# 面试题冲冲冲

## Java常见基础面试题

### 多态

面向对象有4个基本特性：抽象、封装、继承、多态。总结起来就是抽象、封装、继承是多态的基础，多态是抽象、继承、封装的实现。

#### 什么是多态？

不同类的对象对同一消息作出不同的响应叫做多态。在Java中表现为不同子类对同一父类接口方法有不同的实现。

#### 作用

多态的最大作用就是解耦。

#### 存在条件

（1）有继承关系

（2）子类重写父类方法

（3）父类引用指向子类对象。

因此有以下三种类型没法表现出多态：

（1）static方法，属于类而不属于实例，故无法表现。

（2）final方法，final修饰无法被子类重写。

（3）private和protected方法。前者子类不可见，后者子类可见但不能被外部调用。

#### 分类

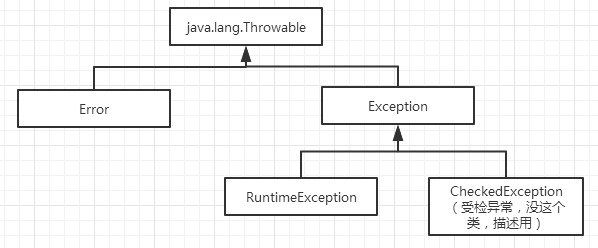
（1）编译时多态，即方法重载，从JVM角度讲，这是一种静态分派（static dispatch）。

（2）运行时多态，即方法重写，从JVM角度讲，这是一种动态分派（dynamic dispatch）。

#### 向上转型与向下转型

向上转型是自动的，向下转型需强转。

### 受检异常与非受检异常与Error



Error：表示底层（硬件、JVM等）错误，与程序本身无关，通常的做法是通知用户并中止程序的执行。

RuntimeException（非受检异常）：运行期时发生的异常，可以选择是否捕获。

受检异常：除RuntimeException（以及其子类）以外的异常都是受检异常，Java编译器要求程序必须捕获或声明抛出异常以保证程序健壮性。

什么时候用受检异常？什么时候用非受检异常？

受检异常大肆捕获会降低代码可读性，影响代码规范，故平时非底层基础异常（比如SQL、IO）时，能使用受检异常就用受检异常，少用非受检异常。

### Java创建对象有几种方式？

（1）new关键字创建对象

（2）反射，调用Class.newInstance方法。

（3）反序列化创建出一个对象

（4）clone方法。

### 自动拆装箱

#### 例子

Integer i1 = 2;  
Integer i2 = Integer.*valueOf*(2);   
System.***out***.println(i1 == i2); *//true*Integer i3 = **new** Integer(2);  
System.***out***.println(i1 == i3); *//false i3在堆重新分配，i2在缓存*Integer i4 = 128;  
System.***out***.println(i4 == Integer.*valueOf*(128)); *//false IntegerCache:-128-127*

#### 例题：如何实现一个方法交换两个Integer的值

Java只有值传递，只能用反射解决。

*// 获取实际存储值得字段*Field field = Integer.**class**.getDeclaredField(**"value"**);  
*// 设置访问权限*field.setAccessible(**true**);  
*// 中间缓存，此处一定不要使用int，使用int类型在自动装箱时，数字若在-128-127之间，会直接取  
// IntegerCache数组中的值（通过计算数组下标索引获得），会使结果不正确。*Integer temp = **new** Integer(i1.intValue());  
field.set(i1, i2.intValue());  
field.set(i2, temp);

这种解法会毁掉底层IntegerCache中原来的顺序，因为调用反射终究会改掉IntegerCache底层数组的值。

### Xml读取

很久没有做xml读取的代码了，还是记录一下：

#### DOM解析

Document Object Model，文档对象模型，基于DOM的xml分析器将xml文档转换成一股对象模型的集合。应用程序正是通过对这个对象模型的操作，来实现对XML文档数据的操作。通过DOM接口，应用程序可以在任何时候访问XML文档中的任何一部分数据，因此，这种利用DOM接口的机制也被称作随机访问机制。

优点：

1、形成了树结构，有助于更好的理解、掌握，且代码容易编写。

2、解析过程中，树结构保存在内存中，方便修改。

缺点：

1、由于文件是一次性读取，所以对内存的耗费比较大。

2、如果XML文件比较大，容易影响解析性能且可能会造成内存溢出。

#### SAX解析

SAX的全称是Simple APIs for XML，也即XML简单应用程序接口。与DOM不同，SAX提供的访问模式是一种顺序模式，这是一种快速读写XML数据的方式。当使用SAX分析器对XML文档进行分析时，会触发一系列事件，并激活相应的事件处理函数，应用程序通过这些事件处理函数实现对XML文档的访问，因而SAX接口也被称作事件驱动接口。

优点：

1、采用事件驱动模式，对内存耗费比较小。

2、适用于只处理XML文件中的数据时。

缺点：

1、编码比较麻烦。

2、很难同时访问XML文件中的多处不同数据。

#### JDOM解析

**特征：**

**1、仅使用具体类，而不使用接口。**

**2、API大量使用了Collections类。**

#### DOM4J解析

**特征：**

**1、JDOM的一种智能分支，它合并了许多超出基本XML文档表示的功能。**

**2、它使用接口和抽象基本类方法。**

**3、具有性能优异、灵活性好、功能强大和极端易用的特点。**

**4、是一个开放源码的文件**

#### 总结

DOM4J性能最好，JDOM和DOM在性能测试时表现不佳，小文档时可以使用，SAX表现较好。

### String、StringBuilder与StringBuffer

String str1 = “abc”; String str2 = “abc”；

问str1 == str2？

结果是true。

#### String的比较

JVM中有一块区域叫常量池，常量池中的数据是那些在编译期间被确定，并被保存在已编译的.class文件中的一些数据。除了8中基本数据类型外，还有String及其数组的常量值，另外还有一些以文本形式出现的符号引用。

上面的问题之所以为true，是因为==比较的是引用的地址，编译时常量池创建了常量”abc”，str1与str2均指向常量池中同一个”abc”常量。但如果此时来一个str3 = new String(“abc”)，则str1==str3为false，因为new关键字会在堆中产生一个”abc”常量。

如果此时执行String str4 = str3.intern()，那么该字符串将会被写入到常量池中（原堆内的字符串还在），此时str1==str4又为true。

#### String能否被继承？

不能，因为final关键字。

#### String是否线程安全？

安全，因为final类，一旦创建则不可改变。

#### String底层存储？

字节数组。

#### 为什么使用StringBuilder和StringBuffer拼接字符串？

如果使用”+”，编译器每次碰到该符号，便会new一个StringBuilder出来，接着调用append方法，再调用toString方法，生成新字符串。可见多个”+”的情况下，性能和效率会爆炸。

StringBuilder和StringBuffer原理一样，底层维护一个char数组，每次append就往里面添加字符，最终执行toString()生成String对象。唯一性能损耗点在于char数组不够时需要进行扩容，扩容需要进行数组拷贝。故如果能估计拼接字符串的长度，尽量通过构造函数指定长度。

同时StringBuffer的所有方法都做了同步，线程安全，但性能低于StringBuilder，故方法内部尽可能使用StringBuilder，避免同步带来的消耗。

当然，对于String str = “1” + “2” + “3” + “4”这种代码，编译器会直接生成”1234”对象，而不是生成多个StringBuilder拼接，这是编译器的内部优化。但最好还是别写成这样，可读性低。

#### hashCode()

1 public int hashCode() {

2 int h = hash;

3 if (h == 0 && value.length > 0) {

4 char val[] = value;

5

6 for (int i = 0; i < value.length; i++) {

7 h = 31 \* h + val[i];

8 }

9 hash = h;

10 }

11 return h;

12 }

以31为权，每一位为字符的ASCII值进行运算，用自然溢出来等效取模。

为什么取31，是因为31是一个奇质数，便于运算。

### hashCode

#### 什么是Hash

Hash是散列的意思，就是把任意长度的输入，通过散列算法变换成固定长度的输出，该输出就是散列值。

关于散列值，有以下几个关键结论：

1、对相同数据运算，输出结果必定相同

2、对不同数据运算，输出长度相同，内容可能相同（碰撞或者hash冲突）

3、同一hash算法的hash运算结果不同，则原始输入必定不同

4、算法不可逆

#### HashCode总结

（1）HashCode的存在是为了查找的快捷性，用来确定散列存储结构中对象的存储地址。

（2）如果两个对象的equals相等，那么两个对象的hashCode一定相等。

（3）如果两个对象的hashCode相等，那么两个对象的equals不一定相等，只能说明这两个对象在散列存储结构中，处在同一个位置。

（4）如果对象的equals方法被重写，那么对象的HashCode也尽量重写。

#### HashCode作用

比如n个元素Set集合，使用equals来比较对象，如果每次add一个对象都只用equals，那么可能会造成n次equals比较，但有了hashCode，就可以先比较HashCode，如果hashcode不同，那么equals肯定也不会相同，只有hashcode相同的情况下，再去比较equals，没有相同的元素就存储，有就不存，这样调用equals的次数大大降低，提高效率。

#### equals与hashCode与==区别

equals：反映的是对象或变量具体的值，即两个对象里面包含的值--可能是对象的引用，也可能是值类型的值，遵守对称性，反射性，类推性，一致性四个原则。一般应用于集合元素的比较，避免重复插入。

hashCode：根据对象的值计算其物理存储位置。

前者在Object中比较对象的地址值，后者为本地方法实现与机器相关。

所以：

equals相等，hashCode一定相等。

equals不等，hashCode不一定不等。

hashCode相等，equals不一定相等。

hashCode不等，equals一定不等。

而==就比较简单了，比较对象的引用是否相等，基本数据类型和包装类比较会自动拆装箱，比具体值。String常量池值唯一，比较也会相等。

### Comparable与Comparator

#### Comparable

内比较器，实现了该接口的类可以与自己比较，主要根据实现的compareTo(T o)方法进行比较。

1、比较者大于被比较者（也就是compareTo方法里面的对象），那么返回正整数

2、比较者等于被比较者，那么返回0

3、比较者小于被比较者，那么返回负整数

#### Comparator

外比较器

1、一个对象不支持自己和自己比较（没有实现Comparable接口），但是又想对两个对象进行比较

2、一个对象实现了Comparable接口，但是开发者认为compareTo方法中的比较方式并不是自己想要的那种比较方式

主要根据compare（T o1, To2）进行比较。

1、o1大于o2，返回正整数

2、o1等于o2，返回0

3、o1小于o3，返回负整数

1、如果实现类没有实现Comparable接口，又想对两个类进行比较（或者实现类实现了Comparable接口，但是对compareTo方法内的比较算法不满意），那么可以实现Comparator接口，自定义一个比较器，写比较算法

2、实现Comparable接口的方式比实现Comparator接口的耦合性要强一些，如果要修改比较算法，要修改Comparable接口的实现类，而实现Comparator的类是在外部进行比较的，不需要对实现类有任何修改。从这个角度说，其实有些不太好，尤其在我们将实现类的.class文件打成一个.jar文件提供给开发者使用的时候。实际上实现Comparator接口的方式后面会写到就是一种典型的**策略模式**。

当然，这不是鼓励用Comparator，意思是开发者还是要在具体场景下选择最合适的那种比较器而已。

### Cloneable接口与clone()方法

#### Cloneable接口

（1）实现了该接口的类，以指示Object的clone()方法可以合法地对该类实例进行按字段复制。

（2）如果在没有实现Cloneable接口的实例上调用Object的clone()方法，则会导致抛出CloneNotSupporteddException

（3）按照惯例，实现此接口的类应该使用公共方法重写Object的clone()方法，Object的clone()方法是一个受保护的方法

#### Object的clone()方法

创建并返回此对象的一个副本。对于任何对象x，表达式：

（1）x.clone() != x为true

（2）x.clone().getClass() == x.getClass()为true

（3）x.clone().equals(x)一般情况下为true，但这并不是必须要满足的要求

#### 浅克隆和深克隆

浅克隆（shallow clone）和深克隆（deep clone）反映的是，当对象中还有对象的时候，那么：

1、浅克隆，即很表层的克隆，如果我们要克隆对象，只克隆它自身以及它所包含的所有**对象的引用地址**

2、深克隆，克隆除自身对象以外的所有对象，包括自身所包含的**所有对象实例**

Object的clone()方法，提供的是一种浅克隆的机制。

可以使用序列化与反序列化实现java深克隆。

### final关键字

#### 作用

（1）修饰于类：该类不可以被继承

（2）修饰于方法：该方法不可以被重写

（3）修饰于变量：该变量（常量）不可以被改变，准确的说，是引用不可改变，引用指向的对象内容是可以改变的。所以如果用final去修饰数组，findbugs会提示“用final修饰数组没有意义”，因为都是引用不可变，引用指向内容可变。

#### 注意点

（1）被final修饰的方法，JVM会尝试为之寻求内联，这对于提升Java的效率非常重要。故如果确定方法不会被继承，尽量将方法定义为final。

（2）被final修饰的常量，在编译阶段会存入调用类的常量池中。

### static关键字

#### 作用

（1）被static修饰的变脸属于类变量，可以通过类名.变脸名直接引用，而不需要new实例调用。

（2）被static修饰的方法属于类方法，可以通过类名.方法名直接引用，而不需要new实例调用。

（3）静态快static{}：静态块的代码只执行一次，且只在初始化类的时候执行。

a.静态资源的加载顺序是严格按照静态资源的定义顺序来执行的。

b. 静态代码块对于定义在它之后的静态变量，可以赋值，但是不能访问。

c. 静态代码块是严格按照父类静态代码块->子类静态代码块的顺序加载的，且只加载一次.

（4）static修饰类：一般用于静态内部类。

（5）import static：用得很少，并且有降低代码可读性的可能，略。

#### 注意点

（1）静态方法能否引用非静态资源？不能，因为实例化的东西在静态资源创建之后，根本不认识。

（2）静态方法能否引用静态资源？可以，因为都是类初始化时加载的。

（3）非静态方法能否引用静态资源？可以，因为静态资源在实例方法之前已创建。

### transient关键字

#### 序列化与反序列化

为什么要有序列化与反序列化？

平时Java在内存中的对象，是无法进行I/O操作的或者网络通信的，对方根本不知道内存中的对象是个什么东西，因此必须将对象以某种方式表示出来，即存储对象的状态。

序列化：将一个对象转换成一串二进制表示的字节数组，通过保存或转移这些字节数据来达到持久化的目的。

反序列化：将字节数组重新构造成对象。

#### 如何实现序列化？

实现java.io.serializable接口。序列化的时候有个serialVersionUID参数，java序列化机制是通过在运行时判断类的serialVersionUID来验证版本一致性。在进行反序列化时，JVM会把传过来的字节流中的serialVersionUID和本地相应实体类的serialVersionUID进行比较，相同则认为是一致的实体类，可以进行反序列化，否则拒绝反序列化并抛出InvalidClassException异常。

serialVersionUID生成方式：

（1）默认的1L。

（2）根据类名、接口名、成员方法以及属性等来生成一个64位的Hash字段。

如果实现serializable接口的实体类没有显式给出serialVersionUID,Java序列化机制会根据编译的.class文件自动生成一个serialVersionUID.如果.class文件没有变化，那么就算编译多次，serialVersionUID也不会变化。换言之，Java为用户定义了默认的序列化、反序列化方法，其实就是ObjectOutputStream的defaultWriteObject方法和ObjectInputStream的defaultReadObject方法。

#### 注意点

（1）序列化之后保存的是对象的状态

（2）transient修饰的属性不会被序列化。

（3）被声明为static的属性不会被序列化，可以理解为序列化保存的是对象信息，但是static修饰的变量是属于类的而不是属于对象的，因此序列化时不会序列化它。

#### 总结

（1）当父类实现serializable接口时，所有子类都可以被序列化。

（2）子类实现了serializable接口，父类没有，父类中的属性不能序列化（不报错，数据丢失），但子类的属性仍能正常序列化。

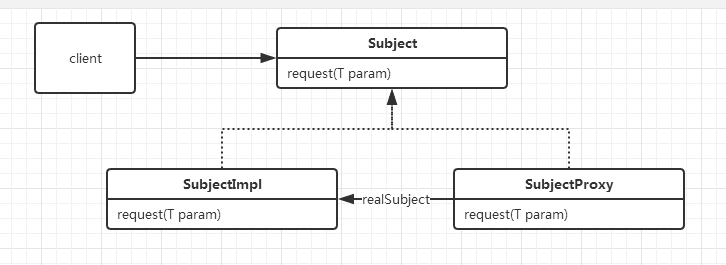
（3）如果序列化的属性是对象，则这个对象也必须实现serializable接口，否则会报错

（4）反序列化时，如果对象的属性有修改或删减，则修改的部分属性会丢失，但不会报错。

（5）反序列化时，如果serialVersionUID被修改，则反序列化时会失败。

### 动态代理

代理uml图大致如下



#### JDK动态代理

JDK动态代理是代理模式的一种实现方式，其只能代理接口，需要实现InvocationHandler接口。

Proxy.newProxyInstance(target.getClass().getClassLoader(),target.getClass().getInterface().this);

##### 如何使用

1、 新建一个接口

2、 为接口创建一个实现类

3、 创建代理类实现java.lang.reflect.InvocationHandler接口

##### 源码分析

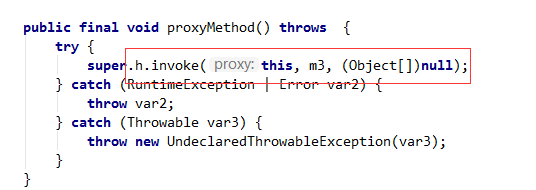
1、代理类继承了Proxy类并且实现了要代理的接口，由于java不支持多继承，所以JDK动态代理不能代理类

2、重写了equals、hashCode、toString

3、有一个静态代码块，通过反射或者代理类的所有方法

4、通过invoke执行代理类中的目标方法，如下图所示为我们实现的InvocationHandler实例。





#### CGLiB动态代理

利用ASM开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类（注意）来处理。

#### 如何使用代理

1）如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP。

2）如果目标对象实现了接口，可以强制使用CGLIB实现AOP。

3）如果目标对象没有实现了接口，必须采用CGLIB库。

Spring会自动在JDK动态代理和CGLIB之间转换：

当Bean实现接口时，Spring就会用JDK的动态代理。当Bean没有实现接口时，Spring使用CGlib实现。但也可以配置<aop:aspectj-autoproxy proxy-target-class="true"/>强制使用CGLIB）。

#### 性能比较

1）使用CGLib实现动态代理，CGLib底层采用ASM字节码生成框架，使用字节码技术生成代理类，

在jdk6之前比使用Java反射效率要高。唯一需要注意的是，CGLib不能对声明为final的方法进行代理，

因为CGLib原理是动态生成被代理类的子类。

2）在jdk6、jdk7、jdk8逐步对JDK动态代理优化之后，在调用次数较少的情况下，JDK代理效率高于CGLIB代理效率，

只有当进行大量调用的时候，jdk6和jdk7比CGLIB代理效率低一点，但是到jdk8的时候，jdk代理效率高于CGLIB代理，

总之，每一次jdk版本升级，jdk代理效率都得到提升，而CGLIB代理消息确有点跟不上步伐。

参考博客

<https://blog.csdn.net/yhl_jxy/article/details/80635012>

### 接口和抽象类区别

（1）构造方法：抽象类有，接口无

（2）成员变量：抽象类可以有普通成员变量，接口只能为常量

（3）方法：抽象类方法可以被static修饰，接口只能为public abstract，不能被static修饰，抽象类可以实现方法也可以以abstract修饰不实现，还可以其他修饰符的方法。

（4）继承：一个类只能继承一个抽象类，但可以实现多个接口。

## Java集合框架

### 介绍下Java中的集合框架（List,Set,Map）?

List：有序、可重复的集合，提供了按索引访问的方式。

Set：无序、不可重复的集合。

Map：键值对、键唯一、值不唯一。

Queue：队列。

其中List与Set均实现Collection接口，Map不符合集合概念，未实现该接口。

### List有哪些实现？如何实现的？如何排序？

个人源码阅读：

JDK8：

#### ArrayList

（1）默认容量为10，有一个size变量存储list大小。

（2）底层为Object数组实现elementData[]，有transeint修饰，但是重写了readObject和writeObject方法，所以还是能序列化。

（3）非线程安全。

（4）默认构造方法创建一个空数组，第一次添加元素，扩展容量为10，扩容（grow）默认是1.5倍（原有容量+原有容量>>1）。

（5）不适合进行删除和插入操作，为了防止数组动态扩充次数过多，建议创建ArrayList时给定初始容量。

#### LinkedList

（1）底层实现为双向循环链表，每一个元素为Node。

（2）同样有size变量记录元素个数

（3）非线程安全。

（4）删除和插入操作效率非常高，但是根据索引查找效率较低。

#### Vector

底层实现与ArrayList大体相似，区别在于方法均有synchronized修饰符，以及扩容算法（增量=0，容量\*2，增量大于0，扩充为容量+增量）是线程安全的。

依然不适合删除和插入。

#### Stack

？

#### CopyOnWriteArrayList

写时复制思想COW（Copy-On-Write）：一开始大家都在共享同一个内容，当某个人想要修改这个内容的时候，才会真正把内从copy出去形成一个新的内容再修改。

COW容器即往该容器添加新的元素时，不直接往容器里添加，而是先将容器进行copy，再在新copy的容器里加入新元素，然后将旧容器的引用指向新容器。这样读取容器时不需要加锁，因为读写是分离的。

实现：

添加元素时使用ReentrantLock加锁。读的时候没有加锁。

应用场景：

读多写少的场景，比如白名单，黑名单之类的，但是并不能保证写入即可读。

缺点：

（1）内存占用：写时会同时驻扎两个对象的内存，内存开销大，可能会引起频繁Full GC，建议使用其他并发容器：ConcurrentHashMap。

（2）数据一致性：只能保证数据最终一致性，但不能保证实时一致性。

#### 排序

使用Collections.sort方法实现正序排序，元素实现Comparable，或者用Comparator来实现比较方法。

### ArrayList与LinkedList区别？

（1）底层存储：ArrayList使用数组实现，LinkedList使用双向循环链表实现。

（2）查询：随机访问（比如索引查找）的效率，ArrayList有下标，性能优于LinkedList。

（3）对于add和remove，LinkedList不需要移动数据，性能优于ArrayList。

总结：对于一个List，大量查操作使用ArrayList，大量增删操作使用LinkedList。

### Set有哪些实现？如何实现的？

#### 如何判断存入Set的对象不相等？

两个对象的equals比较和hashCode比较均相等。

为什么需要比较对象的hashCode？

HashSet

LinkedHashSet

TreeSet

EnumSet

### Map有哪些实现？如何实现的？

#### HashMap

底层为数组+链表+红黑树，链表解决hash冲突，当链表过长（8）时会转为红黑树以实现O(logn)的时间复杂度查找。

（1）值存储

每个键值的存储是Node，包含hash值，key,value,以及下一个Node节点next。hash值为Key的hashCode，用于定位具体存储于数组的何处。

（2）何时由链表转为红黑树？红黑树何时转为链表？

当链表长度超过8时，该链表结构转变为红黑树。为什么是8，可能是经过了性能比较等测试。红黑树个数低于6时，红黑树转换为链表。

（3）数组大小何时扩容？

数组大小默认16。如果元素（Node）个数超过threshold门限值（初始化时16\*0.75（loadFactor负载因子，0.75是空间与时间的一种平衡））就会双倍扩容（数组大小必须是2的n次幂），门限值也同样\*2。

（4）如何确认key应存储于数组的位置？

这里并没有采取取模运算，而是使用了&运算，假如数组长度为16，那么直接取hashCode与15的&运算（高于01111的位数与0取&也只会等于0，相对于取模非常方便），就能够得到0-15之间的二进制数字，但是光取后5位数字的结果，也很有可能取模相同，于是又取高16位和低16位相互异或（^）运算获取一个结果后再与15进行&运算。所以为了0111…的&运算（最后一位二进制数一定不能为0，与0的&操作必然一直为0），所以数组大小一定是2的n次幂，哪怕构造函数传递的参数不是2的n次幂，它依然会强制调整为2的n次幂。

（5）put流程？

0.如果空间不够，先resize()扩容

a.计算key的hash值，经过hash值高16位与低16位的运算，以及与数组大小-1的&运算后获得下标

b.根据下标存放数据，如果没有hash冲突直接放入桶中，如果有，以尾插法塞入链表末尾。

c.如果链表长度超过阈值（8），转为红黑树

d.如果节点已存在就替换旧值并返回旧值。

e.如果桶满了（容量\*负载因子），就要resize扩容（也就是容量与阈值都\*2）。

（6）1.8与1.7的区别

1）链表插入时，1.8是尾插法，1.7是首部，多线程下容易引起死循环。

2）数据扩容后，1.7是直接用hash值和需要扩容的二进制数进行&，1.8是扩容前位置+扩容大小值，后者只需要判断hash值的新增参与运算的位是0还是1（比如16扩容为32，那么只需判断倒数第5位的值是否为1还是0即可）就直接迅速计算了扩容后存储的方式。

3）1.7是数组+单链表，1.8是数组+单链表+红黑树，降低时间复杂度

（7）map用对象做key需要注意什么

必须重写equals()和hashCode()方法。

#### ConcurrentHashMap

（1）初始化数组使用CAS（sc变量）保证线程安全。（HashMap没有考虑初始化安全问题）

（2）如果桶为空Put时也用CAS保证线程安全

（3）如果map正在扩容则当前线程去领取任务协助扩容

（4）剩下hash冲突链表（红黑树）操作时用synchronized(f)，该f为数组元素链表的头结点，降低锁粒度。

HashTable

LinkedHashMap

TreeMap

总结：

chm在1.8中使用cas和synchronized实现线程安全，变量使用volatile来保证get时的数据可见性。和1.7区别为取消了分段锁，使用node实现锁，加入红黑树降低查找时的时间复杂度。

### 如何获取线程安全的集合？

Collections.synchronizedXXX.

### ArrayBlockingQueue

底层为对象数组，使用reentrantlock实现线程安全，使用reentantlock的condition的signal与await实现队列空与满时的阻塞。

## 多线程

### 进程与线程

##### 进程

操作系统分配资源的最小单元，是一段程序的执行过程，是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。

##### 线程

是操作系统调度的最小单元，是独立运行和独立调度的基本单位，一个进程至少得有一个线程。

##### 区别

（1）进程是操作系统分配资源的基本单位，线程是任务调度和执行的基本单位。

（2）一个进程可以拥有多个线程，但必须至少有一个线程，线程是进程的一部分，也被称为轻量级进程。

（3）资源：每个进程有自己独立的内存空间，而线程组只能共享所属进程的资源。

（4）开销：进程间切换开销较大，线程之间切换开销小。

### Java创建线程有几种方式？

（1）实现Runnable接口

（2）继承Thread类，鉴于Java单继承特性，不推荐

（3）实现Callable接口

（4）线程池。

### 线程的状态？

这里可以去参看Thread.State枚举类源码，有描述线程具体状态，共六种。

NEW：初始化，还没有调用start方法。

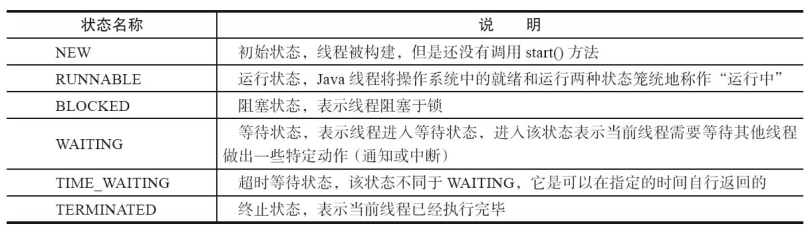
RUNNABLE：运行状态

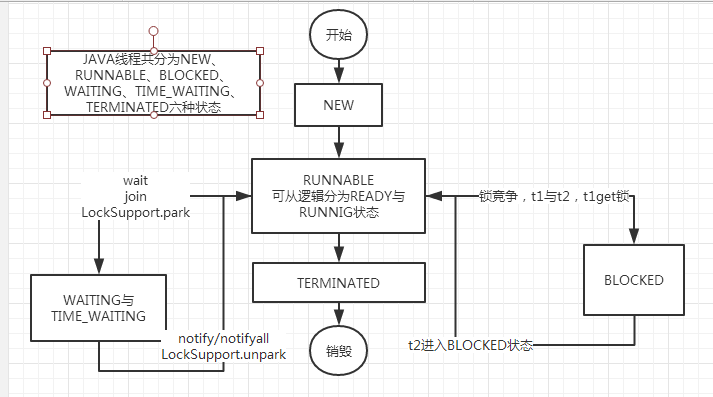
BLOCKED:阻塞状态（等待阻塞，同步阻塞，其他阻塞）

WAITING:等待

TIMED\_WAITING:时间等待

TERMINATED:终止。





### wait()与sleep()区别？

均实现线程的阻塞。

ait需要在同步方法或者同步块使用，不需要捕获异常，会释放锁（最重要）。

sleep任意地方可使用，需要捕获中断异常，不会释放锁。

### wait,notify,notifyAll

wait:实现线程的阻塞。

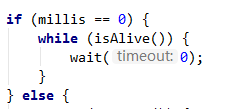
notify/notifyAll：唤醒被阻塞的线程。

### join()

#### 作用

在一个线程中调用另一个线程的join方法，将阻塞该线程直到另一个线程执行结束。

#### 源码实现



假设A线程中调用B.join阻塞A，个人认为此处的意思就是只要B isAlive，将会一直使A一直WAITING，直到B线程结束调用自身的notifyAll()通知其他等待在B线程上的线程。

### 线程的启动和终止

启动：调用线程的start()方法就可以启动线程，start方法含义是当前线程同步告知虚拟机，只要线程规划器空闲，应立即启动调用start()方法的线程。（最好给线程设置名称方便排查）

终止方式：

（1）等待线程自然结束

（2）调用interrupt()方法，改变线程中断标志位，通过判断中断位，优雅地结束线程

（3）设置volatile bool线程中断标志变量，通过逻辑判断优雅地结束线程

### interrupt()与interrupted()区别

interrupt():调用该方法会使当前线程打一个中断标记，并不真的停止线程。意思就是让用户自己根据标记位状态设计程序逻辑优雅地结束线程。也可以使用volatile变量做一个标记位判断线程是否中断，都可以。

在线程受到阻塞时抛出一个中断信号，这样线程就得以退出阻塞的状态。更确切的说，如果线程被Object.wait, Thread.join和Thread.sleep三种方法之一阻塞，那么，它将接收到一个中断异常（InterruptedException），从而提早地终结被阻塞状态。

interrupted():测试当前线程是否已经中断，且清除中断状态。

isInterrupted()：测试线程（**调用该方法的线程**）是否已经中断，不清除中断状态。

注意点：

（1）interrupted是测试当前线程（底层调用currentThread方法），而isInterrupted可以判断其他线程。

（2）二者均调用native方法isInterrupted(boolean clearInterrupted)，只不过前者是true，后者是false。

### ThreadLocal

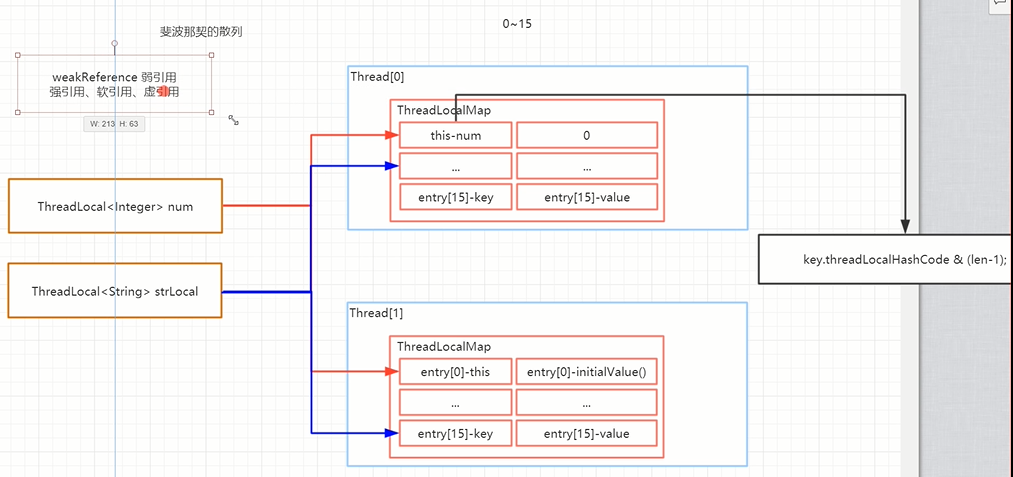
#### 作用

线程本地存储，为变量在每个线程中都创建了一个副本，每个线程可以访问自己内部的副本变量。

#### 实现

内部使用ThreadLocalMap（Thread的一个局部变量）进行存储，key为ThreadLocal，value为具体存储变量副本。

get时，获取当前线程，找到对应的ThreadLocalMap获取其value，如果没有对应的值则调用initialValue方法返回初始值。



#### 注意事项

关于ThreadLocalMap<**ThreadLocal**, **Object**>弱引用问题：

当线程没有结束，但是ThreadLocal已经被回收，则可能导致线程中存在ThreadLocalMap<**null**, **Object**>的键值对，造成内存泄露。（ThreadLocal被回收，ThreadLocal关联的线程共享变量还存在）。

虽然ThreadLocal的get，set方法可以清除ThreadLocalMap中key为null的value，但是get，set方法在内存泄露后并不会必然调用，所以为了防止此类情况的出现，我们有两种手段。

1、使用完线程共享变量后，显示调用ThreadLocalMap.remove方法清除线程共享变量；

2、JDK建议ThreadLocal定义为private static，这样ThreadLocal的弱引用问题则不存在了。

内存泄漏的原因：

由于ThreadLocalMap的生命周期跟Thread一样长，如果没有手动删除key对应的value就会导致内容泄漏，而不是因为弱引用。

一般使用时会重写其initialValue()方法。

ThreadLocal<Integer> threadVarCache = ThreadLocal.*withInitial*(()->0);

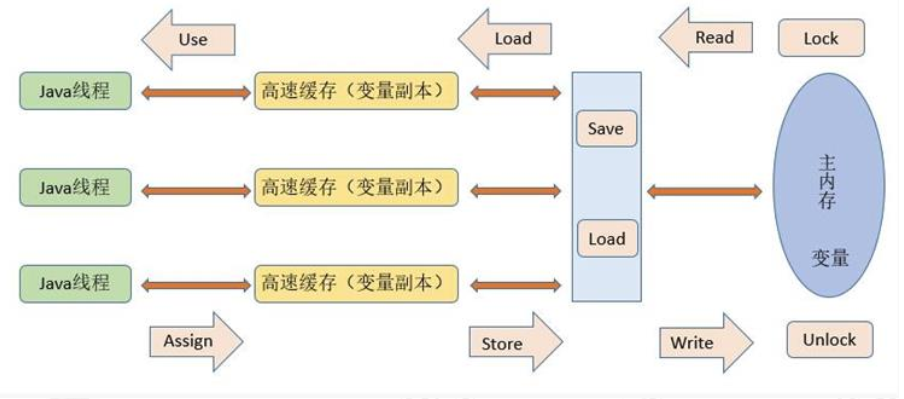
参考博客

https://www.cnblogs.com/coshaho/p/5127135.html

### JMM（Java Memory Model）Java内存模型

JMM是一种规范，定义了线程和内存的交互方式，在JMM抽象模型中，分为主内存工作内存。主内存是线程共享的，工作内存（高速缓存、CPU等）是每个线程独有的。线程对变量的所有操作（读取、赋值）都必须在工作内存中执行，不能直接读写主内存中的变量。并且不同的线程之间无法访问对方工作内存中的变量，线程间的变量值的传递都需要通过主内存完成。

目的是为了解决由于多线程通过共享内存进行通信时，存在的本地内存数据不一致、编译器会对代码指令重排序、处理器会对代码乱序执行等带来的问题。目的是保证并发编程场景中的原子性、可见性和有序性（高速缓存、重排序带来的可见性问题）。



那么线程之间如何通信？

假设线程A与B要通信，必须经过如下2个步骤：

（1）线程A把线程A独占内存中更新的共享变量（副本）刷新到主存中。

（2）线程B将主存中已更新过的共享变量读取到线程B独占内存（副本）中。

### volatile关键字

#### 作用

（1）保证不同线程对该变量操作的内存可见性

（2）禁止指令重排序。

注意，该关键字并不能保证变量复合操作的原子性。

#### 什么是内存可见性？

根据JMM，在线程执行时，首先会从主内存（堆）中read变量值，再load到工作内存（可以理解为栈，也可以理解为高速缓存orCPU上的寄存器）中的副本中，然后再传给处理器执行，执行完毕后再给工作内存中的副本赋值，随后工作内存再把值传给主内存，主存中的值才会更新。

简单来说就是某个线程处理一个主存中的值时，会经历read副本from主存->工作线程的独占内存->CPU->工作线程的独占内存->save到主存。

假如有两个线程对i（假设值为0）执行i++，理论上最终结果i值会变成2，但是如果其中一个线程从主存read值的时候，i还没被另一个线程刷新成1，那么最终结果可能还是1，如果从工作内存回写至主内存的速度慢一些，甚至还会读成0，这就是缓存不一致问题。

所以总结一下，内存可见性就是指一个线程对共享变量值的修改（主存），能够及时地被其他线程看到。

#### 什么是指令重排序？

指令重排序是编译器或处理器为了提高程序性能而做的优化，对指令序列进行重新排序的一种手段，会出现代码书写顺序与实际执行顺序不同的情况。有以下三种：

（1）编译器优化的重排序（编译器优化）

（2）指令级并行重排序（处理器优化）

（3）内存系统的重排序（处理器优化）

指令重排序也可能会造成内存可见性问题。

#### 解决缓存不一致问题的三个特性（并发编程三大特性）

原子性（Atomicity）：原子性指不可中断，要做一定做完，要么完全不做。Java中只有赋值和读取是原子操作，比如i++就不是原子操作。

可见性（Visibility）：意思就是拥有可见性（volatile修饰）的变量，对它的修改会立刻刷新到主存，其他线程能够读到的绝对是最新值。synchronized和Lock也能做到。

**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。**

有序性（Ordering）：意思就是尽管存在指令重排序，但是JMM规定了as-if-serial语义，即单线程中不管怎么重排序，程序的执行结果不能改变。**即程序执行的顺序按照代码的先后顺序执行。**

#### volatile底层是怎么做的？满足了几个特性？

加了volatile关键字的变量，汇编底层会在执行写操作的时候加入lock前缀的指令，将这个变量所在缓存行的数据写回到系统内存，其他线程根据缓存一致性协议，嗅探在总线上传播的数据是否过期，如果发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置为无效状态，当处理器需要对这个数据进行修改操作时，会重新从主内存中把数据读到处理器缓存。

（1）Lock前缀指令会引起处理器缓存会写到内存。

（2）一个处理器的缓存回写到内存会导致其他处理器的缓存无效。

底层是通过加入内存屏障（Memory Barriers）来实现内存可见性的（包含禁止指令重排序）。当volatile变量执行写操作时，会在写操作前加入一条storestore barrier屏障指令，写后面的操作插入storeload barrier指令，将该线程对应的本地内存中的共享变量刷新到主内存；当对volatile变量执行读操作时，会在读操作后加入一条loadload barrier屏障指令，读操作后面插入loadstore barrier，线程接下来将从主内存读取共享变量。

volatile只能满足可见性和有序性，不保证原子性，volatile禁止指令重排序以达到可见性效果。



#### volatile使用范例？

（1）状态量标记，比如作为一个线程是否应当被停止之类的状态标记。

（2）单例中双重校验锁，修饰单例变量。

#### 与synchronized比较？

（1）性能：volatile比synchronized更轻量级（轻量级锁），不会阻塞线程，效率更高。

（2）特点：synchronized满足以上三种特性，标记的变量可以被编译器优化（重排序），而volatile满足内存可见性和有序性，不满足原子性。顺带一提final满足可见性。

（3）作用域：synchronized可以作用于变量，方法和类，volatile只能作用于变量。

#### 内存屏障在JMM中的定义（编译器层面）

storestore barrier

storeload barrier

loadload barrier

loadstore barrier

内存屏障是一组处理指令，用来实现对内存操作的顺序限制。volatile的底层就是通过内存屏障来实现的。

#### MESI协议（CPU层面）

modify,exclusive,shared,invalid。用于解决工作内存与主内存的缓存一致性问题，硬件层面的实现。

CPU乱序执行->重排序->可见性问题

CPU层面提供了指令->内存屏障

内存屏障用于解决可见性问题。CPU层面提供了读屏障，写屏障，全屏障

storebarrier, loadbarrier,fullbarrier。

volatile中lock汇编指令相当于内存屏障。

#### happens-before

as-if-serial:不管怎么重排序，对于单个线程的执行结果不会改变。

1. 同一个线程中的，前面的操作 happen-before 后续的操作。（即单线程内按代码顺序执行。但是，在不影响在单线程环境执行结果的前提下，编译器和处理器可以进行重排序，这是合法的。换句话说，这一是规则无法保证编译重排和指令重排）。
2. 监视器上的解锁操作 happen-before 其后续的加锁操作。（Synchronized 规则）
3. 对volatile变量的写操作 happen-before 后续的读操作。（volatile 规则）
4. 线程的start() 方法 happen-before 该线程所有的后续操作。（线程启动规则）
5. 线程所有的操作 happen-before 其他线程在该线程上调用 join 返回成功后的操作。
6. 如果 a happen-before b，b happen-before c，则a happen-before c（传递性）。

https://www.cnblogs.com/chenssy/p/6379280.html

### synchronized关键字

#### 作用

解决原子性、可见性和有序性问题，使线程依次排队操作共享变量。

#### 使用场景



#### 锁对象？

普通方法：锁是当前实例对象。

静态方法：锁是当前类的Class对象。

同步方法块：锁是synchronized括号里配置的对象。

#### 底层实现？

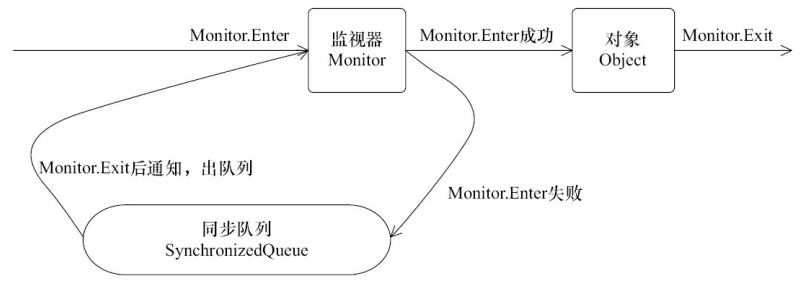
monitorenter monitorexit \* 2

JVM基于进入和退出Monitor对象来实现方法同步和代码块同步，主要是monitorenter和monitorexit指令。

monitorenter在编译后插入到同步代码块开始的位置，monitorexit插入到方法结束和异常处。JVM要保证每个monitorenter必须有对应的monitorexit与之配对。任何对象都有一个monitor与之关联。当一个monitor被持有后，它将处于锁定状态。线程执行到monitorenter指令时，将会尝试获取对象所对应的monitor的所有权，即尝试获取对象的锁。

同步方法则是依靠方法修饰符的ACC\_SYNCHRONIZED实现。

本质是对一个对象的监听器进行获取，而获取过程是排他的，也就是同一时刻只能有一个线程获取到由synchronized所保护对象的监视器。未获取到监视器的线程调用同步代码块或者同步方法时，将会阻塞在入口处，进入BLOCKED状态。



#### 为什么任何一个对象都可以成为锁？

对象头，ObjectMonitor。

#### 锁存在哪？

Java对象头。

#### 锁的4种状态（JDK1.6开始，级别由低到高）

锁一共4种状态，无锁状态、偏向锁状态、轻量锁状态、重量锁状态，并且这几个状态会随着竞争情况逐渐升级。锁可以升级但不能降级，目的为了提高获取锁和释放锁的效率。

##### 偏向锁

大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低而引入了偏向锁。

偏向锁就是当一个线程访问同步块并获取锁时，如果是第一次访问（锁第一次被拥有），会在对象头锁记录里存储锁偏向的线程ID，以后每次同步，检查锁的偏向线程ID是否与当前线程ID一致，如果一致则进入同步，退出同步也无需CAS更新对象头，如果不一致则发生了竞争，说明不再偏向于某一个线程，需要升级为轻量级锁，才能保证线程间公平竞争锁。

偏向锁的加锁在偏向线程第一次进入同步块时，CAS原子操作尝试更新对象的Mark Word（偏向锁标志位为1，并记录偏向线程ID）。

偏向锁的撤销使用了一种等到竞争出现才会释放锁的机制。所以当其他线程竞争锁时，持有偏向锁的线程才会释放锁。撤销需要等待全局安全点（无正在执行的字节码的时间点），首先暂停当前持锁线程，检查偏向锁线程是否活着，未活着将对象头设置为无锁状态，活着，先执行拥有偏向锁的栈，遍历偏向对象的锁记录，栈中的锁记录和对象头的Mark Word要么重新偏向于其他线程，要么恢复到无锁或者标记对象不适合作为偏向锁，最后唤醒暂停的线程。

总结：在没有竞争（只有一个线程访问）的情况下，markword偏向锁标记1，记录线程id，每次访问只需比较id是否相同即可，偏向锁一般是关闭的。

##### 轻量级锁（自旋锁）

在两个线程相互竞争，且锁持有时间非常短的情况下，轻量级锁使线程不断while(true)循环使用if(cas)尝试获取锁，相对于线程挂起来说反而是一种性能优化，因为线程从运行态切换到内核用户态，开销是大于这种死循环竞争锁的。

自旋会占用CPU，如果锁持有时间较长就会浪费CPU资源，所以在指定的自旋次数之后还未获得轻量级锁会转为重量级锁。

设置自旋次数 preBlockSpin，默认为10.JVM参数

还有自适应自旋。

##### 重量级锁

升级到重量级锁之后，没有获得锁的线程会被阻塞。基于Monitorenter与moniterexit指令实现。monitorenter成功表示当前线程获得对象锁，没有获得成功的线程将会加入到一个阻塞队列中（可以这么理解），直到monitorexit执行后随机（非公平锁）唤醒一个，然后再次去争抢锁。

##### 流程总结

（1）无锁->偏向锁

线程1访问同步块，检查对象头是否存储了该线程，没有则用CAS替换markword，成功后转变为偏向锁，markword记录当前线程id，偏向锁位置1，以后该线程访问只需比较线程id是否一次即可进入同步块，也不用替换Markword。

（2）偏向锁->轻量级锁

线程2访问同步块，检查对象头是否存储线程2，没有就使用CAS替换markword，这时Markword存储了线程1，替换失败。

撤销偏向锁，暂停线程，将线程id设置为空（恢复到无锁状态），恢复线程，如果线程1还未执行完，则升级为轻量级锁。

（3）轻量级锁->重量级锁

指定次数后线程还未获得轻量级锁，则膨胀为重量级锁。

锁升级是不能降级的。

#### 其他锁概念

##### 自旋锁（spin lock）

自旋锁是一种非阻塞锁，是指当一个线程在获取锁的时候，如果锁已经被其它线程获取，那么该线程将循环等待，然后不断的判断锁是否能够被成功获取，直到获取到锁才会退出循环（while(true){if(cas){…}}）。

为什么要用自旋锁？

互斥锁有一个缺点，他的执行流程是这样的 托管代码  - 用户态代码 - 内核态代码、上下文切换开销与损耗，假如获取到资源锁的线程A立马处理完逻辑释放掉资源锁，如果是采取互斥的方式，那么线程B从没有获取锁到获取锁这个过程中，就要用户态和内核态调度、上下文切换的开销和损耗。所以就有了自旋锁的模式，让线程B就在用户态循环等着，减少消耗。

自旋锁比较适用于锁使用者保持锁时间比较短的情况，这种情况下自旋锁的效率要远高于互斥锁。

参考博客：

<https://blog.csdn.net/qq_34337272/article/details/81252853>

https://blog.csdn.net/u010372981/article/details/81463201

##### 互斥锁（mutex）

是阻塞锁，当某线程无法获取锁时，该线程会被直接挂起，该线程不再消耗CPU时间，当其他线程释放锁后，操作系统会激活那个被挂起的线程，让其投入运行。

### CAS（Compare And Swap）原理

CAS定义：如果当前状态值等于预期值，则以原子方式将同步状态设置为给定的最新值。内存值V，预期值A，修改值B，若想B覆盖V，则覆盖之前需比较V是否等于A，相等则覆盖，否则拒绝。

CAS的ABA问题，一般使用版本号的方式来解决。

ABA问题即若V已被其他线程改成C，然后又改成了V，此时当前线程依然会误判成功，以为值未做改变。

https://blog.csdn.net/qq\_32998153/article/details/79529704

### forkjoinpool？

### Lock（主要是ReentrantLock）

#### ReentrantLock功能

锁类型

* 可重入锁：在执行对象中所有同步方法不用再次获得锁
* 可中断锁：在等待获取锁过程中可中断
* 公平锁： 按等待获取锁的线程的等待时间进行获取，等待时间长的具有优先获取锁权利
* [读写锁](https://www.baidu.com/s?wd=%E8%AF%BB%E5%86%99%E9%94%81&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)：对资源读取和写入的时候拆分为2部分处理，读的时候可以多线程一起读，写的时候必须同步地写。

ReentrantLock是基于Java层面实现的可重入锁，可以选择是否公平。

#### 与synchronized区别

（1）实现：Lock是Java代码实现，而synchronized是基于JVM实现。

（2）使用：reentrantlock相比于synchronized，实现了后者所有功能，并且更灵活，但是必须在finally处释放，后者只需程序执行完毕或者异常释放锁。

（3）reentrantlock包含各种condition，在线程间通讯更具灵活性。

（4）类型：Lock可重入，锁状态可判断，可公平，等待可中断；synchronized只能可重入，锁状态可判断，非公平，等待不可中断  
（5）性能：reentrantlock在1.6之前性能更优，而后者在1.6以后进行了优化，性能逐渐追了上来

公平锁：就是阻塞等待的线程，并不是随缘获取锁，而是根据先来先到的顺序获取锁

等待可中断：意思是获取锁失败的线程可以去做其它事，而不像synchronized，获取失败就一直阻塞直到成功

#### ReentrantReadWriteLock

共享锁（同一时刻可以有多个线程获得锁），适用于读多写少。比如在处理缓存的读取和更新时，就可以用读写锁。读的时候加读锁，写的时候加写锁。

### AQS（AbstractQueueSynchronizer）同步工具

1.独占->互斥锁

2.共享->读写锁

#### 基本实现

使用双向链表记录阻塞线程（封装成Node），抢锁成功的线程执行，未成功的线程加入该链表并启用死循环（自旋）尝试抢锁，第一次抢锁失败直接挂起（LockSupport.park，这种阻塞方法无法响应中断（抛出Interrrupt异常）），等待抢锁成功的线程lock.unlock释放锁后触发LockSupport.unpark唤醒头结点的线程（非公平锁不一定，拼cas操作的速度），抢锁成功后释放头结点，该节点转换为空节点作为新的头结点。

还有一个Condition队列（单向链表，记录了头和尾），当Lock中condition调用await时，会将当前线程用locksupport.park给挂起，然后将其加入到该队列中。当另一个线程signal/signalAll时，会将该队列的头结点转移到aqs的阻塞队列中，直到aqs队列调用locksupport.unpark的释放锁。

这个实现跟synchronized的实现很类似。

（1）Node

双向链表节点，包含waitStatus，前后节点，线程（被封装成Node的那位），nextWaiter（下一个等待线程Node）成员，SHARED与EXCLUSIVE两个Node属性分别表示共享锁和独占锁。

每次只唤醒一个节点，后一个节点（nextWaiter）处于等待状态。



（2）LockSupport

提供park与unpark（使用Unsafe）方法实现线程的挂起与取消挂起。

### 线程池

#### 作用

（1）控制线程数量

（2）降低频繁创建和销毁线程

（3）对于任务响应速度更快。

#### 创建

（1）Executors的5种线程池工厂方法（Fixed、Cached、Single、Scheduled、workStealing）。最好不要直接调用这些方法，尤其是前4个，他们的阻塞队列默认值没有设置元素大小，有内存撑爆的可能。

（2）直接调用ThreadPoolExecutor构造方法生成线程池。

#### ThreadPoolExecutor

Executors创建线程池几乎都用到了该API。

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

corePoolSize：核心线程数。

maximumPoolSize：最大线程数。

keepAliveTime：超时时间，即超过核心线程数的线程在无任务状况下的最大等待时间，超过即回收。

unit：超时时间单位

workQueue：阻塞队列

threadFactory：线程工厂

RejectedExecutionHandler：线程拒绝策略。

##### 流程

（1）提交任务，为空则抛出空指针异常

（2）如果工作线程数未超过核心线程数，创建核心worker线程执行任务（不停自循环调用runWorker）

（3）否则添加到阻塞队列，成功即等待处理，失败则判断运行线程数是否超过最先大线程数，不超过则创建一个非核心worker线程，该线程用完后存活keepAliveTime后销毁。

（4）以上都不能满足则执行拒绝策略。

核心线程在未工作时会阻塞。核心线程和非核心线程的区别就是正式工和临时工的区别。正式工一直常驻内存，临时工干完活就滚蛋。一开始核心线程和最大线程都不存在，预热（开始execute任务，有prestartAllCoreThreads方法预热所有核心线程）后创建核心线程并执行任务（thread.start），如果线程过多阻塞队列里也有线程的话核心线程也需要执行完第一次提交的任务后再从队列获取任务执行，这样就达到了复用效果（典型的生产者消费者模式）。线程运行结束后即被回收。

#### keepAliveTime怎么去监控多余的最大线程回收的？

实际上就是从队列里poll的超时时间，超过这个时间了线程自己就结束了，即被回收。

#### shutdown()与shutdownNow()

前者是打上中断标记，等待线程运行完毕，后者类似于kill -9强制关闭。

#### 线程池大小设置？

取决于硬件与软件环境

（1）CPU核心数，一般核心数+1。

（2）IO密集型：消耗时间较多，CPU时间片切换频繁，可以设置更多线程，可以为核心数2倍。

CPU密集型：执行以计算为主，时间较短，CPU利用率很高，设置线程数就不要过多，。一般以CPU核心数为准。

公式：（线程等待时间+线程CPU时间）/线程CPU时间\*CPU核心数

改良为(线程等待时间/线程CPU计算时间+1)\*CPU核数。

最好还是以压测结果为准。

#### 拒绝策略

AbortPolicy：默认策略，丢弃任务并抛出RejectedExecutionException异常。

DiscardPolicy：丢弃任务无任何异常。

DiscardOldestPolicy：丢弃最早进入队列的任务。

CallerRunsPolicy：使用此策略，如果添加到线程池失败，那么主线程会自己去执行该任务，不会等待线程池中的线程去执行。就像是个急脾气的人，我等不到别人来做这件事就干脆自己干。

**public void** rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
 **if** (!e.isShutdown()) {  
 r.run();  
 }  
}

## Mysql

### 事务

#### 数据库事务

ACID

Atomic（原子性）：指事务的操作要么全部执行成功，要么不执行

Consistency（一致性）：事务完成时，所有数据必须保持一致性（比如3个100元账户相互转账，最终结果总额还是300，不会变化，而且符合规则，比如个人资产不会低于0）

Isolution（隔离性）：并发情况下事务所做的修改与其他事务所做的修改相互隔离

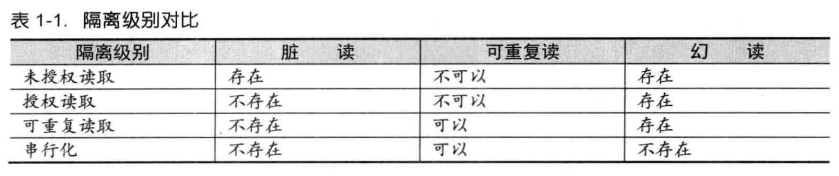
4个隔离级别：

读未提交（Read Uncommitted）:允许脏读。

读已提交（Read Committed）：只允许读取已经被提交的数据。

可重复读取（Repeatable Read）：保证在事务处理过程中，多次读取同一个数据时，其值都和事务开始时刻一致的。可能出现幻读。

串行化（Serilizable）：要求所有事务都被穿行执行，不能并发执行。





InnoDB通过临键锁（写）+mvcc（读）共同解决了幻读的问题。

Duration（持久性）：事务完成后对系统的更改是永久性的。

脏读与幻读与不可重复读：

关键字：幻读，insert。不可重复读，update和delete。

幻读和不可重复读都是同一个事务内，两次读取范围的记录，第一次与第二次之间读取数据间隔中由于另一个事务对范围记录做了修改所产生。

幻读是两次读取中第二次多了若干行数据。

不可重复读是第二次读时已读数据行被做了修改，读到的数据与第一次不一致（已提交）。

脏读是第一个事务读到另一个事务做了修改但却没有提交的临时数据（另一个事务改了又回滚了），导致读取结果与数据表不一致。

Mysql事务操作

事务日志，redo,undo

1、记录undo和redo文件，确保日志在磁盘上的持久化。

2、更新数据记录

3、提交事务，redo写入commit记录

#### 分布式事务

##### 场景

（1）数据库分库分表，数据位于不同数据库

（2）soa架构，服务拆分

##### X/OpenDTP事务模型

X/Open Distributed Transaction Processing Reference Model

X/Open组织定义了一套分布式事务标准，定义了规范的API接口。

Java ee遵循该规范，设计并实现了java里面的分布式事务编程接口规范JTA。

XA是X/Open DTP定义的中间件与数据库之间的接口规范，XA接口函数由数据库厂商提供。

##### X/OpenDTP角色

AP application

RM resource manager

TM transaction manager

### 2PC（Two-Phase Commit）

二阶段提交，应用于分布式系统事务，将事务提交分成二个阶段进行处理:

#### 阶段一：提交事务请求

（1）事务询问

协调者向所有的参与者发送事务内容，询问是否可以执行事务提交操作，并开始等待各参与者响应。

（2）执行事务。

各参与者节点执行事务操作，并将Undo和Redo信息记入事务日志中。

（3）各参与者向协调者反馈事务询问的响应

如果参与者成功执行了事务操作，反馈给协调者yes响应，表示事务可以执行，否则返回no，表示事务不可以执行。

#### 阶段二：进行事务提交

有两种情况：

假如协调者获取的响应均为yes，会执行事务提交。

（1）发送提交请求

协调者向所有参与者节点发出commit请求。

（2）事务提交

参与者接收到commit请求后，会正常执行事务提交操作，并在完成提交之后释放在整个事务执行期间占用的事务资源。

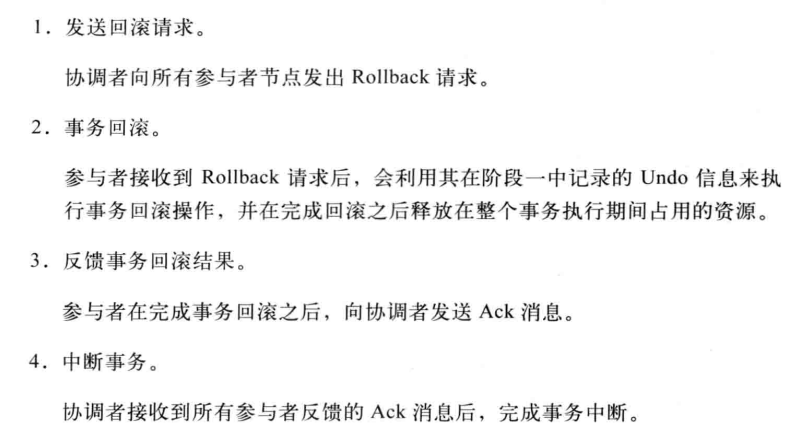
（3）反馈事务提交结果

参与者在完成事务提交之后，向协调者发送ACK消息。

（4）完成事务

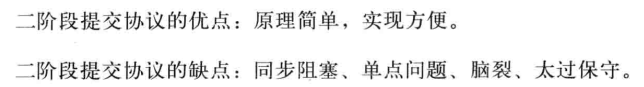
协调者接收到所有参与者反馈的ack消息后，完成事务。

假如任何一个参与者向协调者反馈了no，或者在等待超时之后，协调者尚无法接收到所有参与者的反馈响应，则会执行中断事务。



总之2PC将一个事务的处理过程分为了投票和执行两个阶段，核心是对每个事务都采用先尝试后提交的处理方式，因此也可以将二阶段提交看做一个强一致性算法。

#### 优缺点

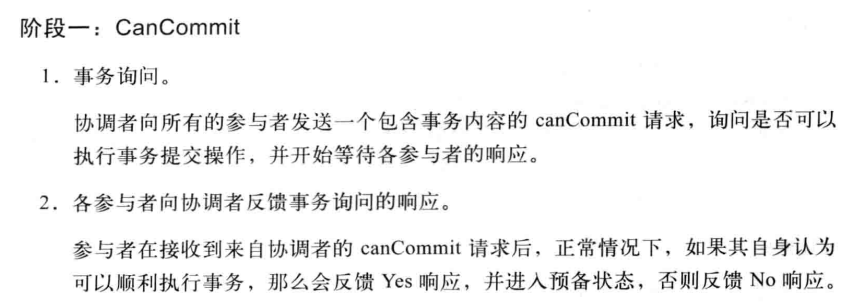


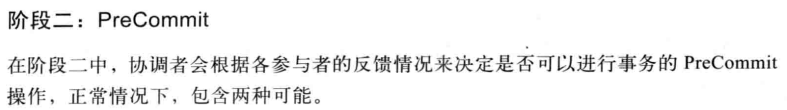
其他：

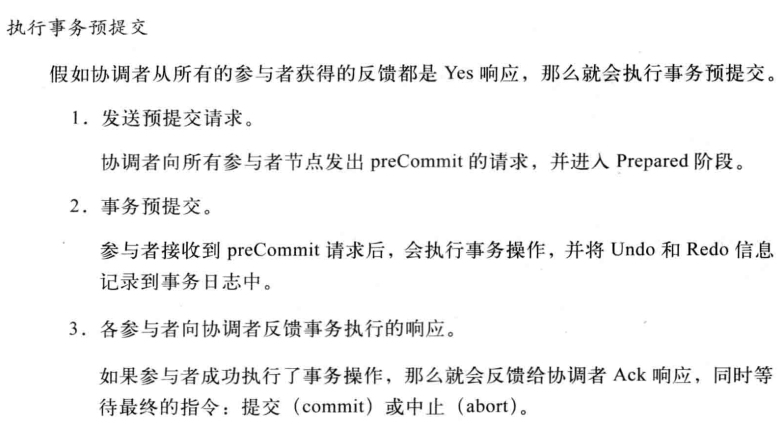
（1）数据一致性问题在tm和ap偶尔挂掉的情况下无法解决。

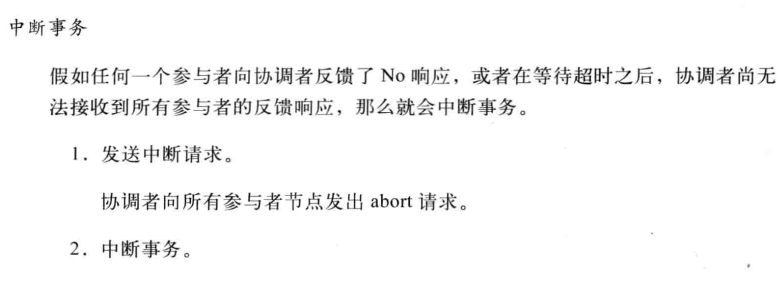
### 3PC（Two-Phase Commit）

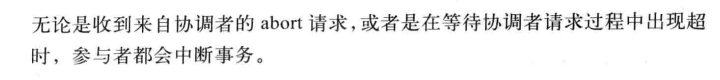
三阶段提交，将2PC的“提交事务请求”过程一分为二，形成CanCommit,PreCommit和doCommit三个阶段组成的事务处理协议。

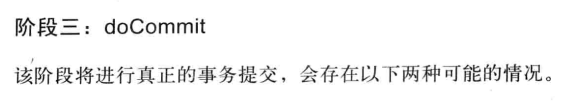


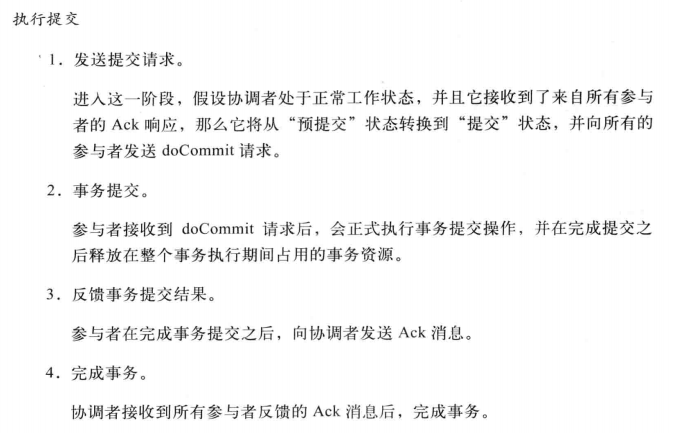


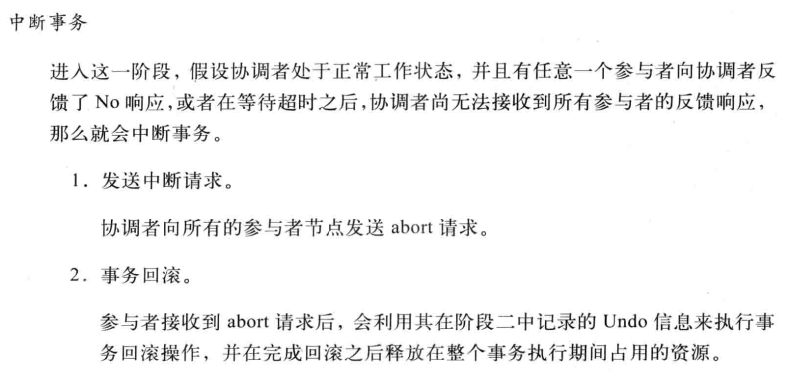


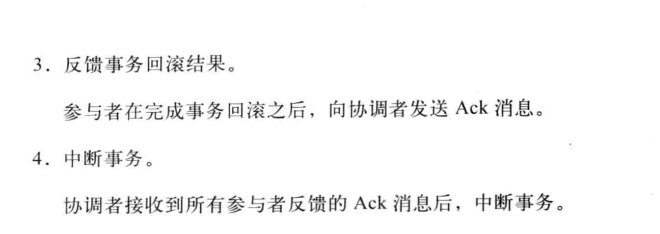


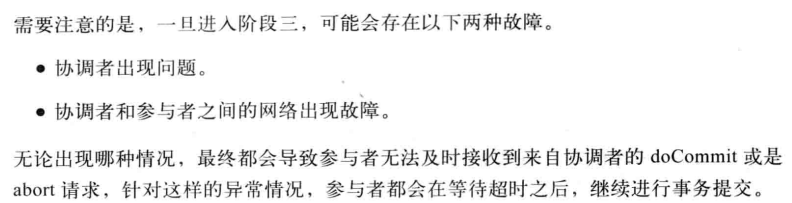












### 分布式事务的实现

JOTM（Java Open Transacetion Manager）基于JTA实现。

Atomikos（开源，集成spring）

### 存储引擎

（1）存储引擎是指定在表之上的，即一个库的每一个表都可以指定专用的存储引擎。

（2）不管表采用什么样的存储引擎，都会在数据区，产生对应的一个frm文件（表结果定义描述文件）。

（3）插拔式的插件方式。

#### InnoDB存储引擎

从Mysql5.5.8开始，为默认存储引擎。

##### 特点

（1）支持事务

（2）行锁设计（读操作不会产生锁），支持外键

（3）使用多版本并发控制（MVCC）获得高并发性，实现sql标准隔离级别，默认为REPEATABLE。同时使用一种被称为next-keylocking的策略来避免幻读（phantom）现象的产生。

（4）对于表中数据采用聚集（clustered）方式，每张表存储都是按主键顺序进行存放，如果没有显式表定义时指定主键，会为每一行生成一个6字节的ROWID，并以此作为主键。

#### MyISAM存储引擎

Mysql5.5版本之前的默认存储引擎。

较多的系统表也还是使用这个存储引擎。系统临时表也会用到MyISAM存储引擎。

##### 特点

（1）不支持事务

（2）表锁设计。

（3）支持全文索引。

（4）数据（MYD）和索引（MYI）分开存储，缓冲池只缓存（cache）索引文件，不缓存数据文件。

（5）select count(\*) from table 无序进行数据扫描

#### CSV存储引擎

数据存储以CSV文件。

##### 特点

（1）不能定义索引、列定义必须为NOT NULL、不能设置自增列

->不适用大表或者数据的在线处理。

（2）CSV数据的存储用”,”隔开，可直接编辑CSV文件进行数据的编排

->数据安全性很低。

编辑之后，要生效使用flush table XXX命令。

##### 应用场景

数据的快速导入导出，表格直接转换为CSV。

#### Archive存储引擎

压缩协议进行数据的存储，数据存储为ARZ文件格式。

##### 特点

（1）只支持insert和select两种操作。

（2）只允许自增ID列建立索引。

（3）行级锁。

（4）不支持事务。

（5）数据占用磁盘少。

##### 应用场景

日志系统，大量的设备数据采集（不过还是用大数据好）。

#### 其他引擎暂略

### 索引

#### 概念

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](https://baike.baidu.com/item/%E6%8C%87%E9%92%88/2878304)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](https://baike.baidu.com/item/%E9%A1%B5%E7%A0%81/7716178)快速找到所需的内容。

总结：索引是为了加速对表中数据行的检索而创建的一种分散存储的数据结构。

语法略。

#### 实现

底层为B+Tree实现（InnoDB），B+Tree特点就是数据全在叶子节点，且每一个叶子节点都包含指向下一个叶子节点的指针，方便叶子节点的遍历，同时还是个平衡树（叶子节点全在同一层）。

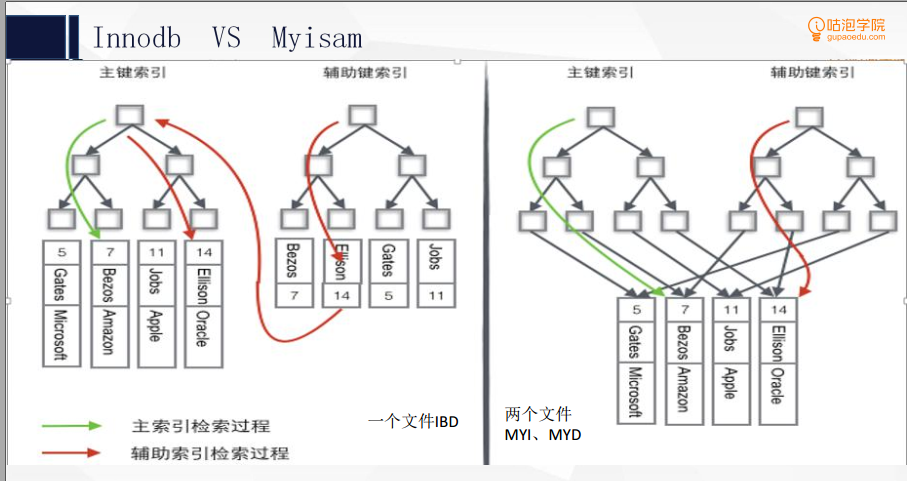
查找效率提升的原理就是查找数据时，从根节点开始往下查找，每读一个节点进行一次IO，IO次数只与树的高度有关，如果是二叉树的话根节点选择不好很可能会变成线性结构，导致每次全部查询。所以平衡树大大降低了IO次数，平衡树的高度越低则IO次数越少，效率更高。

为什么数据都放在叶子节点而不放在树枝节点呢？这个估计跟设计者考虑性能稳定性有关，不一定追求极致的数据查询，而是所有查询的IO次数都在一定范围内，不一会特别快一会比较慢，有利于性能评估。

InnoDB的主键索引叶子节点直接关联数据。

InnoDB的辅助（二级）索引的数据叶子节点直接关联主键索引，记录的是主键索引的ID。

如下图即可理解：



所以InnoDB的查询需要访问两次索引，InnoDB的查询除了覆盖索引外最终都会走到主键索引查询里。。

#### 联合索引

实质上也就是将多个列的值拼接起来，根据大小比较排序。所以单列索引只是特殊的联合索引。要注意最左匹配原则，左边的列是范围查询，那么右边列的索引将会失效。

联合索引列选择原则：

（1）经常用的列优先（最左匹配原则）

（2）选择性（离散度）高的列优先（离散度高原则）

（3）宽度小的列优先（最少空间原则）

#### 聚簇索引

就是指主索引文件和数据文件为同一份文件，以主键索引来组织数据的存储，数据库表中数据的物理顺序和键值的逻辑（索引）顺序相同。聚簇索引主要用在Innodb存储引擎中。在该索引实现方式中B+Tree的叶子节点上的data就是数据本身，key为主键，如果是一般索引（非聚簇索引）的话，data便会指向数据存放的地址。非聚簇索引比聚簇索引多了一次读取数据的IO操作，所以查找性能上会差。

#### 覆盖索引

如果查询列可通过索引节点中的关键字直接返回，则称该索引为覆盖索引。

覆盖索引可减少数据库IO，将随机IO变为顺序IO，提高性能，尤其是InnoDB中如果覆盖索引能够直接在辅助索引中找到值，则可避免再次根据辅助索引的叶子节点的主键id去二次查找聚集索引中的行数据，避免对主键索引的二次查询。

覆盖索引必须要存储索引列的值，而哈希索引、空间索引和全文索引不存储索引列的值，所以mysql**只能用B-tree索引**做覆盖索引。

#### 主键索引与二级索引

**聚簇索引（主键索引）：**索引数据和存储数据都在同一颗树上，比如根据id查找，只要找到该主键就找到数据了。

**二级索引（辅助索引）：**比如根据name查找某个商品，就会先找到这个商品对应的主键id,然后根据id再去查找该商品。

#### 作用（优点）

（1）极大较少存储引擎需要扫描的数据量，大大加快数据检索速度

（2）把随机IO变成顺序IO

（3）在使用分组和排序子句进行数据检索时，可以显著减少查询中分组和排序的时间，避免使用临时表。

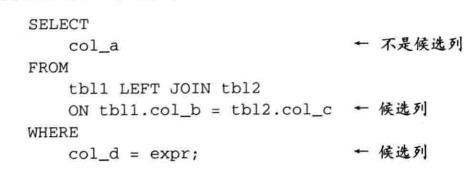
#### 缺点

（1）更多的物理空间占用

（2）对表数据进行增删改时，索引也需要动态维护，降低了数据的维护速度。

#### 如何挑选索引？

（1）为用于搜索、排序或分组的列创建索引，而对于用作输出显示的列则不用创建索引。例如：



（2）认真考虑数据列基数。列基数指它所容纳的所有非重复值的个数。相对于表里行的总数来说，列的基数越高（唯一值多，重复值少），索引的使用效果越好。比如性别这一列，不出意外只有两个值，或者1或0，使用索引效率就非常烂。

（3）索引短小值。应尽量选用较小的数据类型。

（4）索引字符串值得前缀，想要对字符串列进行索引，应当尽可能指定前缀长度。

（5）利用最左前缀。适用于复合索引（多个列的索引）中最左边的那个索引。当创建包含n个列的复合索引时，实际上会创建n个专供mysql使用的索引。最左边的任意数据列集合都可以用于匹配各个行，这样的集合即为“复合前缀”。

（6）不要建立过多的索引。

（7）让参与比较的索引类型保持匹配。

（8）利用慢查询日志找出那些性能低劣的查询。

#### InnoDB与MyISAM索引区别

（1）两者底层均使用B+Tree实现，但是前者以主键为索引来组织数据的存储，索引与数据均在同一个文件（ibd），它的每一个叶子节点即包含数据（聚集索引）；后者索引（.MYI）与数据（.MYD）分离，它的每一个叶子节点只记录数据的磁盘地址。

（2）前者遍历辅助索引时，经过每一个叶子节点后会再次进入主键索引从根节点开始查找，后者直接进入数据区。

#### 索引失效场景？

原理：离散度太差，不满足最左匹配原则都会失效。

（1）模糊查询like ‘%XXX’， 但like ‘aaa%’， 离散型太差有可能没用到索引。

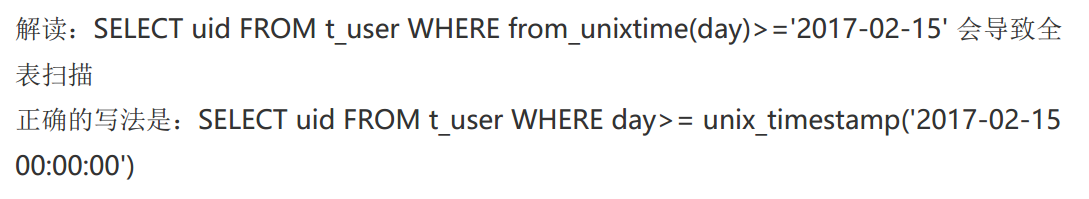
（2）not in与<>操作无法使用索引。

（3）select \*不会用到覆盖索引。

（4）联合索引的查询若不符合最左匹配原则也会失效，比如靠左列是范围查询，或者不按照最左列开始查找。（先精确匹配最左列再查找之后的列是可以的）

（5）隐试转换也不会走索引，比如你定义一个varchar类型的索引列，查的时候传一堆int型的数据过去。

（6）索引列条件用函数，而不是直接范围比较也会失效。



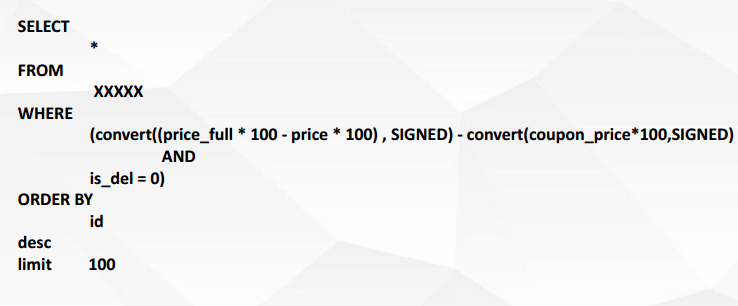
可以看看58同城sql军规。

### 一些查询优化范例

#### where 1 or 2 or 3 与 in（1,2,3）

in参数限制在1000个以内（oracle，mysql是sql长度限制），但是通过2分查找进行数值比较（O(logn)），时间复杂度比前者好。

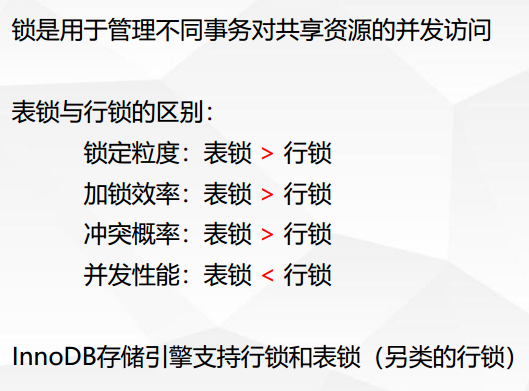
#### 无法优化的sql



问题描述：数据表大约100多W数据，sql已经无法再优化（中间减号换成不等号才是原题）

解决方案：可以将中间的计算结果在创建表时新建一列，存储计算结果并与is\_del组合一个联合索引，以空间换时间。并且设计金钱时最好别用浮点数，单位用分记，数据为整形。

### 锁



InnoDB的表锁很搞，它是给所有行加上锁，转变成了表锁。实质没有表锁。

而且锁实际上都是针对的索引。

#### 锁类型

##### 共享锁（shared lock,S锁）

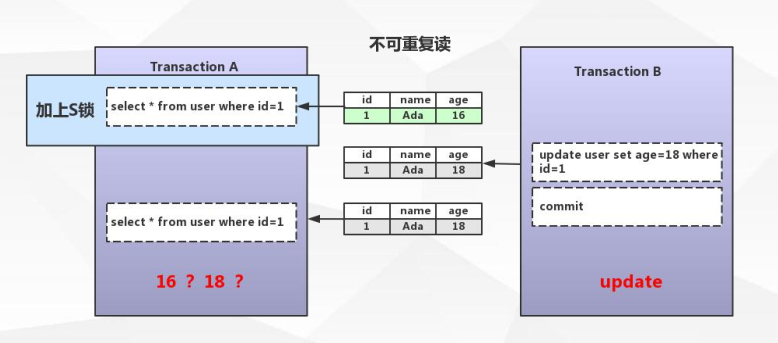
又称读锁，简称S锁，即多个事务对同一数据可以共享一把锁，都能访问到数据，但是只能读不能修改，是行锁。

用法：

select \* from users where id = 1 LOCK IN SHARE MODE;

commit/rollback。

可以通过共享锁解决不可重复读的问题，事务A读的时候锁住这一行数据，这样其他事务就不能在读期间修改该行数据。



##### 互斥锁（写锁，X锁）

不能与其他锁并存，如果一个事务获取了一个数据行的排他锁，其他事务就不能再获取该行的锁，只有该获取了排他锁的事务可以对数据行进行读取和修改，是行锁。（其他事务要读取数据可来自于快照）。

delete/update/insert默认加X锁，select需参考如下使用：

SELECT \* FROM table\_name WHERE … FOR UPDATE

commit/rollback。

可用互斥锁解决脏读问题，事务B在修改这行数据时加上X锁，事务A读取就会被阻塞，改完后才能读取最终结果数据。



##### 意向共享锁（IS）与意向排他锁（IX）

即事务准备给数据行加共享/排他锁时，必须先取得该表的IS/IX锁。这种锁之间（是可以相互兼容的。这两种锁是InnoDB数据操作之前自动加的，不需要用户干预。

作用：

当事务想要去锁表时，可以先判断意向锁是否存在，存在时可快速返回该表不能启用表锁，不需要一行一行判断。

注意，意向锁是表级别的锁，可以当做数据表的一种标志。

##### 自增锁AUTO-INC locks

针对自增列自增长的一个特殊的表级别锁。

show variables like ‘innodb\_automic\_lock\_mode’；

默认取值1，代表连续，事务未提交ID永久丢失。

自增锁是一种特殊的表级别锁（table-level lock），专门针对事务插入AUTO\_INCREMENT类型的列。最简单的情况，如果一个事务正在往表中插入记录，所有其他事务的插入必须等待，以便第一个事务插入的行，是连续的主键值。

#### 锁算法

##### 临键锁（Next-key locks）

当sql执行按照索引进行数据的检索时，查找条件为范围查找（between and、<、>等）并有数据命中则此时sql语句加上的锁为记录锁，锁住命中索引的记录+区间（左开右闭），存在于非唯一索引。

比如explain sql中type为range时，差不多就这么个情况。

范例：此时数据表主键为1、4、7、10

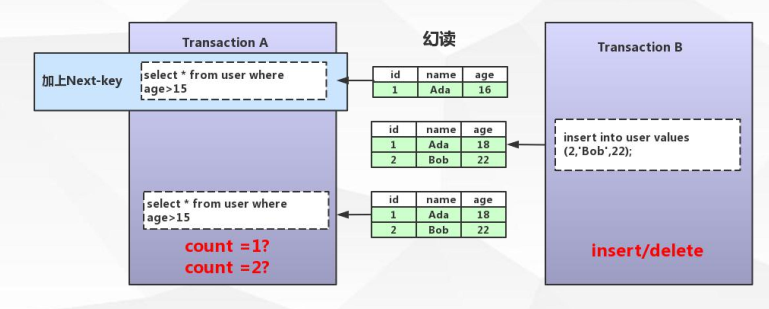


7命中，就会锁住命中值的左右区间（相邻区间，左开右闭）。

临键锁相当于间隙锁和记录锁的整合

为什么InnoDB选择临键锁作为行锁的默认算法？

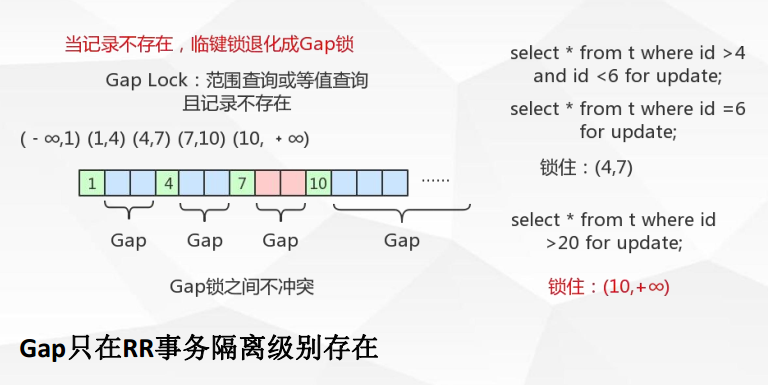
防止幻读。因为索引是B+树，从左往右依次增大。锁住的话，相邻区间就会插不进去值（行锁可是锁的索引），防止幻读。



事务A在查询过程中锁定了age大于15的区间（这里数据少应该是整个正负区间都给锁了），那么B插入大于15的数据时会阻塞，防止幻读。

##### 间隙锁（Gap）

当记录不存在时，临键锁会退化成间隙锁，存在于非唯一索引。



比如4-7查不到结果，就只锁4-7这个区间，只在可重复读这个事务隔离级别存在。

##### 记录锁（Record）

唯一性索引精确匹配时的锁，意思就是通过索引查出来多条数据就不是这么个锁法，是一个1-7的开区间（左右都开，只是结论），存在于唯一索引。



#### InnoDB的行锁特点

场景：

取消当前事务的自动提交

（1）update查询条件为非索引列（包含主键索引与辅助索引）时，其他事务update操作阻塞。

（2）update查询条件为主键时，其他事务update非该主键的数据行成功。

结论：

InnoDB的行锁是通过给索引上的索引项加锁来实现的，如果是2级索引就会2级索引与主键索引一起加锁。

只有通过索引条件进行数据检索，InnoDB才会使用行级锁，否则InnoDB将使用表锁（锁所有行记录）。

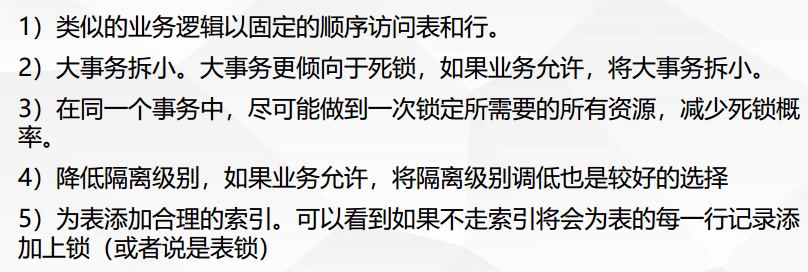
表锁：lock tables xx read/write;

#### 死锁

##### 出现条件

多个并发事务，且每个事务已持有锁（或者正在等待锁），每个事务都需要再继续持有锁，事务之间产生加锁的循环等待，形成死锁。

##### 死锁的避免



##### 如何发现与排查

### MVCC

#### 概念

Multiversion concurrency control多版本并发控制。

即并发访问数据库时，对正在事务内处理的数据做多版本的管理。以达到用来避免写操作的堵塞，从而引发读操作的并发问题。（就是说写互斥时不要读阻塞）

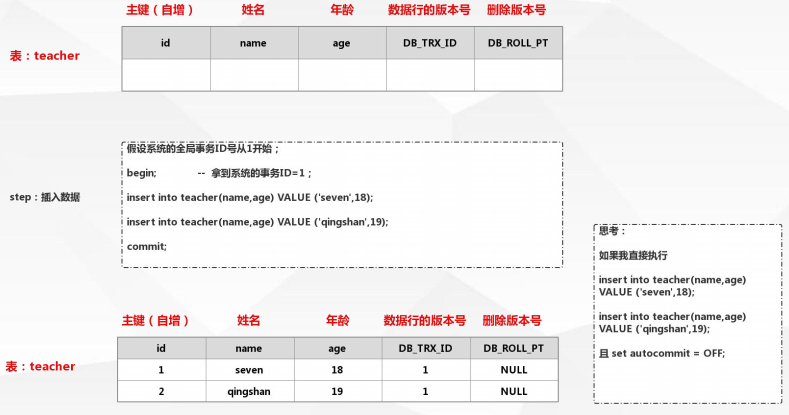
#### 实现原理

即每一张表都有都有隐藏的两列，一类是数据行的版本号DB\_TRX\_ID（入库时版本号），另一列是数据行的删除版本号DB\_ROLL\_PT（删除时版本号）。

开启事务：

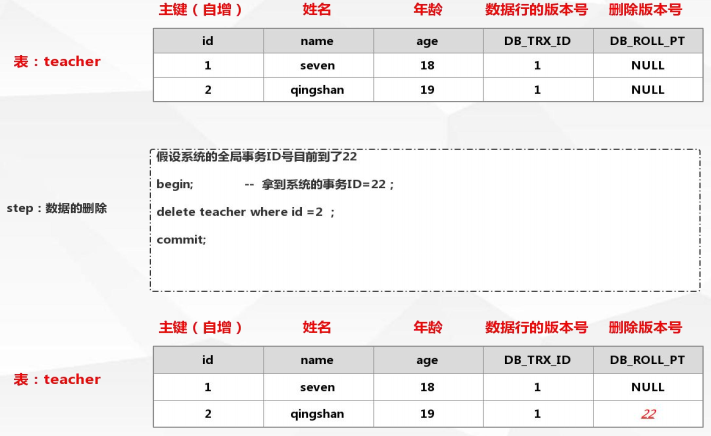
（1）插入

数据行版本号插入当前事务ID，删除版本号为null。



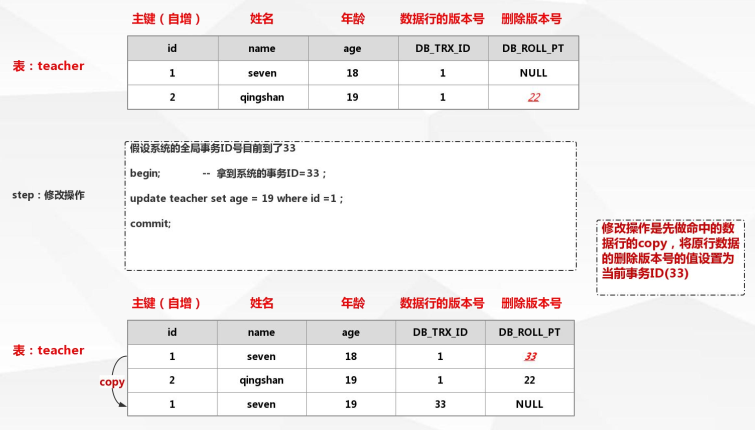
（2）删除

将当前事务版本号插入删除版本号列。



（3）修改

修改时，将命中的数据行进行复制（出现两个数据相同的行，但版本号不相同）；新增数据行版本号插入当前事务ID，删除版本号为null；旧数据行删除版本号改为当前事务ID。

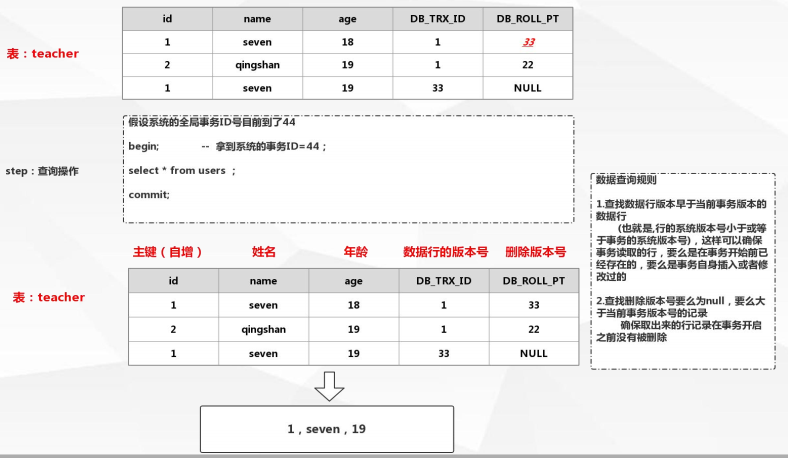


（4）查询

查询时，命中的记录有两个条件：

（1）数据行版本早于（包含）当前事务ID

（2）且删除版本要么为null，要么大于当前事务版本



第一个条件意思就是执行当前查询事务时，数据行要么已经存在，要么是当前事务插入的数据；第二个条件是确保事务开启时该数据还没被删除（null表示没人删它，大于当前事务ID表示至少该事务处理时数据行还存在）。

MVCC手段只适用于隔离级别中的读已提交和可重复读，不适用脏读。MVCC的创建版本和删除版本只要在事务提交后才会产生。总之MVCC主要作用于事务性的，有行锁控制的数据库模型。

在Read Committed下，MVCC读取的是被锁定数据的最新的一份数据。  
在Repeatable Read下，MVCC读取的是事务刚开始时候的数据。

参考文章：

https://www.jianshu.com/p/f628223fa1fc

#### Undo log

纯靠mvcc无法解决出现的脏读问题，undo log是为了实现事务原子性而出现的产物，是mvcc的一种实现。

undo log指事务开始之前，在操作任何数据之前，首先将需要操作的数据备份到一个地方（Undo log）。

Undo log实现事务原子性：事务处理过程中如果出现了错误或者用户执行rollback，Mysql可以利用undo log中的备份将数据恢复到事务开始之前的状态。

Undo log在mysql innodb中用来实现多版本并发控制。

实现流程：事务未提交之前，undo保存了未提交之前的版本数据，undo中的数据可作为数据旧版本快照供其他并发事务进行快照读。

#### 快照读与当前读

快照读：

sql读取的数据时快照版本，也就是历史版本，普通的select就是快照读。

innodb快照读，数据的读取将由cache（原始版本）+undo（事务修改过的数据）两部分组成。

当前读：

sql读取的数据时最新版本，通过锁机制来保证读取的数据无法通过其他事务进行修改。

update、delete、 insert、 select … lock in share mode、 select … for update都是当前读。

#### Redo log

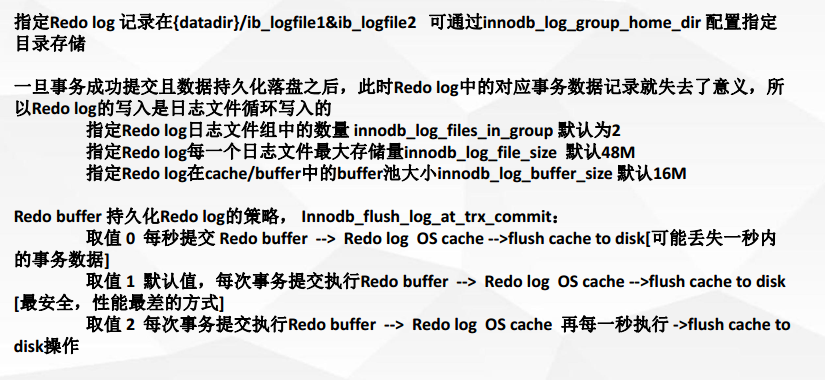
指事务操作的任何数据，将最新的数据备份到一个地方（redo log）。

redo log的持久：

不是随着事务的提交才写入的，而是在事务的执行过程中，便开始写入redo中，具体的落盘策略可以进行配置。

Redo log实现事务持久性：

防止在发生故障的时间点，尚有脏页未写入磁盘，在重启mysql服务的时候，根据redo log进行重做，从而达到事务的未入磁盘数据进行持久化这一特性。是为了实现事务的持久性而出现的产物。Mysql只要事务已经写到了redo日志，就会认为事务已经提交，而不管其是否已经持久化到ibd磁盘文件中。



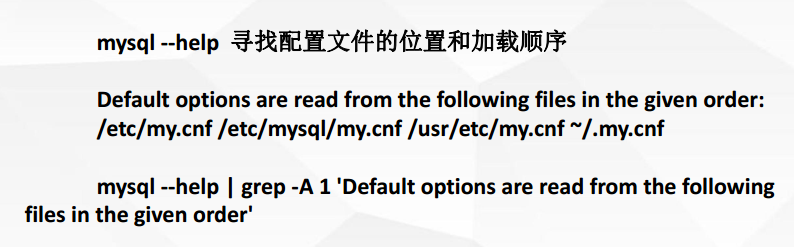
建议取值2.

小结：binlog用于备份恢复，undo log用于保证原子性和并发问题，redo log用于已提交事务的一致性问题

### 配置优化

#### Mysql参数





#### 常用配置

##### max\_connections

最大连接数。5.6以前是100个，5.7是151个。最大数与系统句柄数有关。

linux系统句柄数：

使用ulimit -a查看linux open files配置，默认是65536.在/etc/security/limits.conf下。

mysql句柄数配置：

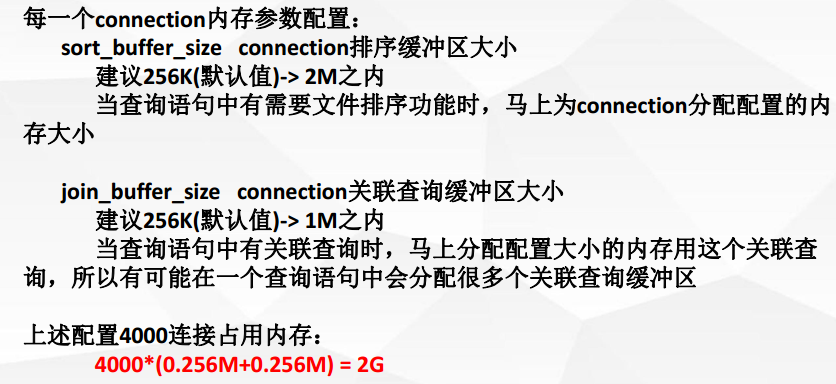
/usr/lib/systemd/system/mysqld.service

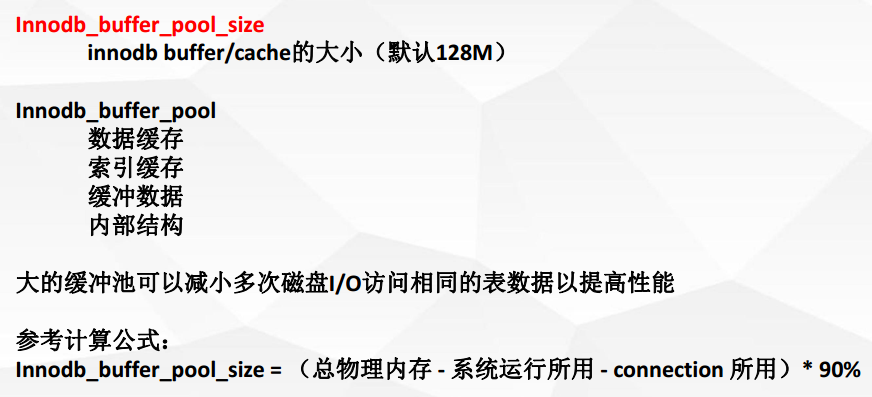
LimitNOFILE = 5000（默认）

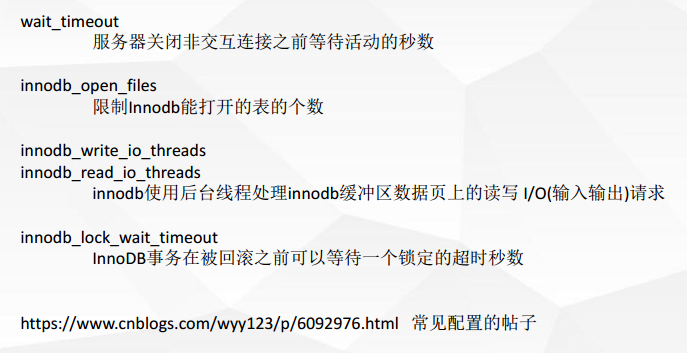
##### 其他常用配置



##### 内存配置



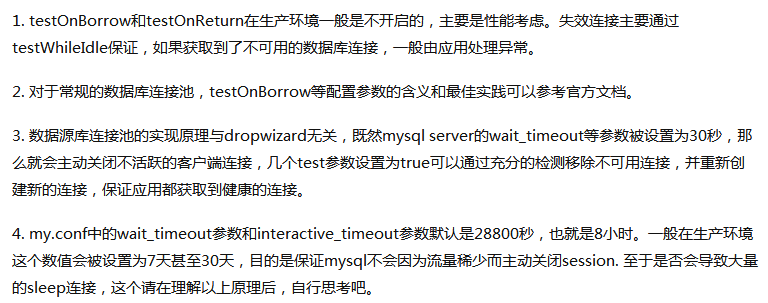




<https://www.cnblogs.com/wyy123/p/6092976.html>.

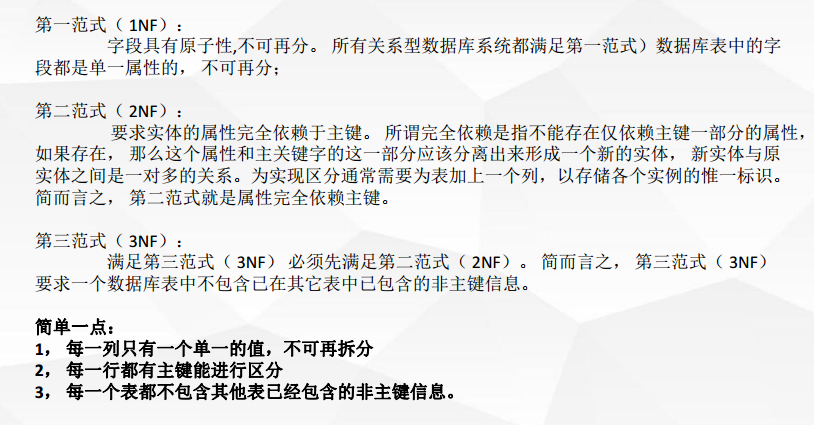
还有mysql默认配置的8小时连接失效问题：

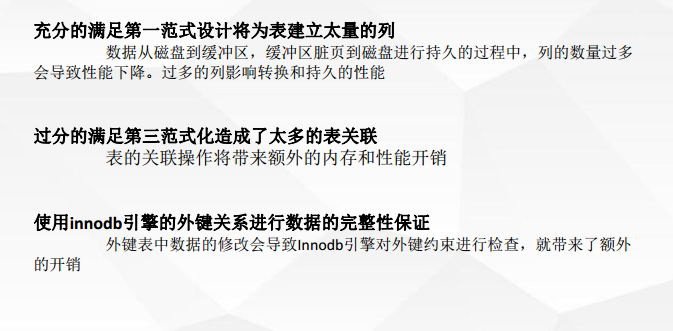
https://blog.csdn.net/zhanxiaoyun1/article/details/79654668



或者尝试在数据库连接的url上加上autoReconnect=true。

### 数据库范式





### 悲观锁和乐观锁

### 连接池

### SQL基础

#### 内连接与外连接

##### 内连接

INNER JOIN指的是把表连接时表与表之间匹配的数据行查询出来，就是两张表之间数据行匹配时，要同时满足ON语句后面的条件才行。

##### 外连接

LEFT (OUTER) JOIN无论是否符合ON后面的表连接条件都会把另外一张表的记录全部描述出来。

##### 左外连接与右外连接



比如，左外是左表全部展示+右表匹配符合条件的行，右外是右表全部展示+左表匹配符合条件的行，缺少数据全部填为NULL。

##### 全外连接

left  join  + union (去除重复数据) + right  join (full outer join貌似不能用？)完整外部联接返回左表和右表中的所有行。当某行在另一个表中没有匹配行时，则另一个表的选择列表列包含空值。

其实就是笛卡尔积，左外连就是左表与右表差集，右外连就是右表减左表差集，全外就是就是取并集。

#### 联合查询

##### union与union all

Union：对两个结果集进行并集操作，不包括重复行，同时进行默认规则的排序。

Union All：对两个结果集进行并集操作，包括重复行，不进行排序。

因为union要进行重复值扫描，所以效率低。

#### 外键约束

最好不用，最好从代码层面控制表（数据）之间的关联（比如有个外键id，但并不建立外键约束）。主要原因大致如下：

（1）在大数据量的情况下对性能有影响。

（2）解耦，否则表结构很容易混乱，一旦有改动也很麻烦，业务层面的数据关联应该在业务层实现。

（3）大数据量时表水平切分，外键约束、触发器、存储过程都是禁区。

#### DISTINCT

##### 单列操作

对一列操作，表示选取该列不重复的数据项

##### 多列操作

表示选取 多列都不重复的数据，相当于 多列拼接的记录 的整个一条记录 , 不重复的记录。

参考博客

<https://blog.csdn.net/u010003835/article/details/79154457>

#### exists 与 in

select \* from t\_a where id IN(select a\_id from t\_b);

select \* from t\_a where exists(select a\_id from t\_b where t\_a.id = t\_b.a\_id);

in是先执行子查询得到结果集，再用子查询的结果去匹配外表（t\_a）。

exists是执行外表的遍历，然后相关的子查询会利用外表的数据对内部表进行匹配。

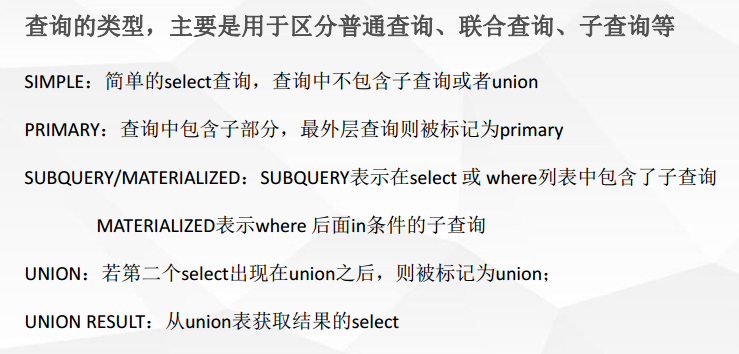
假设t\_b子查询结果为M,t\_a表大小为N

那么in的时间复杂度为O（M\*logN）,exists为O（N\*logM）。

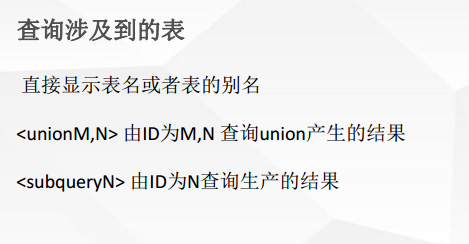
所以根据小数据集驱动大数据集的思想，外表大，子查询结果小，使用in，外表小，内表数据大，使用exists。

#### explain sql的执行计划参数

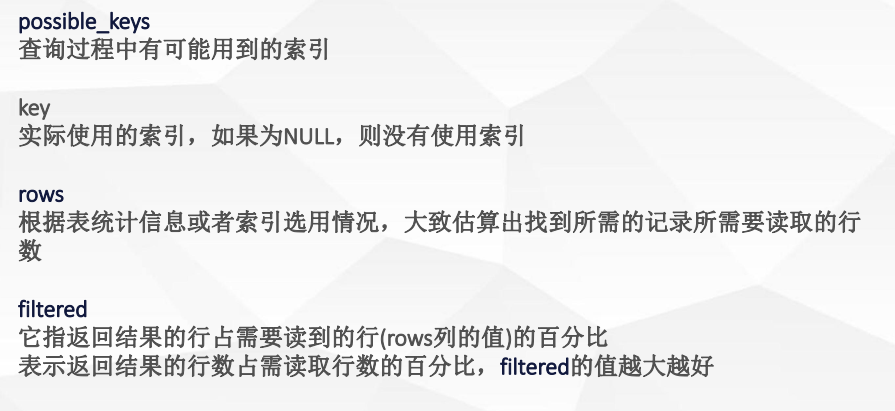
##### select\_type



##### table



##### 其他



##### type（重要，性能指标）



const一般出现于where条件为常量的精确匹配情况，并且走唯一索引。

eq\_ref与ref一般出现于连接查询or子查询中。

别概念搞混了哈，唯一索引一般值主键或者唯一约束的列的索引，非唯一索引就是这索引列可能有相同的多个值。所以eq\_ref一般出现于主键or唯一键索引，ref出现于辅助索引。

range走索引一般出现于范围查询，比如IN，like（遵守最左匹配）。大于小于的情况我还没碰到过。

index走索引全扫描。

all大家都懂，表全扫，不用索引。

##### extra（重要，额外信息）



#### 临时表、视图

#### druid监控

### 其他小事

（1）8.0版本移除了.frm描述文件。

## JVM

### 强、软、弱、虚引用

#### 强引用（Strong Reference）

JVM宁可抛出OOM异常，对象也不会回收的引用，比如Object o = new Object();之类，要么强制赋值为null，要么进程结束（线程）才会销毁的引用。

#### 软引用（Soft Reference）

只有当即将发生OOM异常的时候，对象才会被回收的引用。

Java提供了一个SoftReference借口用以标志软引用对象。

Object softRefObject = **new** Object();  
*// 传入软引用对象*SoftReference<Object> softReference = **new** SoftReference<>(softRefObject);  
softRefObject = **null**;  
System.*gc*();  
*// 获取软引用对象，结果不为空*System.***out***.println(softReference.get());

#### 弱引用（Weak Reference）

只要执行GC，对象就一定会被回收掉的引用。

Object weakRefObject = **new** Object();  
*// 传入虚引用对象*WeakReference<Object> weakReference = **new** WeakReference<>(weakRefObject);  
weakRefObject = **null**;  
System.*gc*();  
*// 获取虚引用对象，结果为空*System.***out***.println(weakReference.get());

不过要注意一点，如果是string类型的弱引用对象，是不会被回收的，因为string类型对象是存放在常量池的，gc不会去回收常量池。

#### 虚引用（Phantom Reference）

虚引用与前面的软、弱引用不同，不影响对象生命周期，如果一个对象与虚引用相关，则跟没有引用关联一样，在任何时候都可能被回收。

虚引用使用必须与引用队列关联使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会把这个虚引用加入到与之 关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

ReferenceQueue<Object> queue = **new** ReferenceQueue<>();  
Object phantomRefObject = **new** Object();  
PhantomReference<Object> phantomReference = **new** PhantomReference<>(phantomRefObject, queue);  
*// 永远为null*System.***out***.println(phantomReference.get());  
  
phantomRefObject = **null**;  
System.*gc*();  
*// null*System.***out***.println(phantomReference.get());  
ThreadUtils.*sleep*(200);  
*// java.lang.ref.PhantomReference对象存在*System.***out***.println(queue.poll());

#### 软、弱引用使用

这两种引用最常用的使用方法是解决OOM问题，比如内存缓存的对象，可以使用SoftReference将其封装，当内存不足时即回收，可以有效避免OOM问题。

### ClassLoader

类加载器：启动类加载器，扩展类加载器，应用类加载器，自定义加载器。

加载流程：

加载->验证->准备->解析->初始化。

加载：将class文件加载进jvm。

验证：验证class文件格式是否规范（元数据，字节码，符号链接）。

准备：为变量分配内存并设置类变量(static)的默认值（只初始化默认值，而不是赋指定值）。

解析：把常量池符号引用替换为直接引用。包括各种类、方法、接口、方法等解析。

初始化：根据定制的初始化值初始化各种变量（比如局部、静态块等）。

### 如何定位CPU中使用较高的进程（线程）？如何发现死锁？

https://www.jianshu.com/p/3667157d63bb

### Java虚拟机生命周期

当启动一个Java程序时，一个虚拟机实例也就诞生了。当程序关闭退出，这个虚拟机实例也就随之消亡。

 Java程序初始类中的main()方法，将作为该程序初始线程的起点，任何其他的线程都是由这个初始线程启动的。   
  Java虚拟机内部有两种线程：守护线程和非守护线程。守护线程通常是由虚拟机自己使用的，比如执行垃圾收集任务的线程。 但是，Java程序也可以把它创建的任何线程标记为守护线程。而Java程序中的初始线程-就是开始于main()的那个是非守护线程。   
   只要还有任何非守护线程在运行，那么这个Java程序也在继续运行(虚拟机仍然存活)。当程序中的所有的非守护线程都终止时，虚拟机实例自动退出。**假如安全管理器允许，程序本身也能够通过调用Runtime类或者System类的exit()方法退出。**

## Spring

### Spring中的bean线程安全吗？

spring bean是通过扫描，反射后并缓存在IOC容器中的实例，spring并没有Bean做任何特殊处理，与线程安全无关。bean线程安全取决于Bean本身方法是否线程安全。

### Spring的bean是如何回收的？

与Bean生命周期有关，singleton是随着Spring的存亡而存亡（Map中的强引用），prototype是使用时创建一个，用完了可能就被回收了，session没了会回收，request是一次请求就结束了。

### Spring的初始化，实例化，销毁？

初始化：事实上, 定义 BeanDefinition （从xml或者注解扫描后的metadata中读取并加载Java类的原始数据到内存，通常是class二进制文件内容）

并注册到 BeanFactory（保存到DefaultListableBeanFactory的属性中） 的这个过程, 就是 bean 的定义和注册（也就是初始化） , 我们通常使用 spring 时并不是把一个 bean 的实例注册, 而是一个 BeanDefinition.

实例化：使用反射调用构造函数实例化class，在内存中生成对象，接着给这个对象设置属性，自动装配依赖。

销毁：

https://www.jianshu.com/p/4b9d7455c8b2

### IOC、DI与AOP

IOC：Inversion of control，控制反转，意思就是本来需要我们创建、依赖的实例代码，交给spring容器来帮我们实现。

DI：Dependency of Injection，依赖注入，指对象依赖的实例，不需要自己主动从容器里找，而是被动地由容器在实例化对象的时候主动注入给它，可以看做是IOC实现的一部分。

AOP：Aspect Oriented Programming，面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

### IOC初始化

无论是applicationContext，还是springmvc的dispatcherServlet，最终殊途同归，调用AbastractApplicationContext的refresh方法开始IOC容器的初始化：

1.定位：定位配置文件和扫描相关的注解（不管是filesystem，classpath，还是网络资源等）。

2.加载：将bean元信息（BeanDefinition）载入到内存中。

3.注册：根据载入的信息，将beanDefinition初始化到Bean信息容器中（DefaultListableBeanFactory下registerBeanDefinition方法，put到beanDefinitionMap中，一个线程安全map）。

##### 总结

Spring中AbstractApplicationContext的refresh方法完成了spring ioc容器的初始化，refresh主要包含定位（getResource）、加载（reader.loadBeanDefinition()）、注册（registerBeanDefiniion()）。主要完成了bean元信息的注册。

### 依赖注入

#### 时间

1.用户第一次调用getBean方法时

2.用户在配置中将bean配置了延迟加载=false的属性，即让容器在解析注册bean时就触发注入。

#### 流程

寻找入口->开始实例化->选择实例化策略->执行实例化->准备依赖注入->解析注入规则->注入赋值。

#### bean的存储位置

单例bean的缓存位置：singletonObjects，是一个concurrenthashmap。位于DefaultSingletonBeanRegistry。

非单例从BeanWrapper里获取。

如果获取的Bean是FactoryBean的话，就会进行再次转换，从FactoryBeanRegistrySupport的factoryBeanObjectCache获取FactoryBean实例，调用其getBean方法获得具体的bean对象。

总结：IOC实例化从abstractBeanFactory中getBean开始，延迟加载生成BeanWrapper（包含代理对象等各种信息），存储于AbstractAutowiredCapableBeanFactory的FactoryBeanInstanceCache线程安全中，如果是单例就从DefaultSingletonBeanRegistry的singletonObjects尝试获取缓存。

AbstractAutowiredCapableBeanFactory创建实例是createBeanInstance，注入的开始方法是populateBean，直至BeanWrapperImpl的setValue方法（反射创建实例）。

#### 循环依赖如何解决

##### 单例

构造器循环依赖：不能解决，报BeanCurrentlyInCreationException.

setter循环依赖（或者说field属性）：可以解决，通过暴露一个单例工厂方法，从而使其他bean能引用到该bean，实现注入。

Spring会将每一个正在创建的bean放入一个当前创建”bean”池中，bean标识符在创建过程中将一直保持在这个池中，因此如果在创建bean过程中发现自己已经在bean池里的bean，会报出上述异常表示循环依赖，创建完成的bean才会从当前创建bean池里清除掉。

构造器循环依赖不能解决的原因正是如此，构造器创建过程中，该bean一定会在当前创建bean池里，而经过循环调用构造器创建依赖bean到再次创建自己的bean时，就会发现自己已经在bean池里了，于是无法解决，而setter依赖可以就可以通过暴露单例工厂方法的形式获取bean。

好吧以上描述太抽象，直接说spring单例bean工厂实现DefaultSingletonBeanRegistry的四个存储器。

Map<String, Object> singletonObjects singletonObjects

用来存放注册的SingletonBean。

Map<String, ObjectFactory<?>> singletonFactories

存储制造 singleton 的工厂。

Map<String, Object> earlySingletonObjects

是singletonFactory 制造出来的 singleton 的缓存。

Set<String> registeredSingletons

按顺序存放已经注册的SingletonBean的名称。

其中singletonObjects被称为1级缓存，earlySingletonObjects为2级缓存，singletonFactories为3级缓存（之后的对象描述就以X级缓存为准）。

1级缓存就是spring成品单例（实例化，属性注入均完成）的存放地，平时的service啊，controller之类的基本都从这里直接拿，2级缓存为提早曝光的bean对象缓存，这个提早曝光的bean意思就是只是构造器实例化，但是没有完成属性注入的bean，也正是3级缓存生产出来的bean，3级缓存是为了解决循环依赖而产生的缓存，spring在创建单例bean时会为该bean注册一个ObjectFactory用以获取未完成依赖注入的不完整的bean，并缓存至3级缓存。

考虑以下场景，A引用B，B引用A，首先A实例化后会注册一个能获取到提前曝光的A的ObjectFactory至缓存3，然后注入B，此时执行B的实例化流程，根据spring AbstractBeanFactory.getSingleton()方法的实现，B依赖注入A时，首先去1级缓存拿A，这时肯定没有A，然后去2级缓存拿，也没有A，再去3级缓存拿，此时能够通过缓存3的A的ObjectFactory获取提前曝光的A，将A存入缓存2中，然后删掉缓存3的ObjectFactory。此时B就能够完成依赖注入，而A也提前曝光到了缓存2。

其中在能够获取缓存3的过程中需要判断allowEarlyReference这个bool值，只要allowEarlyReference这个属性为true，意味着就是允许循环依赖，能够通过访问第3级缓存解决单例的循环依赖问题。

##### 多例

不能解决，报BeanCurrentlyInCreationException。

#### Spring注入的多种方式

##### 配置

1.setter注入

2.构造器注入

3.接口注入（几乎没咋用）

##### 注解

1.字段注入

2.构造器注入

3.setter注入

### BeanFactory和FactoryBean的区别

FactoryBean顾名思义，是一个工厂bean，通过其getBean(id)获得的是该工厂所产生的bean的实例，而不是该factorybean的实例。是spring是内部实现的一种规范。factoryBean使用&开头作为beanName。spring中所有容器都是factoryBean。**这个Bean不是简单的Bean，而是一个能生产或者修饰对象生成的工厂Bean,它的实现与设计模式中的工厂模式和修饰器模式类似**

BeanFactory 是bean工厂的顶层功能规范，IOC容器的核心接口，定义了用户向容器索取bean的getBean()方法，**所有的Bean都是由BeanFactory(也就是IOC容器)来进行管理的，如果需要获取FactoryBean，需要用&转义**。

### BeanFactory和ApplicationContext区别

ApplicationContext是BeanFactory的子接口， BeanFactory是基础，BeanFactory和它的子接口定义的API提供了spring环境中对bean管理和配置；

2. ApplicationContext是扩展，提供了对bean的监控，生命周期管理（各种aware,processor），事件，国际化，统一资源文件读取url等一系列拓展功能，beanfactory有的功能，ApplicationContext都有；

### Spring的bean是否线程安全？

与spring无关，只与自己写的业务代码有关。

spring中的bean在IOC初始化被注册成beanDefinition，getBean实例化包装到BeanWrapper后放到Map<String,BeanWrapper>的缓存中（单例存Map<String,Object>），整个过程是spring自己通过加锁保证的线程安全。Bean中的方法是否线程安全，全是使用者自己实现决定。

### Spring中用到了哪些设计模式？

太多了，责任链（MVC拦截器），动态代理（AOP），装饰器（BeanWrapper），注册式单例，适配器（MVC），模板方法，委派（非23种设计模式），策略，工厂，原型等

### Bean的生命周期

1.实例化bean对象（反射，此处省略InstantitionAwareBeanPostProcessor的前后方法）。

2.设置对象属性（依赖注入）。

3.检查bean是否实现相关aware接口（BeanFactoryAware，BeanNameAware），实现了则调用对应setter方法注入对应实例。

4.BeanPostProcessor前置处理postProcessBeforeInitialization。

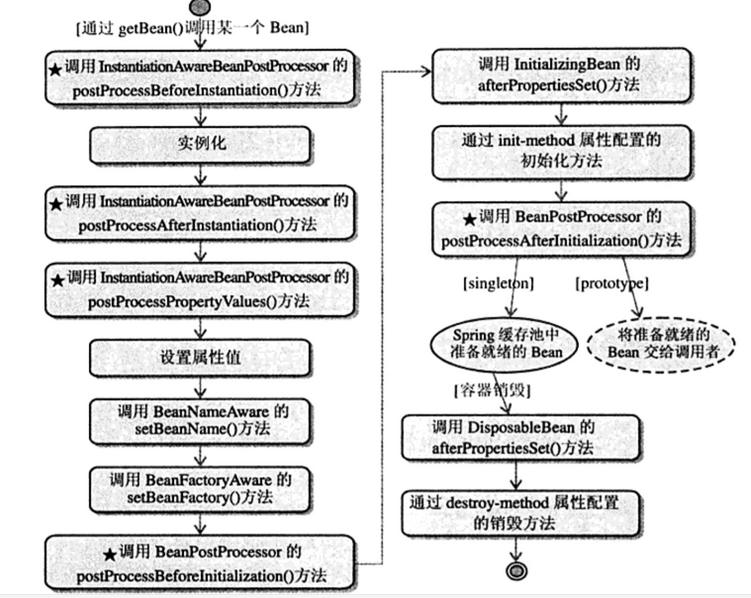
5.检查是否是InitializingBean以决定是否调用afterPropertiesSet()方法，检查是否配置有自定义init-method。可总结为调用bean的初始化方法。

6. BeanPostProcessor后置处理postProcessAfterInitialization。

7.注册必要的Destruction相关回调接口（略），单例Bean加入map缓存，多例直接交给使用者。

8.Bean创建完毕，使用

9.销毁，是否实现DisposableBean接口，或者配置有自定义的destroy方法。可总结为调用bean的销毁方法。



DisposableBean的方法好像有错误。

其实说简单点就是createBeanInstance实例化，populateBean注入属性，init初始化，使用，然后销毁。

### Spring AOP

#### AOP

AOP技术恰恰相反，它利用一种称为"横切"的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为"Aspect"，即切面。所谓"切面"，简单说就是那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块之间的耦合度，并有利于未来的可操作性和可维护性。

#### AOP概念

切面（aspect）：切面是通知和切点的结合。通知和切点定义了切面的全部内容——它是什么，在何时何处完成其功能。是面向规则，具有相同规则的方法的集合体。

切入点（pointcut）：需要代理的具体方法。

连接点（joinpoint）：被拦截到的点，因为Spring只支持方法类型的连接点，所以在Spring中连接点指的就是被拦截到的方法，实际上连接点还可以是字段或者构造器。

通知（advice）：拦截到连接点后需要执行的方法，回调。分为前置、后置、异常、最终、环绕通知五类。

目标对象（target）：代理的具体对象。

AOP代理（AOP Proxy）：主要有JDK动态代理和CGLIB。

前置通知（Before Advice）：在切点（方法）前调用织入的方法。

后置通知（After Advice）：在切点（方法）调用后调用织入的方法。

环绕通知（Around Advice）：只要触发切点调用就会调用织入的方法。

返回后通知（After Return Advice）：返回值为非Void，织入的方法。

异常通知（After Throwing Advice）：抛出异常调用织入的方法。

<https://www.cnblogs.com/lidj/p/7194193.html> AOP执行顺序，通过order控制。

#### 应用场景

Spring本身AOP应用最多的就是事务管理。before开启事务，after关闭事务。

常用场景：事务、日志、安全、异常处理。

#### 底层实现概要

1.使用简单工厂模式决定是jdk代理还是cglib

2.使用责任链模式决定advice的执行与执行顺序（order配置排序）。

流程：

寻找入口->选择策略->调用方法->触发通知。

实际上就是利用BeanPostProcessor的后置处理方法，调用DefaultAopProxyFactory的createAopFactory方法，该工厂模式只提供Jdk和cglib的代理。如果被代理对象有实现接口则默认JDK，没有则CGLIB。

### Spring事务

#### 事务的传播行为

**传播行为：当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。**

　　Spring 定义了如下七中传播行为，这里以A业务和B业务之间如何传播事务为例说明：

　　①、PROPAGATION\_REQUIRED ：required , 必须。默认值，A如果有事务，B将使用该事务；如果A没有事务，B将创建一个新的事务。

　　②、PROPAGATION\_SUPPORTS：supports ，支持。A如果有事务，B将使用该事务；如果A没有事务，B将以非事务执行。

　　③、PROPAGATION\_MANDATORY：mandatory ，强制。A如果有事务，B将使用该事务；如果A没有事务，B将抛异常。

④、PROPAGATION\_REQUIRES\_NEW ：requires\_new，必须新的。如果A有事务，将A的事务挂起，B创建一个新的事务；如果A没有事务，B创建一个新的事务。

意思就是两个事务相互独立，外层事务失败回滚，不能回滚内层事务，内层事务失败抛出异常，外层事务捕获后也可以不回滚。

　　⑤、PROPAGATION\_NOT\_SUPPORTED ：not\_supported ,不支持。如果A有事务，将A的事务挂起，B将以非事务执行；如果A没有事务，B将以非事务执行。

　　⑥、PROPAGATION\_NEVER ：never，从不。如果A有事务，B将抛异常；如果A没有事务，B将以非事务执行。

⑦、PROPAGATION\_NESTED ：nested ，嵌套。A和B底层采用保存点机制，形成嵌套事务。

#### Spring事务的隔离级别

①、ISOLATION\_DEFAULT：使用后端数据库默认的隔离级别

　　②、ISOLATION\_READ\_UNCOMMITTED：最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读

　　③、ISOLATION\_READ\_COMMITTED：允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生

　　④、ISOLATION\_REPEATABLE\_READ：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生

⑤、ISOLATION\_SERIALIZABLE：最高的隔离级别，完全服从ACID的隔离级别，确保阻止脏读、不可重复读以及幻读，也是最慢的事务隔离级别，因为它通常是通过完全锁定事务相关的数据库表来实现的

也就是在四个公共的隔离级别上加了数据库默认的隔离级别。

#### ****只读****

 　　这是事务的第三个特性，是否为只读事务。如果事务只对后端的数据库进行该操作，数据库可以利用事务的只读特性来进行一些特定的优化。通过将事务设置为只读，你就可以给数据库一个机会，让它应用它认为合适的优化措施。

#### ****事务超时****

为了使应用程序很好地运行，事务不能运行太长的时间。因为事务可能涉及对后端数据库的锁定，所以长时间的事务会不必要的占用数据库资源。事务超时就是事务的一个定时器，在特定时间内事务如果没有执行完毕，那么就会自动回滚，而不是一直等待其结束。

#### 编程式事务与声明式事务

**编程式事务处理**：所谓编程式事务指的是通过编码方式实现事务，允许用户在代码中精确定义事务的边界。即类似于JDBC编程实现事务管理。管理使用TransactionTemplate或者直接使用底层的PlatformTransactionManager。对于编程式事务管理，spring推荐使用TransactionTemplate。

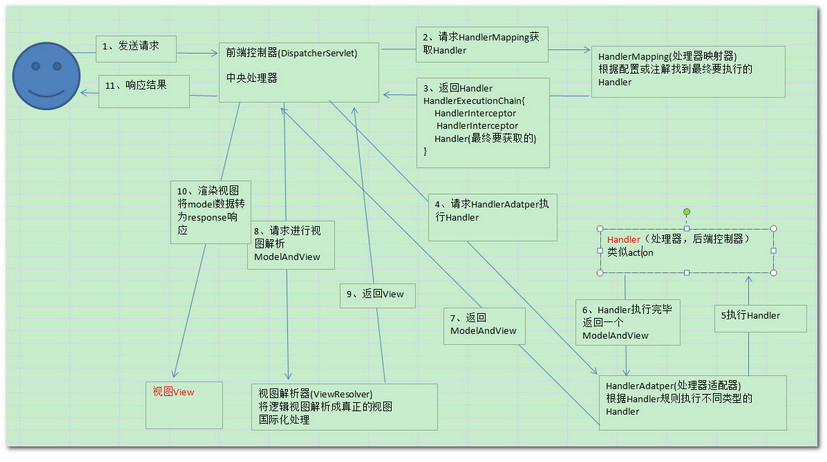
**声明式事务处理**：管理建立在AOP之上的。其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。声明式事务最大的优点就是不需要通过编程的方式管理事务，这样就不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明(或通过基于@Transactional注解的方式)，便可以将事务规则应用到业务逻辑中。

　　简单地说，编程式事务侵入到了业务代码里面，但是提供了更加详细的事务管理；而声明式事务由于基于AOP，所以既能起到事务管理的作用，又可以不影响业务代码的具体实现。

参考博客：https://www.cnblogs.com/ysocean/p/7617620.html

## SpringMVC

### SpringMVC流程



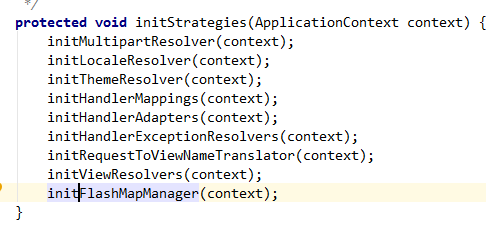
整理步骤如下，序号与图无关：

（1）核心控制器DispatcherServlet接收用户请求，通过HandlerMapping将url映射的handler返回给核心控制器。

（2）通过DispatcherServlet调用HandlerAdapter，HandlerAdapter经过适配调用具体的处理器Controller，并将结果ModelAndView返回给适配器，再由适配器返回给DispatcherServlet

（3）DispatcherServlet调用视图解析器对ModelAndView进行渲染，然后将结果返回给前端。

### Spring MVC九大组件



HandlerMappings：映射器，负责请求url与handler之间的映射关系。

HandlerAdapter：适配器，主要负责请求参数与handler方法参数的实际解析与转换。

HandlerExceptionResolvers：异常解析器，主要处理handler抛出异常后的展示。

ViewResolvers：视图解析器，负责视图的渲染。

RequestToViewNameTranslator：

LocaleResolver：

ThemeResolver：

MultipartResolver：

FlashMapManager：

### Spring MVC阶段

初始化

根据配置的DispatcherServlet，从其父类HttpServletBean中调用init()初始化，最终会调用到ApplicationContext的refresh()方法初始化IOC容器。然后又调用onfresh方法初始化SpringMVC九大组件。其中Url与Controller的关系在HandlerMapping中注册。

运行

### SpringMVC优化建议

1.Controller如果能保持单例则尽量使用单例。

[2.@RequestParam](mailto:2.@RequestParam)用于参数匹配，最好使用，如果不用springmvc会用asm自己读取参数名称，浪费性能。但是j8貌似可以getParammeterName了。

3.Springmvc并没有对url和method的对应关系进行缓存。是为了热加载。建议自己对url和method关系进行缓存。（？咋做，可以继承DispatcherServlet自己玩一个）

## Mybatis

### Mybatis的一级、二级缓存

#### 一级缓存

##### 作用域

作用于一个sqlSession中。

##### 配置

默认开启，非select操作均会更新缓存，虽然说在mapper文件中可以对select操作的useCache，flushCache做设置可以关掉一级缓存，但是好像无效？目前有效的解决办法是在每次执行后调用sqlSession.clearCache()清空缓存，在与spring整合后flushCache貌似就有效。

最简单的方式是在setting中配置<settring name="localCacheScope" value="STATEMENT">。默认是Session，改为statement后源码中每次query会清掉缓存，变相去掉了一级缓存。

##### 底层实现

查看SqlSession的默认实现DefaultSqlSession，可以看到包含一个executor的属性，而其实现BaseExecutor中可以发现名字为PerpetualCache localCache的属性，底层为一个简单的Map，此即为一级缓存存储底层实现。

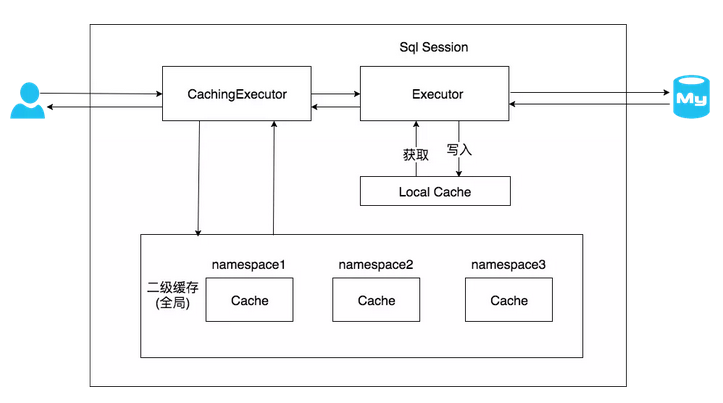
##### 隐患

一级缓存Sql有个严重的问题就是同一句查询sql，如果数据表数据被修改，再次查缓存不经过数据表，则会引起脏读。在多服务集群中该问题特别明显。可使用namespace级别的二级缓存解决。

#### 二级缓存

##### 作用域

对于一个Mapper（namespace），一级缓存是只能作用于一个sqlsession，而二级缓存是作用于所有的sqlsession，即namespace级别全局缓存。如下图所示



开启二级缓存后，数据的查询流程为：二级缓存->一级缓存->数据库。

##### 配置

（1）setting配置

<setting name="cacheEnabled" value="true"/>

（2）mapper.xml配置cache或者cache-ref

<cache/>

<cache-ref namespace="mapper.StudentMapper"/>

cache-ref后为另一个mapper的命名空间。

##### 底层实现

一旦开启二级缓存，mybatis会使用CachingExcutor装饰BaseExecutor子类，实现了缓存的查询和写入功能。

##### 隐患

二级缓存在是二级缓存在多表联查的情况下，如果不使用cache-ref配置多表的缓存关联，当非当前namespace映射的表数据被修改后，还是可能会读到脏数据。

#### 总结

Mybatis的缓存最好不要使用，即使使用也需要一个统一的Cache接口实现，在项目中直接利用缓存组件（redis,memcached）即可。

## Springboot

### springboot如何通过配置文件自动装配的？

Springboot为每一个应用，基本都提供了XXXAutoConfiguration类，该类内涵指定的应用前缀变量，只要遵守springboot应用和前缀变量名在application.properties或者yaml（包含bootstrap）中添加配置参数，即可读取到响应配置内容对应用进行配置。

### @SpringBootApplication

包含了@SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan，开启包扫描，自动配置和配置的功能。

## SpringCloud

微服务里面的问题

增加机器可以增加可用性，增加调用层次（一层调一层）会降低可靠性。

### eureka

#### 概念

服务注册与发现组件。分为eureka server服务注册中心与eureka client客户端。

角色：

register service:服务注册中心，eureka server。

provider service：服务提供者，eureka client

consumer service：服务消费者，eureka client。

流程

首先前要一个服务注册中心 Eureka Server，服务提供者 Eureka Client 向服务注册中心 Eureka Server 注册，将自己的信息（比如服务名和服务的 IP 地址等）  
通过rest api的形式提交给服务注册中心 Eureka Server。同样，服务消费者 Eureka Client 也向服务注册中心 Eureka Server 注册，同时服务消费者获取一份服务注册列表的信息 ， 该列表包含了所有向脱务注册中心 Eureka Server 注册的服务信息。获取服务注册列表信息之后 ，服务消费者就知道服务提供者的 IP 地址，可以通过 Http远程调度来消费服务提供者的服务。

##### Register服务注册

当eureka client向eureka server注册时，eureka client提供自身的元数据，比如ip,port，运行状况指标的url，主页地址信息。

##### Renew服务续约

eureka client默认30s发送一次心跳进行服务续约。正常情况下eureka server如果90s没有收到client心跳，就会从注册列表删除该client，官方建议不要修改。

##### Fetch Registries获取服务注册列表信息

Eureka client从eureka server获取服务注册信息，并缓存在本地，可以使用json和xml来进行通信，默认xml。

##### Cancel服务下线

client下线时可以向eureka server发送请求，server端会删掉该注册服务信息。需要在客户端程序调用如下代码：

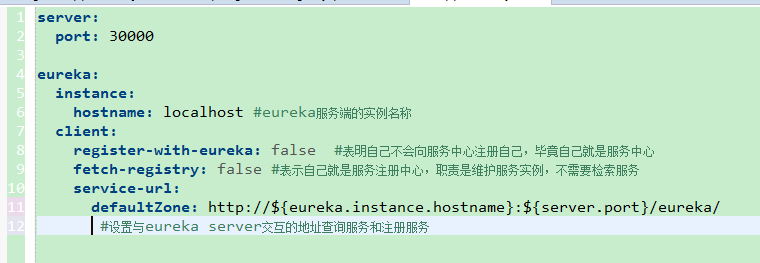


##### Eviction服务剔除

即90s server端没收到消息就会删除注册服务。

#### 服务端配置

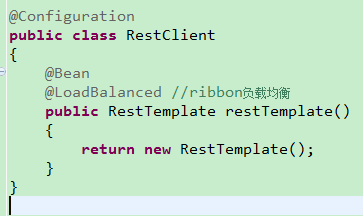








服务之间相互访问用restTemplate实现





#### eureka与zookeeper比较

（1）leader选举

如果某台服务器宕机，eureka不会有类似zookeeper的leader选举过程，客户端会自动切换到eureka新节点。当宕机的服务器重新恢复后，Eureka会再次将其纳入到服务器集群管理之中；而对于它来说，所有要做的无非是同步一些新的服务注册信息而已。所以，再也不用担心有“掉队”的服务器恢复以后，会从Eureka服务器集群中剔除出去的风险了。Eureka甚至被设计用来应付范围更广的网络分割故障，并实现“0”宕机维护需求。（多个zookeeper之间网络出现问题,造成出现多个leader，发生脑裂）当网络分割故障发生时，每个Eureka节点，会持续的对外提供服务（注：ZooKeeper不会）：接收新的服务注册同时将它们提供给下游的服务发现请求。这样一来，就可以实现在同一个子网中（same side of partition），新发布的服务仍然可以被发现与访问。

zookeeper在Leader选举过程中，服务是不可用的，且必须要有至少一半以上的服务能够使用才能提供服务。

（2）心跳服务

eureka与zookeeper都有心跳服务，会将部分心跳出现问题的服务剔除管理范围，但是它能更好地处理网络分割等而导致的大量心跳连接丢失问题，它有“自我保护模式”，会对“心跳死亡”的服务信息进行保留防止过期。此时，这个Eureka节点对于新的服务还能提供注册服务，对于”死亡“的仍然保留，以防还有客户端向其发起请求。当网络故障恢复后，这个Eureka节点会退出”自我保护模式“。这里相对于zookeeper更能体现高可用性。

（3）客户端缓存功能

Eureka还有客户端缓存功能（注：Eureka分为客户端程序与服务器端程序两个部分，客户端程序负责向外提供注册与发现服务接口）。所以即便Eureka集群中所有节点都失效，或者发生网络分割故障导致客户端不能访问任何一台Eureka服务器；Eureka服务的消费者仍然可以通过Eureka客户端缓存来获取现有的服务注册信息。甚至最极端的环境下，所有正常的Eureka节点都不对请求产生相应，也没有更好的服务器解决方案来解决这种问题时；得益于Eureka的客户端缓存技术，消费者服务仍然可以通过Eureka客户端查询与获取注册服务信息，这点很重要。

（4）丰富的功能：

Eureka的构架保证了它能够成为Service发现服务。它相对与ZooKeeper来说剔除了Leader节点的选取或者事务日志机制，这样做有利于减少使用者维护的难度也保证了Eureka的在运行时的健壮性。而且Eureka就是为发现服务所设计的，它有独立的客户端程序库，同时提供心跳服务、服务健康监测、自动发布服务与自动刷新缓存的功能。但是，如果使用ZooKeeper你必须自己来实现这些功能。Eureka的所有库都是开源的，所有人都能看到与使用这些源代码，这比那些只有一两个人能看或者维护的客户端库要好。

（5）更简单的维护

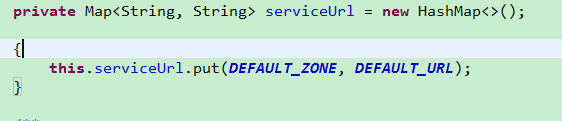
维护Eureka服务器也非常的简单，比如，切换一个节点只需要在现有EIP下移除一个现有的节点然后添加一个新的就行。Eureka提供了一个web-based的图形化的运维界面，在这个界面中可以查看Eureka所管理的注册服务的运行状态信息：是否健康，运行日志等。Eureka甚至提供了Restful-API接口，方便第三方程序集成Eureka的功能。

#### eureka特点

如果eureka客户端配置应用多个注册服务器（集群），那么默认情况下只有第一台可用的服务器存在注册信息，如果第一台可用的eureka服务器挂了，那么eureka客户端将会选择下一台可用的eureka服务器。

#### 配置源码EurekaClientConfigBean

配置项eureka.client.serviceUrl实际映射字段为serviceUrl，它是map类型，key为自定义，默认为”defaultZone”，value为需要配置的eureka注册服务器url。



eureka通过拉取数据实现数据同步，使用推上报信息。

当eureka界面服务/info不便于观察时（有可能服务换了个名字又重启，导致服务数更多，但实质上还是一个），可以修改

eureka.instance.statusPageUrlPath = /health来方便观察

#### eureka emergency

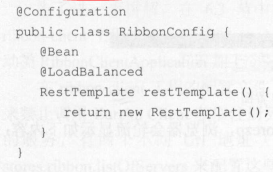
这个是Eureka的自我保护机制。Eureka Server在运行期间，会统计心跳失败的比例在15分钟之内是否低于85%，如果出现低于的情况（在单机调试的时候很容易满足，实际在生产环境上通常是由于网络不稳定导致），Eureka Server会将当前的实例注册信息保护起来（不删除），同时提示这个警告。

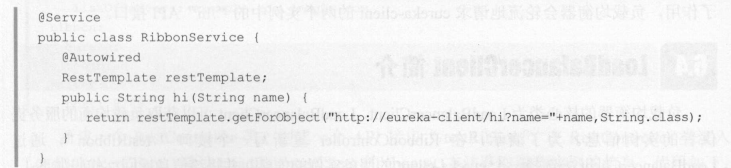
### ribbon

#### 作用

负载均衡。

#### 用法



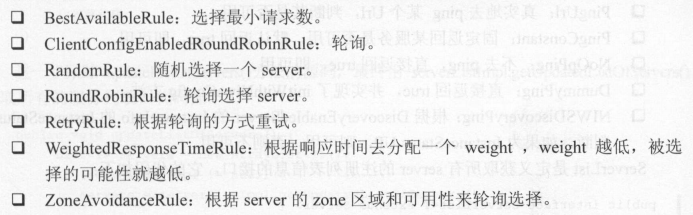


#### 原理

底层实现是RestTemplate增加一个LoadBalancedIntercepter拦截器，调用netflix 中的loadbalancer实现，根据eureka客户端应用调用获取目标应用的ip+port等信息，默认轮询的方式调用服务。如果禁用掉eureka，则需要自己手动去维护一份服务地址列表。

loadBalancerClient.choose

7种已实现轮询策略：



### hystrix

qps:query per second

tps:transaction per second

qps:经过全链路压测，计算极限qps,集群=单机qps\*机器数\*可靠性比率

全链路压测除了压极限qps，还有错误数量。

全链路：一个完整的业务流程操作。

#### 概念

提供熔断功能，组织分布式系统中出现联动故障（比如一点阻塞导致雪崩效应），通过隔离服务的访问点组织联动故障，并提供了故障的解决方案。

#### 设计原则（功能）

（1）防止单个服务故障耗尽整个服务器的线程资源

（2）快速失败机制，如果某个服务出了故障，则调用该服务的请求快速失败，而不是线程等待。

（3）提供回退（fallback）功能，在请求发生故障时，提供设定好的回退方案。

（4）使用熔断机制，防止故障扩散到其他服务。

（5）提供熔断器的监控组件hystrix dashboard，可以实时监控熔断器状态。

#### 运行原理

当服务的某个api接口失败次数在一定时间内大于设定的阈值时，hystrix判定该接口除了故障，打开熔断器，这时请求该api接口会快速执行失败的逻辑（fallback回退逻辑），不执行业务逻辑，请求的线程不会处于阻塞状态。处于打开状态的熔断器，一段时间后会处于半打开状态，并将一定数量的请求执行正常逻辑，剩余的请求会执行快速失败，若执行正常逻辑的请求失败了，则熔断器继续打开，反之关闭，即熔断器具有自我修复能力。

注意：熔断器是作用在客户端上的，有变相限流的效果。

### feign

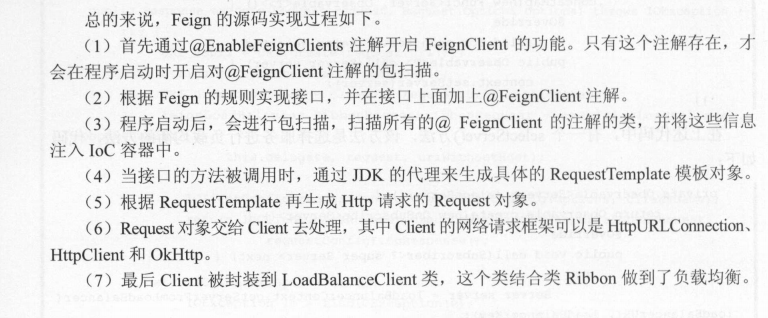
默认继承了ribbon。目标是将http客户端调用过程变得简单。

feign是一个伪java http客户端，本身不做任何处理，通过处理注解生成Request模板，从而简化了http api开发，开发人员可以使用注解方式定制requestapi模板，在发送http request请求之前，feign通过处理注解方式替换掉request模板中的参数，生成真正的request，并交给java http客户端去处理。利用这种方式，开发者只需要关注feign注解模板的开发，而不用关注http本身。

#### Client组件

client在feign源码中是一个接口，默认实现类是Client.Default，底层是HttpURLConnection实现网络请求。Client还支持HttpClient和Okhttp来进行网络请求。

#### 源码实现



#### ribbon和feign区别

Ribbon和Feign都是用于调用其他服务的，不过方式不同。

1.启动类使用的注解不同，Ribbon用的是@RibbonClient，Feign用的是@EnableFeignClients。

2.服务的指定位置不同，Ribbon是在@RibbonClient注解上声明，Feign则是在定义抽象方法的接口中使用@FeignClient声明。

3.调用方式不同，Ribbon需要自己构建http请求，模拟http请求然后使用RestTemplate发送给其他服务，步骤相当繁琐。

  Feign则是在Ribbon的基础上进行了一次改进，采用接口的方式，将需要调用的其他服务的方法定义成抽象方法即可，不需要自己构建http请求。不过要注意的是抽象方法的注解、方法签名要和提供服务的方法完全一致。

### zuul

## Tomcat优化

## 项目牛皮

### 稳健网络项目

#### redis怎么做的任务队列&任务跟踪？

（1）实现

任务队列分为解析任务和评估任务，使用的单机版的redis，客户端使用的jedis，使用redis列表以及lpush与rpop实现任务队列

（2）任务提交

每次前端提交任务时（task模块只有一个服务），将task信息lpush到任务list中。

（3）任务抢夺

每个解析/分析模块各有一个java scheduled线程，每隔3秒执行尝试抢一下任务锁（task分布式锁，该锁由redis实现，底层为setnx，过期时间为5秒），如果抢不到则休息1s后继续尝试tryLock，抢到后尝试计算当前内存和可用线程数是否还支持创建解析/分析任务，成功则lpop任务list中最外面的任务，否则释放锁，休息5s后尝试下一次抢夺。

（4）任务跟踪

每个解析or分析任务的具体百分比存储于redis中，也有过期时间，为1min，直到任务结束后从内存里删除任务，并将结果刷入task主数据表里。如果redis挂了则任务状态为重新启动。

（5）任务状态

任务状态主要为初始状态，解析排队状态，解析状态，解析完成状态，解析失败状态，评估排队状态，评估状态，评估评估完成状态，评估失败状态，用枚举实现。

#### 事务如何实现？

#### 常见问题定位

##### 内存溢出

使用poi读取数据报告时内存溢出。定位方式是通过查看日志时，发现在执行poi方法时产生对象过多导致内存溢出。

#### 运用设计模式

##### 模板方法模式

指标计算方法实现为一层一层抽象接口实现，还有excel事件驱动模式读取数据也是一层一层读取实现。

##### 策略模式

将指标算法大致分类，每个指标有一个具体的数值用作记录计算算法类型，定义一个算法接口，不同的算法实现该接口（或者父级抽象类），使用静态Map记录数值与计算方法的实现类（在配置文件中配置），每次调用就通过算法类型数值获取对应算法实现类后调用算法实现方法即可。新增算法只需要新增实现类后在配置文件加上全类名就行。

JDK中线程池的策略和Comparator比较接口都是策略模式的实现。

#### 如何防止重复提交？

##### 后台

提交任务时先加上一把状态锁（以任务id为准，比如xxx\_parsing），查询数据表（任务表一开始创建项目时任务便已初始化完毕），若状态为初始化，则修改状态为running后提交任务。如果已经为运行中则直接返回运行中。这样避免了进程1查询出来是start，还未来得及做修改进程2也查出来是个start，进程1改了提交任务后进程2又去启动任务处理的情况，不过这里好像spring事务也可以利用数据库的不可重复读隔离级别实现相同功能（进程2修改时出现异常直接回滚，不再提交解析任务）。锁状态大概5s。

### 预防预测项目

#### 内部安全认证如何实现

##### https

跳过了springcloud的enreka注册中心，feign客户端直接绑定具体url（当然是https的url），证书由华为提供，修改springboot的tomcat配置，将协议改为tlsv2.0，设置了多个errorpage，加入证书密码和路径，同时将证书导入jdk（改了lib下security中两个安全加密的jar包）。

在tomcat embeddedservletcontainercustomizer加入ssl配置，包含证书文件路径，证书存储密码，证书密码，证书类型，证书别名，同时设置ssl协议tlsv1.2，设置ciphers加密套件，将证书导入jdk库，

feign直接通过https://域名:端口/resturi访问

##### 访问token

服务之间的信任是设置了默认的机机账号来相互认证，相互访问时（主要是app访问其他模块）每次都需要首先获取访问token，使用spring-security+jwt来实现。首先将用户密码传递给被访问端，利用jwt解析并比较用户密码，成功即返回token，否则返回空串。访问者获取token后将token加入header的authorization字段（好像是这个？）才能继续访问具体内容

##### 平台的配置与认证

平台的配置统一由平台提供的配置jar包实现，到时候在Linux上直接用脚本解压jar包并将配置刷到服务配置中。

平台的sso为他们有一个cas认证中心，他们实现了一个ssoFilter并将该filter注册到各个产品线的web应用中，每个访问后台服务的请求都将经过该filter并且校验token，token校验也是一个服务，该请求url和相关配置一并包含在平台所给sso客户端中，如果token未通过校验则将跳到登录界面。

##### 项目性能优化

加索引

##### 服务信息注册

使用zookeeper客户端注册了一个临时节点，包含产品类型，主题号，服务器ip等各种信息，一旦服务挂掉则节点消失，平台也会感知到该消息，不过后续怎么处理，则不是产品侧负责。

##### Kafka安全

#### 项目问题定位

##### 内存溢出（与调整）

这个问题其实是在测试环境才会出现，由于服务较多，同时机器只有8个G，外加上该服务器还存在一些华为的监控进程，导致可用内存往往只有4-5个G左右。这时通过dump的快照文件，查看占用内存最大的那个对象，将其优化分批读取，其次调整各服务的内存大小，内存较小的注册中心和配置中心调整到只有0.5个G，其他根据数据量一次调大。

##### 内存泄漏

其实是初始化缓存的一个list结构中，代码误编写，每次使用时都莫名add了一些新的对象进去，将会导致内存一直增多直到溢出。是使用mat工具查看内存时发现某个对象一直持续增大。

##### 元空间溢出

##### 死锁（编？）

### 其他项目常见问题&解决方案

#### zookeeper注册中心与服务提供者断开连接，但服务提供者依然可用，如何解决？

#### spring事务失效场景？

#### 秒杀如何实现？

今天面试碰到面试官问秒杀 我说用redis list结构push方法解决秒杀 他说10000个人进来会导致连接超时或者连接满拒绝吗？

#### 分布式事务？

#### 分布式全局唯一id

#### git上的项目？

## 解决方案

#### 消息队列

##### 为什么要使用消息队列？

解耦：各个系统间不必关心系统的上下线等各种问题，只管往消息队列传递数据。

异步：系统之间接口调用，可以不用调用后等待结果，而是消息往消息发送后，直接返回结果，后续流程不必阻塞等待。

削峰：可以使大量的请求积压在消息队列，调整消费者的消费速度，消费者使用系统可扛住的请求处理速度逐步实现请求，避免系统被压垮。

其他：秒杀，幂等实现等。

##### 消息队列的缺陷？

（1）新增加了组件，增大了系统的不稳定性。

（2）加大了系统的复杂程度，增加了成本。

（3）部分消息队列存在消息重复发送等问题，意思就是一致性问题需要解决（也是增加系统复杂程度）。

##### 主流消息队列对比

ActiveMQ

每秒万级处理速度，成熟，但是社区已经不活跃，存在丢失消息的可能，同时是以主从实现高可用。

RabbitMQ

erlang实现，每秒万级处理速度，速度最快（微秒级），社区活跃，以主从实现高可用，图形界面特别不错，适合中小型企业。

RocketMQ

阿里出品，每秒十万级处理速度，吞吐量大，社区活跃，以分布式集群实现高可用，可通过调整参数实现消息不丢失，适合中大型企业。

Kafka

专用于大数据，功能简单，每秒十万级处理速度，吞吐量大，社区活跃，以分布式集群实现高可用，可通过调整参数实现消息不丢失，适合中大型企业。

##### 如何保证高可用性？

我这里只用过kafka，就只以kafka为例。

Kafka的高可用性实现了分布式（ActiveMQ与RabbitMQ没有实现，RocketMQ实现了），它将每个主题分为了多个Partition,partition可以分布在不同broker上（提高吞吐量），但这时还没保证高可用，于是0.8版本后每个Partition又加入了replication副本机制，即数据有多个备份在不同的broker上，每个partition的replications们会选出一个leader作为数据的读写，其他的副本则只需要同步。一旦某个broker挂了，或者说leader挂了，kafka会自动将其他副本选一个leader出来继续运行保证高可用。

##### 消息重复提交（幂等性）？

kafka是通过提交偏移量（offset）来确定消息消费位置的，如果消费者在消息消费后未提交offset后又重启，则之前消费的多条消息又会再次消费。

解决方案：

（1）最常见的：根据每一次消息的id（比如taskId等）进行校验，如果发现重复的就不再处理，之前的项目也是根据每次的taskId查库（id是否已存在）来确定的。

（2）或者在redis set一下id，如果再来一条重复数据，就去exist或者setnx（你懂的），如果已存在说明已消费。

（3）基于数据库的唯一键来保证重复数据不会重复插入多条。重复数据插入数据表，唯一键冲突报错，就不会插入重复数据。

##### 消息丢失如何解决？

消息可能丢失的三种情况：

（1）生产者往mq写消息丢失（网络数据丢失，mq自己未存储便挂了，生产者客户端缓冲区溢出等）。

（2）mq内部消息丢失（mq收到消息消费者未消费便自己挂了）。

（3）消费者往mq拉消息消息丢失（消费者刚收到消息未消费便挂了，但mq可能以为消费者已消费）。

参考如下博客：

<https://www.cnblogs.com/cherish010/p/9764810.html>

（1）生产者发送的消息丢失

如何规避？可以在降低吞吐量的情况下，大致有如下办法：

1）acks=all，这个配置是所有follower都收到消息了才认为提交成功。

2）使用异步+回调函数的客户端（用同步一条一条发吞吐量太烂），自定义回调逻辑处理消息发送失败。

等等，当然，最不怕数据丢失的，就是客户端同步加上ack=-1，但是吞吐量Boom。

（2）mq挂了（或者说kafka弄丢了数据）？

必须要保证消息持久化到磁盘上，才算成功，对于kafka来讲是至少要副本中的leader成功接收到了，才算成功。

至少要设置4个参数

broker:replication.factor>1，指partition副本至少有2个，可以理解，挂了一个broker，其他的还是可用。

broker:min.insync.replicas>1，指leader至少感知到有至少一个follower还跟自己保持联系（消息至少要被写入到这么多副本才算成功），这样可以确保leader挂了还有一个follower。

producer端:设置acks=all（0是不管，1是leader收到就行，-1或者all是全部接收），要求每条数据必须写入所有的replications副本后，才能认为成功了

producer端:retries=MAX，意思就是发送失败就会无限重试。

（3）消费者端数据丢失

关闭自动提交位移enable.auto.commit = false，采用手动提交策略提交消息，而且最好是业务正常处理完业务后，再将消息手动提交。

之前的项目中就是接收到任务消息后，只是将任务信息存入数据表后，才提交offset，后面的业务处理也与提交隔离，这是比较好的处理逻辑，因为这样如果在业务处理过程中出现了宕机等突发情况，重启后依然可以再次消费消息，保证是消息成功处理（消费）后，才提交的offset（向kafka确认消费完毕）。

还有消费者auto.offset.reset要配置为earlist，表示从已提交的offset开始消费。

##### 消息消费的顺序性？

这里还是以kafka为例（我tm只用过kafka，饶了我吧）

解决方案：

kafka写入一个partition的数据一定是有顺序的，生产者写数据时，会根据key使用指定(hash)计算后一定会分发到某一个partition中去，不会改变，所以指定一个固定的key，这样数据的顺序性就有保证。然后一个partition只能被一个消费者消费，所以消费者就一定能保证顺序消费。

多线程性能提升：

当然消费者搞多线程处理消息提高吞吐量的话，那么需要对每个应用线程都搞一个对应的内存队列（根据ConsumerRecord的key来划分队列，比如某个订单的消息，根据这个订单id搞一个队列，从broker接收的消息都按顺序到这个队列中，专门处理这个队列的线程处理完以后再提交offset，消费者挂了也不会造成数据丢失），不然顺序性就不能保证了。实际上这里提高吞吐量就是加快offset的提交。

异步发送消息时注意点：

针对异步发送消息产生的数据乱序,我们可以通过设置参数max.in.flight.requests.per.connection = 1来限制客户端在单个连接上能够发送的未响应请求的个数，设置此值是1 表示kafka broker在响应请求之前client不能再向同一个broker发送请求，这样便可以避免消息乱序。

当然，同步（阻塞）发送是没有这些问题的。。

##### 大量数据积压如何处理（包含数据过期）？

假设积压了上千万条数据。

解决方案比较复杂，快速处理积压数据：

修改之前的消费者程序，将接收的消息都重新发送到一个新的topic中，这个topic有数量超过10倍（或者更多）的partition，然后再新增对应个数的consumer消费。快速处理完毕后再恢复原来的消费者。

数据过期：

有些公司是不允许数据过期的，所以就没设置数据过期失效，如果真的过期了，那只能去查一下究竟缺失的数据时哪些，在低峰期进行数据补全（重灌数据到mq。。。）。

##### 如果让你自己设计一个mq，怎么设计？

这个就是考验对mq的原理理解了，主要考虑如下问题：

（1）可伸缩性

（2）数据持久化能力（磁盘）

（3）数据丢失问题

（4）高可用性

等等

##### Kafka其他知识点补充

（1）多次消费同一个Partition数据

Kafka的一个Partition，同一个group中只能有一个消费者可以消费该分区，想要再消费得划分为另一个group。

（2）auto.offset.reset

latest：意思是在偏移量无效的情况下，消费者从最新的记录开始读取数据。

earliest：意思是在偏移量无效的情况下，消费者从起始位置（实际上是已提交偏移量的下一个起始位）读取分区记录。

（3）消息分发机制

默认情况下kafka根据key采用hash取模算法，如果为null则随机分配一个分区（在”metadata.max.age.ms”的时间范围内随机选择一个）。

Metadata包含topic和partition和broker的映射关系，每一个topic的每一个partition，需要知道对应的broker列表是什么，follower是谁，leader是谁。

（4）消费者分区分配策略

a.Range strategy（范围分区）

根据partition个数除以消费者线程总数来决定如何分配分区。比如10个分区和3个线程，那么线程1（0,1,2,3），线程2（4,5,6），线程3（7,8,9）这样消费。会发现这种方式在多topic情况下会导致某些消费者线程总是多消费分区（弊端）。

b.RoundRobin strategy（轮询分区）

将所有的partition和consumer线程都罗列出来按照hashcode排序，最后通过轮询算法分配给partition消费线程。

使用轮询分区的条件：

1.每个主题的消费者实例具有相同数量的流

2.每个消费者订阅的主题必须是相同的。

（5）Kafka消费者rebalance（触发分区分配策略）

触发条件：

a.同一个消费者组增加了消费者

b.消费者离开当前所属的消费者组（停机或宕机，说白了消费者减少）

c.topic分区变化（新增）。

（6）谁来执行Rebalance以及消费者组的管理？

a.coordinator

Kafka提供一个coordinator执行消费者组管理，当消费者组第一个consumer启动时，它会去和kafka broker确认谁是他们组的coordinator，之后该group内所有成员都会和该coordinator进行协调通信。

b.如何确定？

消费者向kafka集群任意一个broker发送GroupCoordinatorRequest请求，服务端将返回负载最小的broker节点的id，并将该broker设置为coordinator。

c.join group过程

所有成员都会向coordinator发送join group请求，一旦所有成员都发送了join group请求，那么coordinator会选择一个consumer担任leader角色，并将组成员信息和订阅信息发送给消费者。

接下来就行Synchronizing Group State阶段，就是leader将消费者的partition分配方案同步给consumer group中所有的consumer。

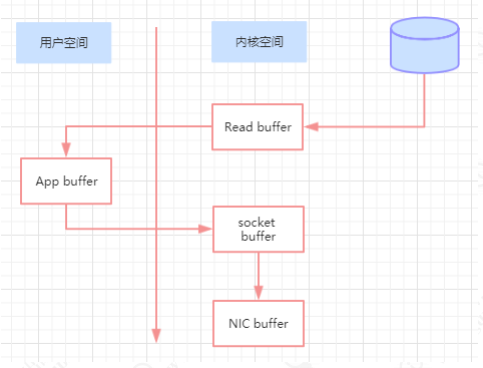
注意:Kafka消费者的分区方案是在客户端执行的，之所以这么做是为了更好的灵活性。

（7）Kafka消息存储以及offset存储

略。

（8）Kafka零拷贝策略

消息从发送到落地保存的过程：



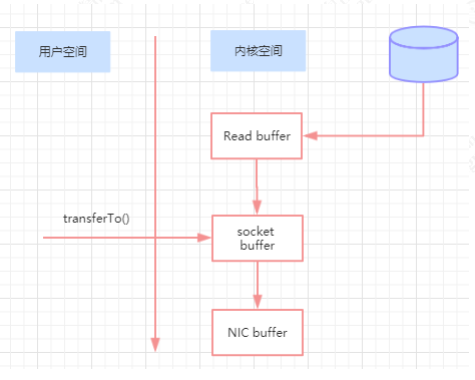
1.操作系统将数据从磁盘读入到内核空间的页缓存

2.应用程序将数据从内核空间读入到用户空间缓存中

3.应用程序将数据写回到内核空间到socket缓存中

4.操作系统将数据从socket缓冲区复制到网卡缓冲区，以便数据经网络发出。

零拷贝实现



linux使用sendfile系统调用来完成，Java则是提供FileChannel.transferToAPI实现。只需要一次拷贝就将数据从页缓存发送到网络上。

#### 缓存组件

##### 缓存的作用

（1）高性能：缓存数据库查询的数据，每次查询直接从内存获取，不再走sql查询，速度提升max。

（2）高并发：常用数据存储于缓存，减少高峰期数据库压力，防止数据库宕机（数据库一般并发不超过2000/s，但缓存可以轻易4w+/s）。

##### redis与memcached区别

（1）前者拥有更丰富的数据结构

（2）前者单线程，后者多线程

（3）前者原生支持cluster模式，后者不支持，只能在客户端自己实现分片。

memcached目前已不再是主流，接下来基本以redis为主进行面试题分析。

##### redis线程模型（待深入）

文件事件处理器

Redis基于reactor模式开发了网络事件处理器，叫文件事件处理器，file event handler，该处理器是单线程的，所以redis才叫单线程模型，采用I/O多路复用机制同时监听多个socket，根据socket上的事件来选择对应的事件处理器来处理这个事件。

主要包括4个部分：

多个socket，IO多路复用程序，文件事件分派器，事件处理器（连接应答处理器，命令请求处理器，命令回复处理器，等等）。

流程：

（1）客户端每次请求访问产生一个socket会话。

（2）IO多路复用程序监听会话产生的AE\_READABLE,AE\_WRITEABLE等事件，将socket压入一个队列。

（3）文件事件分派器判断socket事件，选择对应事件处理器处理，处理完后再从队列获取socket处理。

1）连接应答处理器：将socket的AE\_READABLE事件与命令请求处理器向关联。

2）命令请求处理器：读取socket的key和value，在内存中完成key和value的设置，将socket的AE\_WRITEABLE事件跟命令回复处理器相关联。

3）命令回复处理器：对socket输出操作的结果，将socket的AE\_WRITEABLE事件跟命令回复处理器解除关联。

##### redis单线程为什么能支持高并发？

redis单机，1s能处理几万请求。

（1）非阻塞IO多路复用机制（只负责监听socket请求然后将其压入队列，速度当然快）。

（2）纯内存操作。

（3）单线程避免了多线程频繁上下文切换问题。

（4）简单的数据结构（不是很重要，可答可不答）。

##### redis数据类型与应用场景

请注意redis的五种类型都是redis实现的数据结构，而不是redis本身的类型。

（1）string

简单的kv缓存。

（2）set

无序集合，自动去重，用于数据的快速去重。

可以玩交集、并集、差集的操作。比如查看粉丝的共同的好友等。

（3）sorted set(zset)

排序的set，根据分数自动排序。

最简单的就是排行榜。

（4）hash

类似map的一种结构，处理结构化的数据，比如一个简单的对象（没嵌套其他的对象）。

（5）list

有序列表，可以实现很多功能。

比如某个XXX的粉丝列表（key作者名，list是粉丝列表），文章评论列表等。

通过lrange命令从某个元素开始读取多少个元素，实现高性能分页查询，类似于微博那种下拉不断分页的东西。

简单的消息队列，坐进右出或者右进左出，任务队列也行。

##### redis key过期策略（内存回收策略）

主要是定时删除+惰性删除。

惰性删除：访问设置过期属性的键时，发现键过期即删除，同时返回空。但是这样如果久了不访问，同样挂在内存，有造成内存溢出的风险。

定时任务删除：维护一个定时任务，默认每秒运行10次（通过配置hz控制）。删除键的逻辑采用了自适应算法，根据键的过期比例、使用快慢两种速率模式回收键。

（1）定时任务在每个数据库空间随机检查20个键，发现过期时删除对应的键

（2）如果超过检查数25%的键过期，循环执行回收逻辑直到不足25%或运行超时为止，慢模式下超时时间为25ms。

（3）如果之前回收逻辑超时，则在redis触发内部事件之前再次以快模式运行回收过期键任务，快模式下超时时间为1ms且2秒内只运行1次。

（4）快慢两种模式内部删除逻辑相同，只是执行的超时时间不同。

##### redis内存淘汰（溢出控制）策略

lru算法：实际也就是最少使用算法。

当redis所用内存达到maxmemory上限时会触发相应的溢出控制策略。具体策略受maxmemory-policy参数控制。一共6种策略：

（1）noeviction：默认策略，不删除任何数据，只接受读操作并拒绝所有写操作并给客户端返回错误信息。

（2）allkeys-lru：根据lru算法删除最近最少使用的key（不管有没有过期属性），直到腾出足够空间为止。

（3）allkeys-random：随机删除所有键，直到腾出足够空间为止。

（4）volatile-lru：根据lru算法（最近最少使用）删除设置了超时属性的键，直到腾出足够空间为止，否则回退noeviction。

（5）volatile-random：随机删除过期属性键，直到腾出足够空间为止。

（6）volatile-ttl：根据键值对象的ttl属性，删除最近将要过期的数据，如果没有回退noeviction。

总结：一noeviction默认不删,二allkeys随机删，三volatile删过期属性键。

##### Redis如何保证高并发？

一般来说单机redis能够承载5W的QPS，但假如说需要有20W的QPS呢？

解决方案：

使用主从（master-slave）模式，主写从读，以上面案例来说，1主写，4从读，可以满足20W的QPS。

##### redis主从数据同步

主节点每次收到写命令后，先在内部写入数据，然后一步发送给从节点。

##### redis主从replication的特点

（1）redis采用异步方式复制数据到slave节点，不过redis2.8开始，从节点会周期地确定自己每次复制的数据量。

（2）一个主节点可配置多个从节点，从节点可以连接其他从节点。

（3）从节点复制时，不会阻塞主节点工作，也不会阻塞自己的读操作。但是复制完成时，需要删除旧数据集，加载新数据集，就会暂停服务。

（4）从节点主要用于横向扩容，读写分离提高吞吐量。

注意：主从架构一定要开启主节点的持久化！不建议用丛节点作为主节点的数据热备，一旦主节点宕机重启，如果没有主节点的持久化，可能启动时主节点数据是空的，一启动从节点数据复制后也丢了。哪怕采用了哨兵机制，在主节点没有持久化的情况下也可能造成数据丢失故障。

##### redis主从replication的核心机制

（1）offset，offset记录数据偏移量，slave每秒都会上报offset给主节点，主节点也会记录每个从节点的offset。

（2）backlog，用于断点续传，默认1M，超过该log的offset还是会触发全量复制

（3）master run id，redis运行时自动生成，用于唯一确定master，比host+id靠谱。

（4）psync，从节点使用该命令从主节点同步复制数据。

##### redis主从replication的原理

启动从节点时，会发送一个psync命令给主节点。

如果从节点重新连接主节点，主节点仅复制从节点缺少的数据，如果是第一次连接，就会触发一次全量复制。

如果传输过程断掉，重连后触发断点续传。如果无法满足断点续传条件，或者有多个从节点都来重连，那么只会执行一个rdb save操作，用一份数据同步所有从节点。

主从节点都会保存一个数据的offset来确定数据是否同步，以及偏差量。

##### redis主从的全量复制

全量复制时，Redis会创建一个线程，执行save生成一份rdb快照文件，同时缓存这个时间点以后后续的写操作。rdb生成完成后发送给从节点，从节点先将其写入磁盘，然后加载到内存里，然后再同步主节点缓存的写命令。

复制过程需要注意几个参数：

（1）repl-timeout，默认60s，超过该时间从节点会认为复制失败，一般4G需要1分半-2分钟时间，应适当调大该参数。

（2）client-output-buffer-limit slave 256MB 64MB 60,如果在复制期间，内存缓冲区持续消耗超过64MB，或者一次性超过256MB，那么停止复制，复制失败。

（3）如果从节点开启了AOF，那么很可能会立即执行BGWRITEAOF，重写AOF。

##### Redis主从的断点续传（增量复制）

从redis2.8开始支持该功能，即主从复制过程中，网络断掉了，可以接着上一次断掉的传，而不是重头复制一份。

实际实现是内存里有一个backlog，只要从节点已复制的offset能够在backlog找到，那么就能继续复制，否则还是会执行全量复制。

##### Redis主从的无磁盘化复制

即master在内存中直接创建rdb发送给slave，不经过磁盘落地。感觉不是很重要，有点考验服务器的内存。

repl-diskless-sync

repl-diskless-sync-delay,等待一定时长再开始复制，因为要等更多slave重新连接过来。

##### Redis主从的过期key处理

主节点过期key，或者lru淘汰一个key，那么会模拟一条del命令给从节点，从节点不会自己过期key。

##### Redis主从的heartbeat

主节点默认10s发送一次心跳，从节点每隔1s发送一次心跳。

##### Redis如何保证99.99%高可用？

99.99%高可用大致就是系统可用时间/（可用时间+故障时间）大于等于99.99%。

slave节点挂了还能保持可用，master挂了直接gg，故引入哨兵模式支持高可用。

##### Redis Sentinel哨兵模式

特殊的Redis组件进程实例，几乎不支持任何redis读写操作，提供集群监控，消息通知，故障转移，配置中心功能。哨兵本身也是分布式的，即部分哨兵节点挂了，依然能提供服务。其master故障转移的master选举类似于zookeeper算法，需要一半以上节点（quorum）同意方可执行。

特点：

（1）至少需要3个实例来保证健壮性

（2）不保证数据丢失（Master数据还未复制便挂掉），只保证可用性

（3）哨兵+主从的复杂结构，需尽量在测试环境和生产环境多进行测试和演练。

经典的哨兵模式就是三台服务器，1主+2从+3哨兵。

##### Redis Sentinel的异步复制和集群脑裂的数据丢失问题

异步复制：master部分数据还未完全复制给从节点（异步的）就宕机重选master，导致数据丢失。

脑裂：由于网络分区等因素原master断开连接后又重连，这时已经重选出新的master，导致两个master存在，脑裂。假如原客户端还保持往原master写数据，如果原master变成新master的从节点，那么新写入的数据将会丢失。

解决：

min-slaves-to-write 1

min-slaves-max-lag 10

要求至少有1个slave的数据复制和同步的延迟不能超过10s.（1与10可调整）

如果一旦所有的slave数据复制和同步的延迟都超过了10s，master就会拒绝任何请求。

可减少异步复制和脑裂导致的数据丢失。

##### Redis Sentinel原理

（1）sdown和odown

sdown主观宕机：如果一个哨兵自己觉得一个master宕机了，那么就是主观宕机

odown客观宕机：如果quorum数量的哨兵都觉得一个master宕机了，那么就是客观宕机。

sdown条件：如果一个哨兵ping一个master，超过了is-master-down-after-milliseconds指定的毫秒数之后，就主观认为master宕机

sdown转odown条件：如果一个哨兵在指定时间内收到quorum数量的其他哨兵也认为那个master宕机了，就客观认为客户端master宕机。

（2）哨兵和slave集群的自我发现机制

哨兵互相之间的发现，是通过 redis 的 pub/sub 系统实现的，每个哨兵都会往 \_\_sentinel\_\_:hello 这个 channel 里发送一个消息，这时候所有其他哨兵都可以消费到这个消息，并感知到其他的哨兵的存在。

每隔两秒钟，每个哨兵都会往自己监控的某个 master+slaves 对应的 \_\_sentinel\_\_:hello channel 里发送一个消息，内容是自己的 host、ip 和 runid 还有对这个 master 的监控配置。

每个哨兵也会去监听自己监控的每个 master+slaves 对应的 \_\_sentinel\_\_:hello channel，然后去感知到同样在监听这个 master+slaves 的其他哨兵的存在。

每个哨兵还会跟其他哨兵交换对 master 的监控配置，互相进行监控配置的同步。

其他略。

<https://github.com/ThreeCatCold/advanced-java/blob/master/docs/high-concurrency/redis-sentinel.md>

##### Redis重启的数据恢复（持久化）

redis宕机或者重启，需要通过加载持久化的数据恢复数据，可能会丢失少许数据，但不会所有数据都丢失。

##### Redis持久化之RDB

RDB（redis database）：对redis数据进行周期性持久化，生成xxx.rdb文件。

（1）触发条件

a.手动触发分为save和bgsave命令，

save：阻塞当前redis服务器，直到rdb过程完成为止，数据量较大可能会造成长时间阻塞，线上不建议使用。

bgsave：redis进程执行fork操作创建子进程，rdb持久化由子进程负责，完成后结束，阻塞只发生在fork阶段，一般时间很短。

b.自动触发：save m n配置，表示m秒内数据集存在n次修改时，自动触发bgsave。

比如如下配置（redis.conf）：

(save seconds changes)

save 900 1 ：900秒内如果有1个key被更改，则触发快照

save 300 10：300秒内如果有10个key被更改，则触发快照

save 60 10000：60秒内如果有10000个key被更改，则触发快照

##### Redis持久化之AOF

AOF（append only file），将每条写入命令作为日志（appendonly.aof），以append-only的模式追加到一个日志文件中（打开文件可以看见全是可阅读命令，有check-redis-aof的恢复工具），默认每隔1s执行一次fsync刷新操作。

（1）aof问题

文件会特别大，性能消耗比rdb高，恢复更慢。

（2）AOF配置

aof重写压缩配置

auto-aof-rewrite-percentage 100 超过上一次重写的AOF文件的大小的多少时会执行重写，默认100%

auto-aof-rewrite-min-size 64mb 允许重写的最小的aof文件大小，默认64mb

也是fork一个子进程，将内存里面的数据写出来。

重写aof文件时，服务依然能接收外部请求，将会有重写缓存，等aof重写完成后将数据追加到后面。

appendfsync everysec（always/no），默认每秒将缓存数据（操作系统的缓存）同步至磁盘上(AOF文件).

##### Redis持久化之RDB与AOF的比较与使用

（1）使用

如果只想将redis作为纯内存的缓存，可以禁用掉RDB和AOF的持久化机制。

如果RDB与AOF两种机制都使用，那么恢复数据是用AOF，因为AOF的数据更加完整。

RDB记录的是redis某个时刻的数据，定时将rdb快照存储到某些存储服务器可以用来做冷备。AOF也可以采用脚本等手段定时copy AOF文件。

AOF甚至可以做到0丢失，每写入一条数据即执行fsync刷新AOF文件，但是这样QPS炸裂。

如果需要持久化，一般AOF和RDB都启用，光启用RDB丢失更多数据，光启用AOF恢复速度慢，并且不如RDB简单健壮。两者都启用可以做到不同时间的冷备（rdb）和第一时间恢复数据的冷备（aof）。

（2）比较

a.加载与加载性能：rdb加载速度远大于aof，并且rdb对redis的读写服务影响非常小，因为是主进程fork一个子进程去执行rdb持久化，但如果fork的文件特别大，也会导致毫秒级的服务不可用。

b.占用空间：aof远大于rdb，并且需要根据策略重写aof减少空间占用（重写优化写操作记录，生成更小aof文件并删除旧文件），rdb是二进制压缩文件所以占用小很多。

c.数据丢失：aof能做到秒级数据丢失（理论上是1s数据，但由于其他一些原因一般会大于1s,原因待学习），而且可以通过修改aof文件的格式强行恢复数据（比如误操作flushall，可以直接改aof文件去掉改命令），rdb丢失数据较多，往往能丢失5分钟甚至更多的数据，所以一般不要让rdb的间隔太长。

d.qps的影响：AOF模式下的QPS低于RDB。

##### Redis集群Cluster

解决单master内存容量问题，存储海量数据

redis master（多master+读写分离+高可用）

对比主从+哨兵：

数据量少，用主从+哨兵，多几个slave增加读吞吐量

redis cluster针对海量数据+高并发+高可用。

##### Redis集群 hash&一致性hash&hash slot

数据分片算法升级：

hash->一致性hash（memcached）->redis cluster:hash slot

用于解决多个master节点，数据如何分布到这些节点上。

（1）hash

根据key计算hash值根据节点数取模，根据取模结果决定放置于哪个节点，但一旦节点数变更将会造成大量节点缓存数据的重新构建（节点越多越接近100%），一样可能造成缓存穿透击垮mysql。

（2）一致性hash

同样计算hash值，根据该值顺时针在hash环（一个圆环，上面有一些固定节点）上对应的各个节点比对，找距离最近的节点，决定落于该点。

相比于hash好处就是如果节点有变动，只会造成之前在的那个master上的数据收到影响，master宕机后，会顺时针走到下一个master去。

当然如果丢失的节点的数据特别多，可能会导致大量的数据都涌入一个master，造成缓存热点问题。

为了解决上问题然后又搞了虚拟节点的概念，增多了节点数（实际物理节点可没这么多，比如节点1本来只有一个节点，给整了更多的1节点分散到整个环上，虚拟分布）。



（3）哈希槽

一共固定16384个槽位，对每个key计算CRC16值，然后对16384取模，即可以通过key获取对应的hash slot。

cluster中每个master会持有部分slot，比如3个master基本会每个master持有5000多个slot。

hash slot让node的增加和移除很简单，增加一个master，就将其他master的hash slot移动过去，减少一个master，就将它的hash slot移动到其他master上，移动hash slot的成本非常低。最多也就1个master的数据重建。

客户端api可以对指定的数据，让它们走同一个hash slot，通过hash tag来实现。

##### Redis事务

最好别用，redis事务不支持一般关系型数据库的事务特性，不支持回滚以及原子性。

##### 缓存的问题

（1）缓存与数据库一致性问题

（2）缓存雪崩

（3）缓存穿透

（4）缓存并发竞争