、zk等中间件生成
- 雪花算法

生成一个64bit的整性数字
第一位符号位固定为0，41位时间戳，10位workId，12位序列号
位数可以有不同实现
优点：
每个毫秒值包含的ID值很多，不够可以变动位数来增加，性能佳（依赖workId的实现）。
时间戳值在高位，中间是固定的机器码，自增的序列在低位，整个ID是趋势递增的。
能够根据业务场景数据库节点布置灵活挑战bit位划分，灵活度高。
缺点：
强依赖于机器时钟，如果时钟回拨，会导致重复的ID生成，所以一般基于此的算法发现时钟回拨，都会抛异常处 理，阻止ID生成，这可能导致服务不可用。

分布式锁解决方案

需要这个锁独立于每一个服务之外，而不是在服务里面。

数据库：利用主键冲突控制一次只有一个线程能获取锁，非阻塞、不可重入、单点、失效时间

Zookeeper分布式锁：

zk通过临时节点，解决了死锁的问题，一旦客户端获取到锁之后突然挂掉（Session连接断开），那么这个临时节点就会自动删除掉，其他客户端自动获取锁。临时顺序节点解决惊群效应

Redis分布式锁：setNX，单线程处理网络请求，不需要考虑并发安全性

所有服务节点设置相同的key，返回为0、则锁获取失败

```
setnx
问题：
    1、早期版本没有超时参数，需要单独设置，存在死锁问题（中途宕机）
    2、后期版本提供加锁与设置时间原子操作，但是存在任务超时，锁自动释放，导致并发问题，加锁与释放锁不是同一线程问题
```

删除锁：判断线程唯一标志，再删除

可重入性及锁续期没有实现，通过redisson解决（类似AQS的实现，看门狗监听机制）

redlock：意思的机制都只操作单节点、即使Redis通过sentinel保证高可用，如果这个master节点由于某些原因发生了主从切换，那么就会出现锁丢失的情况（redis同步设置可能数据丢失）。redlock从多个节点申请锁，当一半以上节点获取成功、锁才算获取成功，redission有相应的实现

分布式事务解决方案

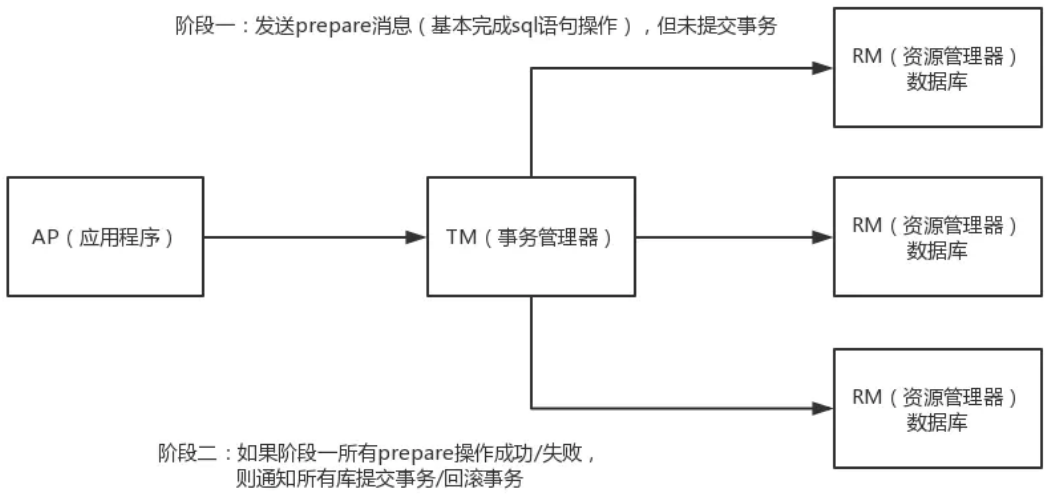
XA规范：分布式事务规范，定义了分布式事务模型

四个角色：事务管理器(协调者TM)、资源管理器(参与者RM)，应用程序AP，通信资源管理器CRM

全局事务：一个横跨多个数据库的事务，要么全部提交、要么全部回滚

JTA事务时java对XA规范的实现，对应JDBC的单库事务

两阶段协议：



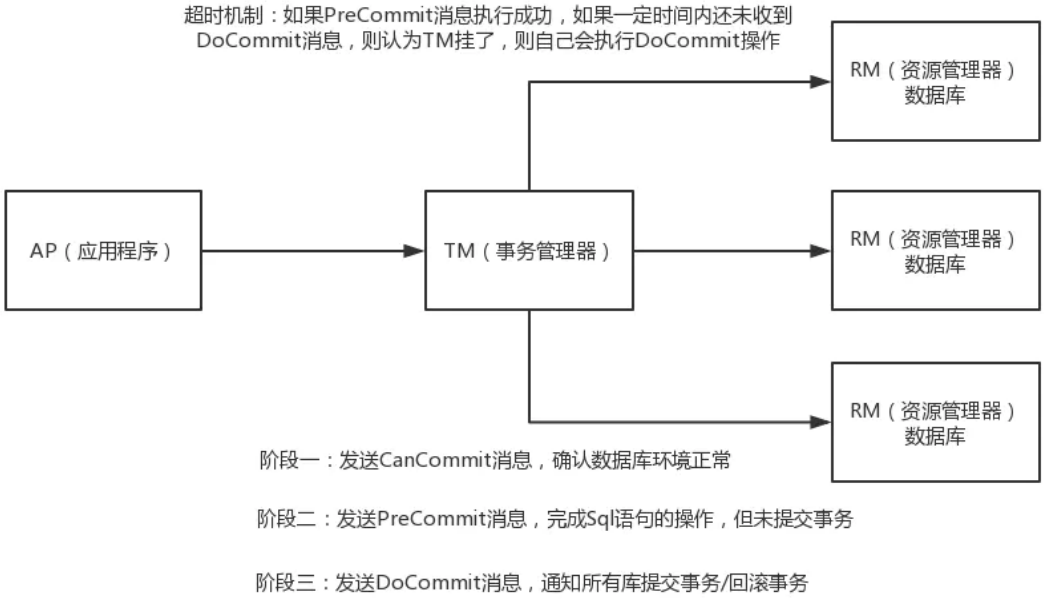
第一阶段（prepare）：每个参与者执行本地事务但不提交，进入 ready 状态，并通知协调者已经准备就绪。

第二阶段（commit） 当协调者确认每个参与者都 ready 后，通知参与者进行 commit 操作；如果有参与者 fail，则发送 rollback 命令，各参与者做回滚。

问题：

- 单点故障：一旦事务管理器出现故障，整个系统不可用（参与者都会阻塞住）
- 数据不一致：在阶段二，如果事务管理器只发送了部分 commit 消息，此时网络发生异常，那么只有部分参与者接收到 commit 消息，也就是说只有部分参与者提交了事务，使得系统数据不一致。
- 响应时间较长：参与者和协调者资源都被锁住，提交或者回滚之后才能释放
- 不确定性：当协事务管理器发送 commit 之后，并且此时只有一个参与者收到了 commit，那么当该参与者与事务管理器同时宕机之后，重新选举的事务管理器无法确定该条消息是否提交成功。

三阶段协议：主要是针对两阶段的优化，解决了2PC单点故障的问题，但是性能问题和不一致问题仍然没有根本解决



引入了超时机制解决参与者阻塞的问题，超时后本地提交，2pc只有协调者有超时机制

- 第一阶段：CanCommit阶段，协调者询问事务参与者，是否有能力完成此次事务。
  - 如果都返回yes，则进入第二阶段
  - 有一个返回no或等待响应超时，则中断事务，并向所有参与者发送abort请求
- 第二阶段：PreCommit阶段，此时协调者会向所有的参与者发送PreCommit请求，参与者收到后开始执行事务操作。参与者执行完事务操作后（此时属于未提交事务的状态），就会向协调者反馈“Ack”表示我已经准备好提交了，并等待协调者的下一步指令。
- 第三阶段：DoCommit阶段， 在阶段二中如果所有的参与者节点都返回了Ack，那么协调者就会从“预提交状态”转变为“提交状态”。然后向所有的参与者节点发送"doCommit"请求，参与者节点在收到提交请求后就会各自执行事务提交操作，并向协调者节点反馈“Ack”消息，协调者收到所有参与者的Ack消息后完成事务。 相反，如果有一个参与者节点未完成PreCommit的反馈或者反馈超时，那么协调者都会向所有的参与者节点发送abort请求，从而中断事务。

TCC（补偿事务）：Try、Confirm、Cancel

针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作

Try操作做业务检查及资源预留，Confirm做业务确认操作，Cancel实现一个与Try相反的操作既回滚操作。TM首先发起所有的分支事务的try操作，任何一个分支事务的try操作执行失败，TM将会发起所有分支事务的Cancel操作，若try操作全部成功，TM将会发起所有分支事务的Confirm操作，其中Confirm/Cancel操作若执行失败，TM会进行重试。

TCC模型对业务的侵入性较强，改造的难度较大，每个操作都需要有 `try`、`confirm`、`cancel`三个接口实现

`confirm` 和 `cancel` 接口还必须实现幂等性。

消息队列的事务消息：

- 发送prepare消息到消息中间件
- 发送成功后，执行本地事务
    - 如果事务执行成功，则commit，消息中间件将消息下发至消费端（commit前，消息不会被消费）
    - 如果事务执行失败，则回滚，消息中间件将这条prepare消息删除
- 消费端接收到消息进行消费，如果消费失败，则不断重试

如何实现接口的幂等性

- 唯一id。每次操作，都根据操作和内容生成唯一的id，在执行之前先判断id是否存在，如果不存在则执行后续操作，并且保存到数据库或者redis等。
- 服务端提供发送token的接口，业务调用接口前先获取token,然后调用业务接口请求时，把token携带过去,务器判断token是否存在redis中，存在表示第一次请求，可以继续执行业务，执行业务完成后，最后需要把redis中的token删除
- 建去重表。将业务中有唯一标识的字段保存到去重表，如果表中存在，则表示已经处理过了
- 版本控制。增加版本号，当版本号符合时，才能更新数据
- 状态控制。例如订单有状态已支付 未支付 支付中 支付失败，当处于未支付的时候才允许修改为支付中等

简述ZAB 协议

ZAB 协议是为分布式协调服务 Zookeeper 专门设计的一种支持崩溃恢复的原子广播协议，实现分布式数据一致性

所有客户端的请求都是写入到 Leader 进程中，然后，由 Leader 同步到其他节点，称为 Follower。在集群数据同步的过程中，如果出现 Follower 节点崩溃或者 Leader 进程崩溃时，都会通过 Zab 协议来保证数据一致性

ZAB 协议包括两种基本的模式：**崩溃恢复和消息广播。**

消息广播：

集群中所有的事务请求都由 Leader 节点来处理，其他服务器为 Follower，Leader 将客户端的事务请求转换为事务 Proposal，并且将 Proposal 分发给集群中其他所有的 Follower。

完成广播之后，Leader 等待 Follwer 反馈，当有过半数的 Follower 反馈信息后，Leader 将再次向集群内 Follower 广播 Commit 信息，Commit 信息就是确认将之前的 Proposal 提交。

Leader 节点的写入是一个两步操作，第一步是广播事务操作，第二步是广播提交操作，其中过半数指的是反馈的节点数 >=N/2+1，N 是全部的 Follower 节点数量。

崩溃恢复：

- 初始化集群，刚刚启动的时候
- Leader 崩溃，因为故障宕机
- Leader 失去了半数的机器支持，与集群中超过一半的节点断连

此时开启新一轮 Leader 选举，选举产生的 Leader 会与过半的 Follower 进行同步，使数据一致，当与过半的机器同步完成后，就退出恢复模式， 然后进入消息广播模式

整个 ZooKeeper 集群的一致性保证就是在上面两个状态之前切换，当 Leader 服务正常时，就是正常的消息广播模式；当 Leader 不可用时，则进入崩溃恢复模式，崩溃恢复阶段会进行数据同步，完成以后，重新进入消息广播阶段。

Zxid 是 Zab 协议的一个事务编号，Zxid 是一个 64 位的数字，其中低 32 位是一个简单的单调递增计数器，针对客户端每一个事务请求，计数器加 1；而高 32 位则代表 Leader 周期年代的编号。

Leader 周期（ epoch），可以理解为当前集群所处的年代或者周期，每当有一个新的 Leader 选举出现时，就会从这个 Leader 服务器上取出其本地日志中最大事务的 Zxid，并从中读取 epoch 值，然后加 1，以此作为新的周期 ID。高 32 位代表了每代 Leader 的唯一性，低 32 位则代表了每代 Leader 中事务的唯一性。

zab节点的三种状态：

following：服从leader的命令

leading：负责协调事务

election/looking：选举状态

zk的数据模型和节点类型

数据模型：树形结构

zk维护的数据主要有：客户端的会话（session）状态及数据节点（dataNode）信息。

zk在内存中构造了个DataTree的数据结构，维护着path到dataNode的映射以及dataNode间的树状层级关系。为了提高读取性能，集群中每个服务节点都是将数据全量存储在内存中。所以，zk最适于读多写少且轻量级数据的应用场景。

数据仅存储在内存是很不安全的，zk采用事务日志文件及快照文件的方案来落盘数据，保障数据在不丢失的情况下能快速恢复。

树中的每个节点被称为— Znode

Znode 兼具文件和目录两种特点。可以做路径标识，也可以存储数据，并可以具有子 Znode。具有增、删、改、查等操作。

Znode 具有原子性操作，读操作将获取与节点相关的所有数据，写操作也将 替换掉节点的所有数据。另外，每一个节点都拥有自己的 ACL(访问控制列 表)，这个列表规定了用户的权限，即限定了特定用户对目标节点可以执行的操作

Znode 存储数据大小有限制。每个 Znode 的数据大小至多 1M，常规使用中应该远小于此值。

Znode 通过路径引用，如同 Unix 中的文件路径。路径必须是绝对的，因此他们必须由斜杠字符来开头。除此以外，他们必须是唯一的，也就是说每一个路径只有一个表示，因此这些路径不能改变。在 ZooKeeper 中，路径由 Unicode 字符串组成，并且有一些限制。字符串"/zookeeper"用以保存管理信息，比如关键配额信息。

持久节点：一旦创建、该数据节点会一直存储在zk服务器上、即使创建该节点的客户端与服务端的会话关闭了、该节点也不会被删除

临时节点：当创建该节点的客户端会话因超时或发生异常而关闭时、该节点也相应的在zk上被删除 。

有序节点：不是一种单独种类的节点、而是在持久节点和临时节点的基础上、增加了一个节点有序的性质 。

简述zk的命名服务、配置管理、集群管理

命名服务：

通过指定的名字来获取资源或者服务地址。Zookeeper可以创建一个全局唯一的路径，这个路径就可以作为一个名字。被命名的实体可以是集群中的机器，服务的地址，或者是远程的对象等。一些分布式服务框架（RPC、RMI）中的服务地址列表，通过使用命名服务，客户端应用能够根据特定的名字来获取资源的实体、服务地址和提供者信息等

配置管理：

实际项目开发中，经常使用.properties或者xml需要配置很多信息，如数据库连接信息、fps地址端口等等。程序分布式部署时，如果把程序的这些配置信息保存在zk的znode节点下，当你要修改配置，即znode会发生变化时，可以通过改变zk中某个目录节点的内容，利用watcher通知给各个客户端，从而更改配置。

集群管理：

集群管理包括集群监控和集群控制，就是监控集群机器状态，剔除机器和加入机器。zookeeper可以方便集群机器的管理，它可以实时监控znode节点的变化，一旦发现有机器挂了，该机器就会与zk断开连接，对应的临时目录节点会被删除，其他所有机器都收到通知。新机器加入也是类似。

讲下Zookeeper watch机制

客户端，可以通过在znode上设置watch，实现实时监听znode的变化

Watch事件是一个一次性的触发器，当被设置了Watch的数据发生了改变的时候，则服务器将这个改变发送给设置了Watch的客户端

- 父节点的创建，修改，删除都会触发Watcher事件。
- 子节点的创建，删除会触发Watcher事件。

一次性：一旦被触发就会移除，再次使用需要重新注册，因为每次变动都需要通知所有客户端，一次性可以减轻压力，3.6.0默认持久递归，可以触发多次

轻量：只通知发生了事件，不会告知事件内容，减轻服务器和带宽压力

Watcher 机制包括三个角色：客户端线程、客户端的 WatchManager 以及 ZooKeeper 服务器

1. 客户端向 ZooKeeper 服务器注册一个 Watcher 监听，
2. 把这个监听信息存储到客户端的 WatchManager 中
3. 当 ZooKeeper 中的节点发生变化时，会通知客户端，客户端会调用相应 Watcher 对象中的回调方法。watch回调是串行同步的

zk和eureka的区别

zk：CP设计(强一致性)，目标是一个分布式的协调系统，用于进行资源的统一管理。

当节点crash后，需要进行leader的选举，在这个期间内，zk服务是不可用的。

eureka：AP设计（高可用），目标是一个服务注册发现系统，专门用于微服务的服务发现注册。

Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册时如果发现连接失败，会自动切换至其他节点，只要有一台Eureka还在，就能保证注册服务可用（保证可用性），只不过查到的信息可能不是最新的（不保证强一致性）

同时当eureka的服务端发现85%以上的服务都没有心跳的话，它就会认为自己的网络出了问题，就不会从服务列表中删除这些失去心跳的服务，同时eureka的客户端也会缓存服务信息。eureka对于服务注册发现来说是非常好的选择。

Spring Cloud和Dubbo的区别

底层协议：springcloud基于http协议，dubbo基于Tcp协议，决定了dubbo的性能相对会比较好

注册中心：Spring Cloud 使用的 eureka ，dubbo推荐使用zookeeper

模型定义：dubbo 将一个接口定义为一个服务，SpringCloud 则是将一个应用定义为一个服务

SpringCloud是一个生态，而Dubbo是SpringCloud生态中关于服务调用一种解决方案（服务治理）

什么是Hystrix？简述实现机制

分布式容错框架

- 阻止故障的连锁反应，实现熔断
- 快速失败，实现优雅降级
- 提供实时的监控和告警

资源隔离：线程隔离，信号量隔离

- 线程隔离：Hystrix会给每一个Command分配一个单独的线程池，这样在进行单个服务调用的时候，就可以在独立的线程池里面进行，而不会对其他线程池造成影响

- 信号量隔离：客户端需向依赖服务发起请求时，首先要获取一个信号量才能真正发起调用，由于信号量的数量有限，当并发请求量超过信号量个数时，后续的请求都会直接拒绝，进入fallback流程。信号量隔离主要是通过控制并发请求量，防止请求线程大面积阻塞，从而达到限流和防止雪崩的目的。

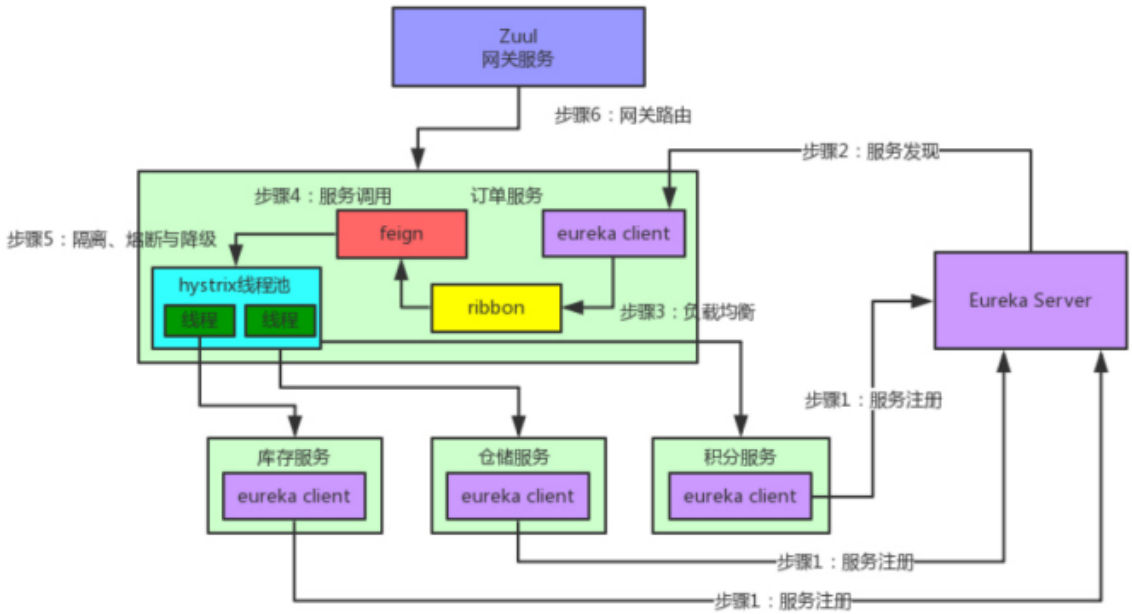
熔断和降级：调用服务失败后快速失败

熔断是为了防止异常不扩散，保证系统的稳定性

降级：编写好调用失败的补救逻辑，然后对服务直接停止运行，这样这些接口就无法正常调用，但又不至于直接报错，只是服务水平下降

- 通过HystrixCommand 或者HystrixObservableCommand 将所有的外部系统（或者称为依赖）包装起来，整个包装对象是单独运行在一个线程之中（这是典型的命令模式）。
- 超时请求应该超过你定义的阈值
- 为每个依赖关系维护一个小的线程池（或信号量）；如果它变满了，那么依赖关系的请求将立即被拒绝，而不是排队等待。
- 统计成功，失败（由客户端抛出的异常），超时和线程拒绝。
- 打开断路器可以在一段时间内停止对特定服务的所有请求，如果服务的错误百分比通过阈值，手动或自动的关闭断路器。
- 当请求被拒绝、连接超时或者断路器打开，直接执行fallback逻辑。
- 近乎实时监控指标和配置变化。

springcloud核心组件及其作用



Eureka：服务注册与发现

注册：每个服务都向Eureka登记自己提供服务的元数据，包括服务的ip地址、端口号、版本号、通信协议等。eureka将各个服务维护在了一个服务清单中（双层Map，第一层key是服务名，第二层key是实例名，value是服务地址加端口）。同时对服务维持心跳，剔除不可用的服务，eureka集群各节点相互注册每个实例中都有一样的服务清单。

发现：eureka注册的服务之间调用不需要指定服务地址，而是通过服务名向注册中心咨询，并获取所有服务实例清单(缓存到本地)，然后实现服务的请求访问。

Ribbon：服务间发起请求的时候，基于Ribbon做负载均衡，从一个服务的多台机器中选择一台 （被调用方的服务地址有多个），Ribbon也是通过发起http请求，来进行的调用，只不过是通过调用服务名的地址来实现的。虽然说Ribbon不用去具体请求服务实例的ip地址或域名了，但是每调用一个接口都还要手动去发起Http请求

```
@RestController
public class ConsumerController {
    @Autowired
    RestTemplate restTemplate;
    @GetMapping("/ribbon-consumer")
    public String helloConsumer(){
        return
restTemplate.getForEntity("http://exampleservice/index",String.class).getBody();
    }
}
```

Feign：基于Feign的动态代理机制，根据注解和选择的机器，拼接请求URL地址，发起请求，简化服务间的调用，在Ribbon的基础上进行了进一步的封装。单独抽出了一个组件，就是Spring Cloud Feign。在引入Spring Cloud Feign后，我们只需要创建一个接口并用注解的方式来配置它，即可完成对服务提供方的接口绑定。

调用远程就像调用本地服务一样

```
@RestController
public class UserController {
    @GetMapping("/getUser")
    public String getUser(){
        List<String> list = new ArrayList<>();
        list.add("张三");
        String json = JSON.toJSONString(list);
        return json;
    }
}



@FeignClient(name = "user")
public interface UserClient {
    @GetMapping("/getUser")
    String getUser();
}
```

```
@RestController
public class TestController {
    @Resource
    UserClient userClient;

    @RequestMapping("/test")
    public String test(){
        String user = userClient.getUser();
        return user;
    }
}
```

Hystrix：发起请求是通过Hystrix的线程池来走的，不同的服务走不同的线程池，实现了不同服务调用的隔离，通过统计接口超时次数返回默认值，实现服务熔断和降级

Zuul：如果前端、移动端要调用后端系统，统一从Zuul网关进入，由Zuul网关转发请求给对应的服务，通过与Eureka进行整合，将自身注册为Eureka下的应用，从Eureka下获取所有服务的实例，来进行服务的路由。Zuul还提供了一套过滤器机制，开发者可以自己指定哪些规则的请求需要执行校验逻辑，只有通过校验逻辑的请求才会被路由到具体服务实例上，否则返回错误提示。

Dubbo 的整体架构设计及分层

五个角色：

注册中心registry：服务注册与发现
服务提供者provider：暴露服务
服务消费者consumer：调用远程服务
监控中心monitor：统计服务的调用次数和调用时间
容器container：服务允许容器

调用流程：

1：container容器负责启动、加载、运行provider
2：provider在启动时，向regisitry中心注册自己提供的服务
3：consumer在启动时，向regisitry中心订阅自己所需的服务
4：regisitry返回服务提供者列表给consumer，如果有变更，registry将基于长连接推送变更数据给consumer
5：consumer调用provider服务，基于负载均衡算法进行调用
6：consumer调用provider的统计，基于短链接定时每分钟一次统计到monitor

分层：

接口服务层（Service）：面向开发者，业务代码、接口、实现等

配置层（Config）：对外配置接口，以 ServiceConfig 和 ReferenceConfig 为中心

服务代理层（Proxy）：对生产者和消费者、dubbo都会产生一个代理类封装调用细节，业务层对远程调用无感

服务注册层（Registry）： 封装服务地址的注册和发现， 以服务 URL 为中心

路由层（Cluster）： 封装多个提供者的路由和负载均衡， 并桥接注册中心

监控层（Monitor）： RPC 调用次数和调用时间监控

远程调用层（Protocal）：封装 RPC 调用

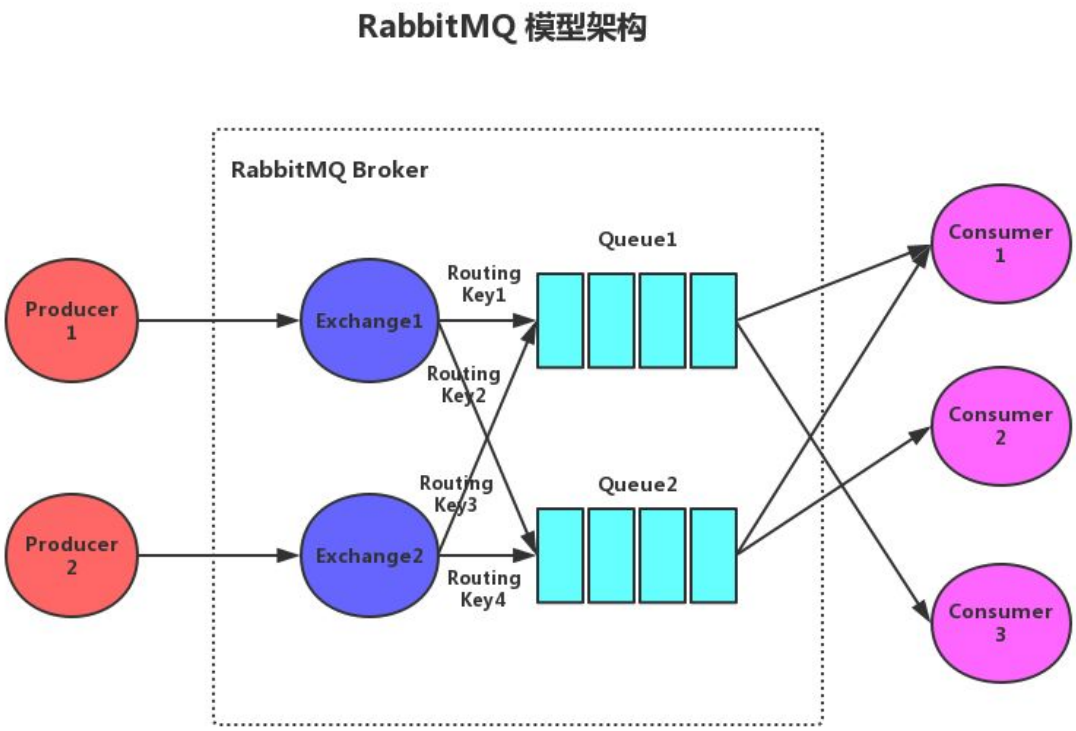
信息交换层（Exchange）： 封装请求响应模式， 同步转异步

网络传输层（Transport）：抽象 mina 和 netty 为统一接口，统一网络传输接口

数据序列化层（Serialize）： 数据传输的序列化和反序列化

MQ

简述RabbitMQ的架构设计



Broker：rabbitmq的服务节点

Queue：队列，是RabbitMQ的内部对象，用于存储消息。RabbitMQ中消息只能存储在队列中。生产者投递消息到队列，消费者从队列中获取消息并消费。多个消费者可以订阅同一个队列，这时队列中的消息会被平均分摊(轮询)给多个消费者进行消费，而不是每个消费者都收到所有的消息进行消费。(注意：RabbitMQ不支持队列层面的广播消费，如果需要广播消费，可以采用一个交换器通过路由Key绑定多个队列，由多个消费者来订阅这些队列的方式。

Exchange：交换器。生产者将消息发送到Exchange，由交换器将消息路由到一个或多个队列中。如果路由不到，或返回给生产者，或直接丢弃，或做其它处理。

RoutingKey：路由Key。生产者将消息发送给交换器的时候，一般会指定一个RoutingKey，用来指定这个消息的路由规则。这个路由Key需要与交换器类型和绑定键(BindingKey)联合使用才能最终生效。在交换器类型和绑定键固定的情况下，生产者可以在发送消息给交换器时通过指定RoutingKey来决定消息流向哪里。

Binding：通过绑定将交换器和队列关联起来，在绑定的时候一般会指定一个绑定键，这样RabbitMQ就可以指定如何正确的路由到队列了。

交换器和队列实际上是多对多关系。就像关系数据库中的两张表。他们通过BindingKey做关联(多对多关系表)。在投递消息时，可以通过Exchange和RoutingKey(对应BindingKey)就可以找到相对应的队列。

信道：信道是建立在Connection 之上的虚拟连接。当应用程序与Rabbit Broker建立TCP连接的时候，客户端紧接着可以创建一个AMQP 信道(Channel)，每个信道都会被指派一个唯一的D。RabbitMQ 处理的每条AMQP 指令都是通过信道完成的。信道就像电缆里的光纤束。一条电缆内含有许多光纤束，允许所有的连接通过多条光线束进行传输和接收。

RabbitMQ如何确保消息发送？ 消息接收?

发送方确认机制：

> 信道需要设置为 `confirm` 模式，则所有在信道上发布的消息都会分配一个唯一 `ID`。
>
> 一旦消息被投递到`queue`（可持久化的消息需要写入磁盘），信道会发送一个确认给生产者（包含消息唯一`ID`）。
>
> 如果 `RabbitMQ` 发生内部错误从而导致消息丢失，会发送一条 `nack`（未确认）消息给生产者。
>
> 所有被发送的消息都将被 `confirm`（即 `ack`） 或者被`nack`一次。但是没有对消息被 `confirm` 的快慢做任何保证，并且同一条消息不会既被 `confirm`又被`nack`
>
> 发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者，生产者的回调方法会被触发。
>
> `ConfirmCallback`接口：只确认是否正确到达 `Exchange` 中，成功到达则回调
>
> `ReturnCallback`接口：消息失败返回时回调

接收方确认机制：

> 消费者在声明队列时，可以指定noAck参数，当noAck=false时，RabbitMQ会等待消费者显式发回ack信号后才从内存(或者磁盘，持久化消息)中移去消息。否则，消息被消费后会被立即删除。
>
> 消费者接收每一条消息后都必须进行确认（消息接收和消息确认是两个不同操作）。只有消费者确认了消息，RabbitMQ 才能安全地把消息从队列中删除。
>
> RabbitMQ不会为未ack的消息设置超时时间，它判断此消息是否需要重新投递给消费者的唯一依据是消费该消息的消费者连接是否已经断开。这么设计的原因是RabbitMQ允许消费者消费一条消息的时间可以很长。保证数据的最终一致性；
>
> 如果消费者返回ack之前断开了链接，RabbitMQ 会重新分发给下一个订阅的消费者。（可能存在消息重复消费的隐患，需要去重）

RabbitMQ事务消息

通过对信道的设置实现

1. channel.txSelect()；通知服务器开启事务模式；服务端会返回Tx.Select-Ok
2. channel.basicPublish；发送消息，可以是多条，可以是消费消息提交ack
3. channel.txCommit()提交事务；
4. channel.txRollback()回滚事务；

消费者使用事务：

1. autoAck=false，手动提交ack，以事务提交或回滚为准；
2. autoAck=true，不支持事务的，也就是说你即使在收到消息之后在回滚事务也是于事无补的，队列已经把消息移除了

如果其中任意一个环节出现问题，就会抛出IoException异常，用户可以拦截异常进行事务回滚，或决定要不要重复消息。

事务消息会降低rabbitmq的性能

RabbitMQ死信队列、延时队列

1. 消息被消费方否定确认，使用 `channel.basicNack` 或 `channel.basicReject` ，并且此时 `requeue` 属性被设置为 `false`。
2. 消息在队列的存活时间超过设置的TTL时间。
3. 消息队列的消息数量已经超过最大队列长度。

那么该消息将成为“死信”。“死信”消息会被RabbitMQ进行特殊处理，如果配置了死信队列信息，那么该消息将会被丢进死信队列中，如果没有配置，则该消息将会被丢弃

为每个需要使用死信的业务队列配置一个死信交换机，这里同一个项目的死信交换机可以共用一个，然后为每个业务队列分配一个单独的路由key，死信队列只不过是绑定在死信交换机上的队列，死信交换机也不是什么特殊的交换机，只不过是用来接受死信的交换机，所以可以为任何类型【Direct、Fanout、Topic】

TTL：一条消息或者该队列中的所有消息的最大存活时间

如果一条消息设置了TTL属性或者进入了设置TTL属性的队列，那么这条消息如果在TTL设置的时间内没有被消费，则会成为“死信”。如果同时配置了队列的TTL和消息的TTL，那么较小的那个值将会被使用。

只需要消费者一直消费死信队列里的消息

RabbitMQ镜像队列机制

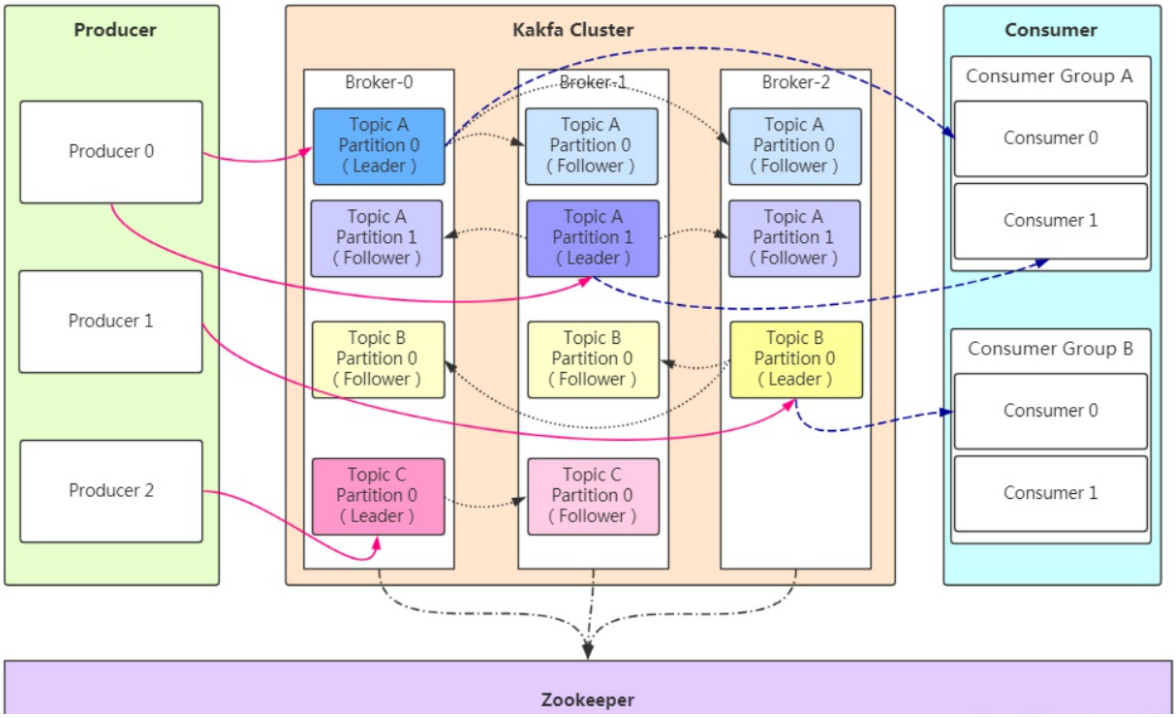
镜像queue有master节点和slave节点。master和slave是针对一个queue而言的，而不是一个node作为所有queue的master，其它node作为slave。一个queue第一次创建的node为它的master节点，其它node为slave节点。

无论客户端的请求打到master还是slave最终数据都是从master节点获取。当请求打到master节点时，master节点直接将消息返回给client，同时master节点会通过GM（Guaranteed Multicast）协议将queue的最新状态广播到slave节点。GM保证了广播消息的原子性，即要么都更新要么都不更新。

当请求打到slave节点时，slave节点需要将请求先重定向到master节点，master节点将将消息返回给client，同时master节点会通过GM协议将queue的最新状态广播到slave节点。

如果有新节点加入，RabbitMQ不会同步之前的历史数据，新节点只会复制该节点加入到集群之后新增的消息。

简述kafka架构设计



Consumer Group：消费者组，消费者组内每个消费者负责消费不同分区的数据，提高消费能力。逻辑上的一个订阅者。

Topic：可以理解为一个队列，Topic 将消息分类，生产者和消费者面向的是同一个 Topic。

Partition：为了实现扩展性，提高并发能力，一个Topic 以多个Partition的方式分布到多个 Broker 上，每个 Partition 是一个 有序的队列。一个 Topic 的每个Partition都有若干个副本（Replica），一个 Leader 和若干个 Follower。生产者发送数据的对象，以及消费者消费数据的对象，都是 Leader。Follower负责实时从 Leader 中同步数据，保持和 Leader 数据的同步。Leader 发生故障时，某个 Follower 还会成为新的 Leader。

Offset：消费者消费的位置信息，监控数据消费到什么位置，当消费者挂掉再重新恢复的时候，可以从消费位置继续消费。

Zookeeper：Kafka 集群能够正常工作，需要依赖于 Zookeeper，Zookeeper 帮助 Kafka 存储和管理集群信息。

kafka怎么处理消息顺序、重复发送、重复消费、消息丢失

Kafka在什么情况下会出现消息丢失及解决方案?

1）消息发送

1、ack=0，不重试
producer发送消息完，不管结果了，如果发送失败也就丢失了。

2、ack=1，leader crash
producer发送消息完，只等待lead写入成功就返回了，leader crash了，这时follower没来及同步，消息丢失。

3、unclean.leader.election.enable 配置true
允许选举ISR以外的副本作为leader,会导致数据丢失，默认为false。producer发送异步消息完，只等待lead写入成功就返回了，leader crash了，这时ISR中没有follower，leader从OSR中选举，因为OSR中本来落后于Leader造成消息丢失。

解决方案：
1、配置：ack=all / -1,tries > 1,unclean.leader.election.enable : false
producer发送消息完，等待follower同步完再返回，如果异常则重试。副本的数量可能影响吞吐量。

不允许选举ISR以外的副本作为leader。

2、配置：min.insync.replicas > 1
副本指定必须确认写操作成功的最小副本数量。如果不能满足这个最小值，则生产者将引发一个异常(要么是NotEnoughReplicas，要么是NotEnoughReplicasAfterAppend)。

min.insync.replicas和ack更大的持久性保证。确保如果大多数副本没有收到写操作，则生产者将引发异常。

3、失败的offset单独记录
producer发送消息，会自动重试，遇到不可恢复异常会抛出，这时可以捕获异常记录到数据库或缓存，进行单独处理。

2）消费

先commit再处理消息。如果在处理消息的时候异常了，但是offset 已经提交了，这条消息对于该消费者来说就是丢失了，再也不会消费到了。

3）broker的刷盘

减小刷盘间隔

Kafka是pull？push？优劣势分析

pull模式：

- 根据consumer的消费能力进行数据拉取，可以控制速率

- 可以批量拉取、也可以单条拉取
- 可以设置不同的提交方式，实现不同的传输语义

缺点：如果kafka没有数据，会导致consumer空循环，消耗资源

解决：通过参数设置，consumer拉取数据为空或者没有达到一定数量时进行阻塞

push模式：不会导致consumer循环等待

缺点：速率固定、忽略了consumer的消费能力，可能导致拒绝服务或者网络拥塞等情况

Kafka中zk的作用

/brokers/ids：临时节点，保存所有broker节点信息，存储broker的物理地址、版本信息、启动时间等，节点名称为brokerID，broker定时发送心跳到zk，如果断开则该brokerID会被删除

/brokers/topics：临时节点，节点保存broker节点下所有的topic信息，每一个topic节点下包含一个固定的partitions节点，partitions的子节点就是topic的分区，每个分区下保存一个state节点、保存着当前leader分区和ISR的brokerID，state节点由leader创建，若leader宕机该节点会被删除，直到有新的leader选举产生、重新生成state节点

/consumers/[group_id]/owners/[topic]/[broker_id-partition_id]：维护消费者和分区的注册关系

/consumers/[group_id]/offsets/[topic]/[broker_id-partition_id]：分区消息的消费进度Offset

client通过topic找到topic树下的state节点、获取leader的brokerID，到broker树中找到broker的物理地址，但是client不会直连zk，而是通过配置的broker获取到zk中的信息

简述kafka的rebalance机制

consumer group中的消费者与topic下的partion重新匹配的过程

何时会产生rebalance：

- consumer group中的成员个数发生变化
- consumer消费超时
- group订阅的topic个数发生变化
- group订阅的topic的分区数发生变化

coordinator：通常是partition的leader节点所在的broker，负责监控group中consumer的存活，consumer维持到coordinator的心跳，判断consumer的消费超时

- coordinator通过心跳返回通知consumer进行rebalance
- consumer请求coordinator加入组，coordinator选举产生leader consumer
- leader consumer从coordinator获取所有的consumer，发送syncGroup(分配信息)给到coordinator
- coordinator通过心跳机制将syncGroup下发给consumer
- 完成rebalance

leader consumer监控topic的变化，通知coordinator触发rebalance

如果C1消费消息超时，触发rebalance，重新分配后、该消息会被其他消费者消费，此时C1消费完成提交offset、导致错误

解决：coordinator每次rebalance，会标记一个Generation给到consumer，每次rebalance该Generation会+1，consumer提交offset时，coordinator会比对Generation，不一致则拒绝提交

Kafka的性能好在什么地方

kafka不基于内存，而是硬盘存储，因此消息堆积能力更强

顺序写：利用磁盘的顺序访问速度可以接近内存，kafka的消息都是append操作，partition是有序的，节省了磁盘的寻道时间，同时通过批量操作、节省写入次数，partition物理上分为多个segment存储，方便删除

传统：

- 读取磁盘文件数据到内核缓冲区
- 将内核缓冲区的数据copy到用户缓冲区
- 将用户缓冲区的数据copy到socket的发送缓冲区
- 将socket发送缓冲区中的数据发送到网卡、进行传输

零拷贝：

- 直接将内核缓冲区的数据发送到网卡传输
- 使用的是操作系统的指令支持

kafka不太依赖jvm，主要理由操作系统的pageCache，如果生产消费速率相当，则直接用pageCache交换数据，不需要经过磁盘IO