# Cookie和Session的区别

## 概念

浏览器访问服务器时，服务器会创建一个session对象，返回JSESSIONID=ID，并创建一个Cookie对象key为JSSIONID，value为ID的值，将这个Cookie写回浏览器。

浏览器在第二次访问服务器的时候携带Cookie信息JSESSIONID=ID的值，如果该JSESSIONID的session已经销毁，那么会重新创建一个新的session再返回一个新的JSESSIONID通过Cookie返回到浏览器。

session是基于Cookie技术实现，重启浏览器后再次访问原有的连接依然会创建一个新的session，因为Cookie在关闭浏览器后就会消失，但是原来服务器的Session还在，只有等到了销毁的时间会自动销毁。除非浏览器cookie有效期设置的较长，一般浏览器默认是存储cookie 如果你没设置关闭的话cookie在浏览器关闭后还是存在的。

浏览器可以禁用cookie，这时候想用session，就得通过服务器端程序重写URL了。

## 大小限制

cookie不能太大，传输不方便。各种浏览器限制了cookie大小，最高允许50个cookie，最大4096字节。

session是保存在服务器端，理论上是没有限制，只要你的内存够大。

但是Session是保存在服务器端上会存在一段时间才会消失，如果session过多会增加服务器的压力。

## 安全性

Cookie有安全隐患，通过拦截或本地文件找得到你的cookie后可以进行攻击。

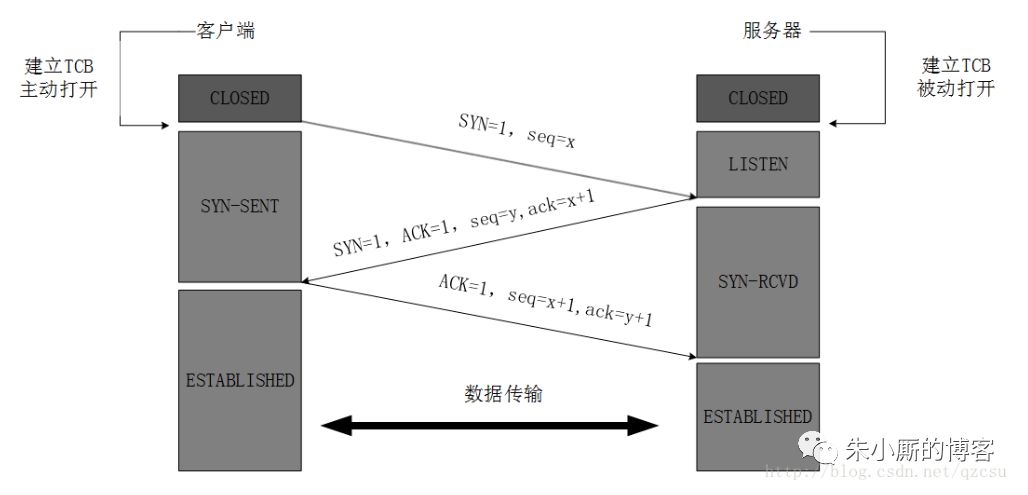
Session数据无法伪造，因为数据存在服务器。

登录信息等重要信息要存放在session，避免被盗。

# TCP概念

TCP报文的字段实现了TCP的功能，标识进程、对字节流拆分组装、**差错控制、流量控制**、**建立和释放连接**等。

## 三次握手



需要记的点是：

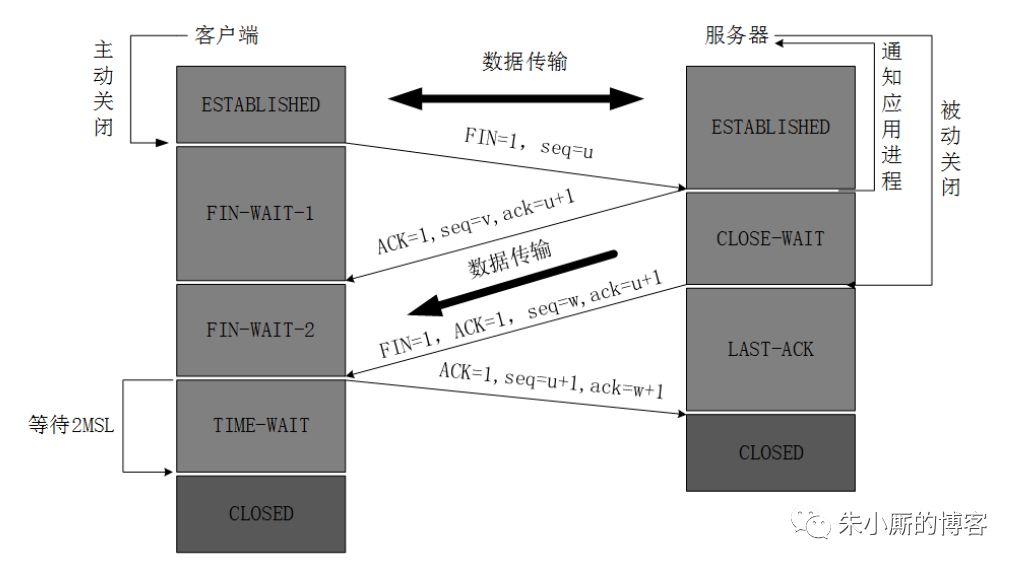
1、客户端的状态依次是CLOSED、SYN-SENT、ESTABLISHED，服务器的状态依次是CLOSED、LISTEN、SYN-RCVD、ESTABLISED。

2、客户端发了两次包，服务端发了一次包。第一次，客户端发的是SYN=1，seq=x。第二次，客户端发的是ACK=1，seq=x+1，ack=y+1。服务端发的是SYN=1，ACK=1，ack=x+1，seq=y。

**三次握手主要目的是：**信息对等和防止超时。（两次握手在超时的情况下会导致建立多条连接，浪费服务器资源。）

三次握手可以确认双发收发功能都正常，两次握手服务器端不能确定客户端是否能收到，只能确定客户端可以发服务端可以收，但是服务端能否发出去客户端能否收到是不确定的。两次握手也不能确定客户端的请求是否有效，比如客户端发的请求因为延迟送达，导致客户端重发请求，最后服务端接到两条客户端的请求，服务端会认为两个请求都是有效的。

## 四次挥手



CLOSED\_WAIT的含义：TCP服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个CLOSE-WAIT状态持续的时间。

## 异常

三次握手：

（1）client第一个syn包丢失，没有收到server的ack，则client进行持续重传syn包。总尝试时间为75秒。

（2）server收到了client的syn，发出了syn+ack包，syn+ack包丢失。client方面，因为没收server的。将执行情况（1）；server方面，超时时间内没有收到client的ack包（或者数据包），会持续发送syn+ack包。

server收到client的SYN包并回复SYN+ACK包后，server会把这条连接放入“半连接队列”。这边有个问题，假设这个client是恶意的，client只发SYN包，收到SYN+ACK后不回复，那在server方面，会一直存有这条“半连接”，client数量达到一定程度，server就炸了。这就是所谓的SYN FLOOD攻击；

（3）当Client端收到Server的SYN+ACK应答后，其状态变为ESTABLISHED，并发送ACK包给Server。如果此时ACK在网络中丢失，那么Server端该TCP连接的状态为SYN\_RECV，并且依次等待3秒、6秒、12秒后重新发送SYN+ACK包，以便Client重新发送ACK包。Server重发SYN+ACK包的次数，可以通过设置/proc/sys/net/ipv4/tcp\_synack\_retries修改，默认值为5。如果重发指定次数后，仍然未收到ACK应答，那么一段时间后，Server自动关闭这个连接。

如果此时client向server发送数据包，server能正常接收数据。则认为连接已正常。

四次挥手：

（1）client发的FIN包丢了，对于client，因为没收对应的ACK包，应当一直重传(像普通包一样)，直至到达上限次数，直接关闭连接；对于server，无任何感知。

（2）server回client的ACK包丢了，对于client，将执行（1），对于server将像丢普通的ack一样，再次收到FIN后，再发一个ACK包；

（3）如果client收到ACK后，server直接跑路，client将永远停留在这个状态（半打开状态，就像client关闭了输出一样）。linux有tcp\_fin\_timeout这个参数，设置一个超时时间 cat /proc/sys/net/ipv4/tcp\_fin\_timeout 查看，默认60s，可否修改看linux具体版本； windows 注册表有TcpTimedWaitDelay，win10默认值30s；

（4）server发的FIN包丢了，对于server，像丢普通的包一样，重传。若此时client早已跑路且与其他人建立连接，client应会不认识这个FIN包，直接回个RST包给server。如若client没跑路，且没收到server的FIN包，如（3）描述

（5）client回复ACK后，按道理来说，可以跑路了，但防止回复的ACK包丢失（丢失后，server因为没收FIN的ACK，所以会再发一个FIN），将等待2MSL(最大报文存活时间)（RFC793定义了MSL为2分钟，Linux设置成了30s）为什么要这有TIME\_WAIT？为什么不直接给转成CLOSED状态呢？主要有两个原因：1）TIME\_WAIT确保有足够的时间让对端收到了ACK，如果被动关闭的那方没有收到Ack，就会触发被动端重发Fin，一来一去正好2个MSL，2）有足够的时间让这个连接不会跟后面的连接混在一起（你要知道，有些自做主张的路由器会缓存IP数据包，如果连接被重用了，那么这些延迟收到的包就有可能会跟新连接混在一起），这期间如若再收到server的FIN，则再回复ACK；

防止SYN FLOOD攻击：

1、减少SYN-RECEIVED定时

2、防火墙。用一个简单的防火墙规则阻止带有攻击者IP地址的数据包就可以了。这种方法在如今的防火墙软件中通常都是自动执行的。

3、代理。一个有防火墙或者代理的设备在网络中就能够通过两种方法缓冲SYN洪泛攻击，一种是对连接发起人伪装SYN-ACK包，另一种是对服务器伪装ACK包。

**对连接发起人伪装SYN-ACK包：**由防火墙/代理来响应SYN包。如果连接发起人是合法的，防火墙/代理就会收到ACK，然后在它自己和服务器之间建立连接并伪装连接发起人的地址。只要防火墙/代理实现了一些基于TCP的防御策略，比如SYN cookies或SYN 缓存，他就能够保护所有在其后面的服务器免于SYN洪泛攻击。

**对服务器伪装ACK包：**ACK包通过防火墙/代理到达服务器。这种伪装防止服务器的TCB一直停留在SYN-RECEIVED状态，因此保证了backlog队列中的空余空间。防火墙/代理将会停留等待一段时间，如果连接发起人正确的ACK没有被检测到，它将会通过伪装的TCP RET报文使服务器释放TCB。

# HTTP与HTTPS

## 区别

一、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443

二、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议

## HTTPS工作原理

* 首先服务端给客户端传输证书,这个证书就是公钥,只是包含了很多的信息,比如说证书的颁发机构,证书的过期时间
* 客户端进行证书的解析,比如说验证颁发机构,过期时间,如果发现没有任何问题,就生成一个随机值(私钥),然后用证书对这个私钥进行加密,并发送给服务端
* 服务端使用私钥将这个信息进行解密,得到客户端的私钥,然后客户端和服务端就可以通过这个私钥进行通信了
* 服务端将消息进行对称加密(简单来说就是将消息和私钥进行混合,除非知道私钥否则服务进行解密),私钥正好只有客户端和服务端知道,所以信息就比较安全了
* 服务端将进行对称加密后的消息进行传送
* 客户端使用私钥进行信息的解密

SSL（Secure Socket Layer）工作在传输层和应用层之间。

**RSA算法：**

公钥（n，e）

私钥（n，d）

加密：信息是m，（m是对称秘钥对应的数字，m必须小于n）。计算m的e次方，对n取模，得到c

解密：计算c的d次方，对n取模，必然得到m

e与n的关系：其中n是两个不相等大质数p、q的乘积，n的欧拉函数值是p-1、q-1的乘积。e是小于欧拉函数值的数，且与欧拉函数值互质。实际中，e取65537，它是一个质数，也是2的16次方加1。

d与n的关系：e与d的乘积，模欧拉函数值，余1。即d是e的模反元素。

如果能把n还原成p、q，那么我们就可以知道n的欧拉函数值，从而根据e求出d。但是把n还原成p、q，是很难的事，目前没找到好的方法。

<http://www.ruanyifeng.com/blog/2013/07/rsa_algorithm_part_two.html>

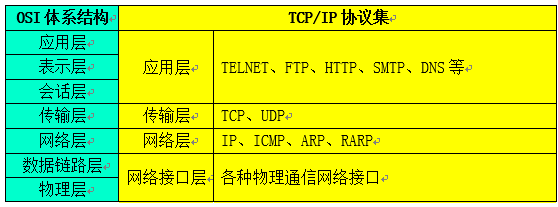
## 输入url发生了什么

（1）DNS解析：比如把[www.baidu.com](http://www.baidu.com)解析成IP地址。DNS解析是递归查找，首先查询本地的host文件、浏览器缓存等，如果没有查询到相应的IP地址，那么浏览器就会发送一个 DNS 请求到本地DNS服务器；然后本地DNS服务器首先会查询自己的缓存，如果缓存中存在直接返回相应的IP地址，如果不存在，会把请求转发给根域名服务器。

（2）建立TCP连接：拿到域名对应的IP地址后，会以随机端口（1024~~65535）向WEB服务器程序80端口发起TCP的连接请求，这个连接请求进入到内核的TCP/IP协议栈（用于识别该连接请求，解封包，一层一层的剥开），最终到达WEB程序，最终建立了TCP/IP的连接。

（3）建立起TCP连接后，浏览器开始发送 HTTP 请求。http是工作在应用层的协议，http请求报文加上TCP头部，封装成数据包，向下传输。数据包经过IP协议，以太网协议添加源IP地址，目标IP地址，源mac地址，目标mac地址等头部信息，层层封装后形成最终的数据包，通过网络接口层或者物理层进行传输。

（4）解析http数据：解析获取到的HTML、CSS、JS、图片等资源。



## OSI体系

应用层：TFTP，HTTP，SNMP，FTP，SMTP，DNS，Telnet 等等

传输层：TCP，UDP

网络层：IP，ICMP，OSPF，EIGRP，IGMP

数据链路层：SLIP，CSLIP，PPP，MTU

# JAVA

## 锁

对象锁与类锁：

（1）当两个并发线程访问同一个对象object中的这个synchronized(this)同步代码块时，一个时间内针对该对象的操作只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。

（2）但是另一个线程仍然可以访问该object中的非synchronized(this)同步代码块。

（3）当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对该object中所有其它synchronized(this)同步代码块的访问将被阻塞。

（4）同步加锁的是对象，因此，如果你的类中有一个同步方法，这个方法可以被两个不同的线程同时执行，只要每个线程自己创建一个的该类的实例即可。不同的对象实例的synchronized方法是不相干扰的。也就是说，其它线程照样可以同时访问相同类的另一个对象实例中的synchronized方法。

（5）synchronized关键字是不能继承的，也就是说，基类的方法synchronized f(){} 在继承类中并不自动是synchronized f(){}，而是变成了f(){}。继承类需要你显式的指定它的某个方法为synchronized方法。

**对一个全局对象或者类加锁时，对该类的所有对象都起作用。例如synchronized(ClassName.class) {}，锁住了所有ClassName对象，例如public synchronized static void method() {}，锁住了这个类的所有对象，因为静态方法是属于类的而不属于对象的。**

**ReentrantLock：**

从Java 5开始，引入了一个高级的处理并发的java.util.concurrent包，它提供了大量更高级的并发功能，能大大简化多线程程序的编写。我们知道Java语言直接提供了synchronized关键字用于加锁，但这种锁一是很重，二是获取时必须一直等待，没有额外的尝试机制。

例如：

public class Counter {

private int count;

public void add(int n) {

synchronized(this) {

count += n;

}

}

}

可以用ReentrantLock改写：

public class Counter {

private final Lock lock = new ReentrantLock();

private int count;

public void add(int n) {

lock.lock();

try {

count += n;

} finally {

lock.unlock();

}

}

}

可以尝试获取锁，获取不到就干别的事：

//尝试获取锁的时候，最多等待1秒。如果1秒后仍未获取到锁，tryLock()返回false，程序就可以做一些额外处理，而不是无限等待下去。可以避免死锁

if (lock.tryLock(1, TimeUnit.SECONDS)) {

try {

...

} finally {

lock.unlock();

}

}

synchronized可以配合wait（释放synchronized锁住的对象）和notify实现线程在条件不满足时等待，条件满足时唤醒，用ReentrantLock我们怎么编写wait和notify的功能呢？答案是使用Condition对象来实现wait和notify的功能。

class TaskQueue {

private final Lock lock = new ReentrantLock();

private final Condition condition = lock.newCondition();

private Queue<String> queue = new LinkedList<>();

public void addTask(String s) {

lock.lock();

try {

queue.add(s);

condition.signalAll();

} finally {

lock.unlock();

}

}

public String getTask() {

lock.lock();

try {

while (queue.isEmpty()) {

condition.await();

}

return queue.remove();

} finally {

lock.unlock();

}

}

}

Condition提供的await()、signal()、signalAll()原理和synchronized锁对象的wait()、notify()、notifyAll()是一致的，并且其行为也是一样的。

<https://www.liaoxuefeng.com/wiki/1252599548343744/1306581033549858>

new ReentrantLock(true)可以指定公平锁，让等待时间最长的线程优先获得锁。

例如用ReentrantLock写一个阻塞队列：

public class SimpleQueue {

private static ReentrantLock lock = new ReentrantLock();

private T[] nodes;

private int tail = 0; // 入元素下标

private int count = 0; // 元素个数

private int head = 0; // 出元素下标

public SimpleQueue(int size) {

nodes = (T[]) new Object[size];

}

private static Condition notFull = lock.newCondition();

private static Condition notEmpty = lock.newCondition();

public void put(T t) {

try {

lock.lock();

if (count == nodes.length) { // 队列已满，阻塞

System.out.println("目前队列已满，等待取值中");

notFull.await();

}

if (tail > (nodes.length - 1)) { // 当前游标值已经大于数组游标最大值了，则从0开始计算

tail = 0;

}

nodes[tail] = t; // 给当前游标位赋值

count++; // 入元素元素个数+1

tail++; // 游标值+1

notEmpty.signalAll(); // 走到这里说明队列内至少有一个元素，则唤醒取值

} catch (Exception e) {

e.printStackTrace();

} finally {

lock.unlock();

}

}

public T take() {

T t = null;

try {

lock.lock();

if (count == 0) { // 队列已空，等待加值

System.out.println("目前队列已空，等待入值中");

notEmpty.await();

}

if (head > (nodes.length - 1)) { // 若取值游标大于游标最大值，则从0开始计算

head = 0;

}

t = nodes[head]; // 拿到元素值

nodes[head] = null; // 清空原有位置上的值

head++; // 取值游标+1

count--; // 元素个数-1

notFull.signalAll(); // 走到这里说明队列至少有一个空位，则唤醒入值

} catch (Exception e) {

e.printStackTrace();

} finally {

lock.unlock();

}

return t;

}

}

<https://blog.csdn.net/zxd8080666/article/details/83214089>

## 多线程

### 线程安全

满足以下三个条件，才会出现线程安全问题：**多线程、共享、可变变量**。

### ThreadLocal

ThreadLocal是线程安全的，因为它是线程私有的，不共享的。

ThreadLocal的用途是保存线程上下文信息。比如Spring的事务管理，用ThreadLocal存储Connection，从而各个DAO可以获取同一Connection，可以进行事务回滚，提交等操作。

其实用方法传参也可以实现ThreadLocal的功能，但是有个很大的弊端是需要在很多地方传递这个参数。ThreadLocal避免了参数传递，保证了线程安全。

但是ThreadLocal的线程安全也是有条件的，ThreadLocal里的共享变量必须是一个局部变量或者共享变量本身是线程安全的。

这种将某个对象封闭在一个线程中的技术称为线程封闭。

即使多个线程共享ThreadLocal，也不会出现线程安全问题，示例：

public class ThreadLocalTest {

private static ThreadLocal<Integer> threadLocal = new ThreadLocal<>();

public static void main(String[] args) {

new Thread(() -> {

try {

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

threadLocal.set(i);

System.out.println(Thread.currentThread().getName() + "====" + threadLocal.get());

try {

Thread.sleep(200);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

} finally {

threadLocal.remove();

}

}, "threadLocal1").start();

new Thread(() -> {

try {

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

//threadLocal.set(i);

System.out.println(Thread.currentThread().getName() + "====" + threadLocal.get());

try {

Thread.sleep(200);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

} finally {

threadLocal.remove();

}

}, "threadLocal2").start();

}

}

输出结果如下，threadLocal2这个线程始终拿不到threadLocal1这个线程的上下文信息：

threadLocal1====0

threadLocal2====null

threadLocal2====null

threadLocal1====1

threadLocal2====null

threadLocal1====2

threadLocal1====3

threadLocal2====null

……

threadLocal2====null

threadLocal1====97

threadLocal1====98

threadLocal2====null

threadLocal2====null

threadLocal1====99

因为ThreadLocal会把当前ThreadLocal作为key，上下文信息作为value，保存在ThreadLocalMap里。

ThreadLocal源码get如下：

public void set(T value) {

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

if (map != null)

map.set(this, value);

else

createMap(t, value); //ThreadLocalMap是ThreadLocal的静态内部类。它的初始容量是16。

}

ThreadLocalMap getMap(Thread t) {

return t.threadLocals; //ThreadLocal和Thread都在java.lang包下，而Thread的threadLocals变量是包内可以访问的。

}

public T get() {

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

if (map != null) {

ThreadLocalMap.Entry e = map.getEntry(this);

if (e != null) {

@SuppressWarnings("unchecked")

T result = (T)e.value;

return result;

}

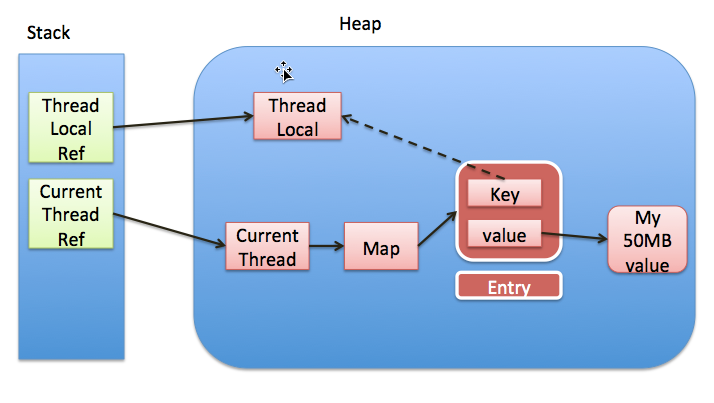
}

return setInitialValue();

}

一个ThreadLocal只能存储一个Object对象，如果需要存储多个Object对象那么就需要多个ThreadLocal。这一点不太友好。

ThreadLocal存在内存泄漏风险。如果这样操作，就可能带来内存泄漏：设置我们的ThreadLocal变量为null，会导致ThreadLocal被jvm当垃圾回收掉，然后我们在ThreadLocalMap里存的value，因为没有ThreadLocal这个key，无法被获取。但是只要Thread不被销毁，Thread的threadLocals就不会被释放，threadLocals里存的value也就没办法释放。



如图，每个thread中都存在一个map, map的类型是ThreadLocal的静态内部类ThreadLocalMap。 Map中的key为一个threadlocal实例。因为存在一条从current thread连接过来的强引用. 只有当前thread结束以后, current thread就不会存在栈中,强引用断开, Current Thread, Map, value将全部被GC回收。

只要这个线程对象被gc回收，就不会出现内存泄露。要命的是线程对象不被回收的情况，这就发生了内存泄露。比如使用线程池的时候，线程结束是不会销毁的，会再次使用。就**可能出现内存泄露**。

Java为了最小化减少内存泄露的可能性和影响，在ThreadLocal的get,set的时候都会清除线程Map里所有key为null的value。所以**最可怕的情况**就是，threadLocal对象设null了，开始发生“内存泄露”，然后使用线程池，这个线程结束，线程放回线程池中不销毁，这个线程一直不被使用，或者分配使用了又不再调用threadLocal的get,set方法，那么这个期间就会发生真正的内存泄露。

为了避免内存泄漏，每次使用完ThreadLocal后，调用其remove方法，清除value。

为什么ThreadLocalMap 中的 key 被设计成弱引用？因为如果key是强引用，当我们想让ThreadLocal被回收时，ThreadLocalMap 还持有 ThreadLocal 的强引用，导致ThreadLocal不能被回收，除非我们手动删除ThreadLocalMap里的ThreadLocal，但这样一来很麻烦，如果不删，又会导致超出我们预期的内存泄漏。如果key是弱引用，即ThreadLocalMap 持有 ThreadLocal 的弱引用，这样ThreadLocal就可以被回收。

DataSourceTransactionManager 是spring的数据源事务管理器， 它会在你调用getConnection()的时候从数据库连接池中获取一个connection， 然后将其与ThreadLocal绑定， 事务完成后解除绑定。这样就保证了事务在同一连接下完成。

所在位置：org.springframework.jdbc.datasource.DataSourceTransactionManager#doBegin

　　　　->TransactionSynchronizationManager#bindResource

　　　　->org.springframework.transaction.support.TransactionSynchronizationManager#bindResource

public static void bindResource(Object key, Object value) throws IllegalStateException {

Object actualKey = TransactionSynchronizationUtils.unwrapResourceIfNecessary(key);

Assert.notNull(value, "Value must not be null");

// ThreadLocal<Map<Obj,Obj>> resources; 从ThreadLocal中拿到value值，即Map对象

　　 Map<Object, Object> map = (Map)resources.get();

if (map == null) {

map = new HashMap();

resources.set(map);

}

// 往Map中赋值，将DBSource与Conn分别作为kv存储

Object oldValue = ((Map)map).put(actualKey, value);

if (oldValue instanceof ResourceHolder && ((ResourceHolder)oldValue).isVoid()) {

oldValue = null;

}

if (oldValue != null) {

throw new IllegalStateException("Already value [" + oldValue + "] for key ["

　　　　　　　　+ actualKey + "] bound to thread [" + Thread.currentThread().getName() + "]");

} else {

if (logger.isTraceEnabled()) {

logger.trace("Bound value [" + value + "] for key [" + actualKey + "] to thread [" + Thread.currentThread().getName() + "]");

}

}

}

### 创建线程

**创建线程的方式一：**继承Thread，重写run方法。

**创建线程的方式二：**实现Runnable接口，实现run方法。然后放到Thread里。

RunnableThreadTest rtt = new RunnableThreadTest(); //实现runnable的类

new Thread(rtt,"新线程1").start(); //启动线程

**创建线程的方式三：**实现Callable接口，实现call方法。使用FutureTask类来包装Callable对象。使用Thread类来包装FutureTask对象。

CallableThreadTest ctt = new CallableThreadTest();

FutureTask<Integer> ft = new FutureTask<>(ctt);

new Thread(ft,"有返回值的线程").start();

System.out.println("子线程的返回值："+ft.get());

三种方式对比：

（1）继承Thread，就不能再继承别的类。实现Runnable接口或Callable接口，还可以继承其他类。

（2）继承Thread，编写简单。实现Runnable接口或Callable接口，编写复杂，但多个线程可以共享同一个Runnable对象，适合多个相同线程来处理同一份资源的情况。

## 泛型

那有没有一种办法在编译阶段，即能合并成同一个，又能在编译时检查出来传进去类型不对呢？当然，这就是泛型。

class Point<T>{// 此处可以随便写标识符号

private T x ;

private T y ;

public void setX(T x){//作为参数

this.x = x ;

}

public T getX(){//作为返回值

return this.x ;

}

};

//IntegerPoint使用

Point<Integer> p = new Point<Integer>() ;

p.setX(new Integer(100)) ;

System.out.println(p.getX());

//FloatPoint使用

Point<Float> p = new Point<Float>() ;

这样setX的时候就能帮忙检查是不是我们定义的类型了。

## final

final修饰的变量一旦初始化，就不能被修改。如果是类变量，只能在构造方法中初始化。写在构造方法外的成员变量初始化，等价于写在构造方法内紧跟super()后的初始化。

如果final修饰的变量是一个对象，比如ArrayList，或者数组类型，比如int[]，那么虽然不可以修改final修饰的对象，但是可以更改对象里的属性。

这样是可以的，示例：

public class Son {

public final int[] names;

public Son() {

names = new int[10];

names[0] = 1;

}

public static void main(String[] args) {

Son son = new Son();

son.names[1] = 2;

System.out.println(son.names[1]); //输出2

}

}

## 注解

Java 注解（Annotation）又称 Java 标注，是 JDK5.0 引入的一种注释机制。

和 Javadoc 不同，Java 标注可以通过反射获取标注内容。在编译器生成类文件时，标注可以被嵌入到字节码中。Java 虚拟机可以保留标注内容，在运行时可以获取到标注内容 。

Javadoc示例，里面的@XXX就是Javadoc的写法，参考：<https://blog.csdn.net/vbirdbest/article/details/80296136>

package com.example.demo;

/\*\*

\* 类 {@code OrderService} 订单服务层.

\*

\* <p> 主要包括 创建订单、取消订单、查询订单等功能更

\*

\* @see Order

\* @author <a href="mailto:mengday.zhang@gmail.com">Mengday Zhang</a>

\* @since 2018/5/12

\*/

public class OrderService {

/\*\* 默认数量 {@value} \*/

private static final Integer QUANTITY = 1;

/\*\*

\* 创建订单.

\*

\* <p> 创建订单需要传用户id和商品列表(商品id和商品数量).

\*

\* <p><pre>{@code

\* 演示如何使用该方法

\* List<Goods> items = new ArrayList<>();

\* Goods goods = new Goods(1L, BigDecimal.ONE);

\* Goods goods2 = new Goods(2L, BigDecimal.TEN);

\* items.add(goods);

\* items.add(goods2);

\*

\* Order order1 = new Order();

\* order.setUserId("1");

\* order.setItems(items);

\* OrderService#createOrder(order);

\* }

\* </pre>

\*

\* @param order 订单信息

\* @throws NullPointerException 参数信息为空

\* @exception IllegalArgumentException 数量不合法

\* @return 是否创建成功

\* @version 1.0

\* @see {@link Order}

\*/

public boolean createOrder(Order order) throws IllegalArgumentException{

Objects.requireNonNull(order);

List<Goods> items = order.getItems();

items.forEach(goods -> {

BigDecimal quantity = goods.getQuantity();

if (quantity == null || BigDecimal.ZERO.compareTo(quantity) == 0) {

throw new IllegalArgumentException();

}

});

System.out.println("create order...");

return true;

}

}

下面来看Annotation的定义：

Annotation：

package java.lang.annotation;

public interface Annotation {

boolean equals(Object obj);

int hashCode();

String toString();

Class<? extends Annotation> annotationType();

}

ElementType：

package java.lang.annotation;

public enum ElementType {

/\*\* Class, interface (including annotation type), or enum declaration \*/

TYPE,

/\*\* Field declaration (includes enum constants) \*/

FIELD,

/\*\* Method declaration \*/

METHOD,

/\*\* Formal parameter declaration \*/

PARAMETER,

/\*\* Constructor declaration \*/

CONSTRUCTOR,

/\*\* Local variable declaration \*/

LOCAL\_VARIABLE,

/\*\* Annotation type declaration \*/

ANNOTATION\_TYPE,

/\*\* Package declaration \*/

PACKAGE,

/\*\*Type parameter declaration\*/

TYPE\_PARAMETER,

/\*\*Use of a type\*/

TYPE\_USE

}

RetentionPolicy：

package java.lang.annotation;

public enum RetentionPolicy {

SOURCE, /\* Annotation信息仅存在于编译器处理期间，编译器处理完之后就没有该Annotation信息了。例如，" @Override" 标志就是一个 Annotation。当它修饰一个方法的时候，就意味着该方法覆盖父类的方法；并且在编译期间会进行语法检查！编译器处理完后，"@Override" 就没有任何作用了。\*/

**CLASS**, /\* 编译器将Annotation存储于类对应的.class文件中。**默认行为** \*/

RUNTIME /\* 编译器将Annotation存储于class文件中，并且可由JVM读入 \*/

}

例如定义一个自己的Annotation：

@Documented

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

public @interface MyAnnotation1 {

}

（1）其中@interface表示把MyAnnotation1作为注解。通过 @interface 定义注解后，该注解不能继承其他的注解或接口。

（2）类和方法的 Annotation 在缺省情况下是不出现在 javadoc 中的。如果使用 @Documented 修饰该 Annotation，则表示它可以出现在 javadoc 中。

（3）@Target(ElementType.TYPE) 的意思就是指定该 Annotation 的类型是 ElementType.TYPE。这就意味着，MyAnnotation1 是来修饰"类、接口（包括注释类型）或枚举声明"的注解。

定义 Annotation 时，@Target 可有可无。若有 @Target，则该 Annotation 只能用于它所指定的地方；若没有 @Target，则该 Annotation 可以用于任何地方。

例如Deprecated：

package java.lang.annotation;

@Documented

@Retention(RetentionPolicy.RUNTIME)

public @interface Deprecated {

}

指定 Deprecated 的策略是 RetentionPolicy.RUNTIME。这就意味着，编译器会将Deprecated 的信息保留在 .class 文件中，并且能被虚拟机读取。

如果有开发人员试图使用或重写被 @Deprecated 标示的方法，编译器会给相应的提示信息。

例如Inherited：

package java.lang.annotation;

@Documented

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.ANNOTATION\_TYPE)

public @interface Inherited {

}

@Target(ElementType.ANNOTATION\_TYPE)的作用是指定 Inherited 的类型是 ANNOTATION\_TYPE，意味着@Inherited 只能被用来标注 "Annotation 类型"。

@Inherited 的含义是，它所标注的Annotation将具有继承性。

例如，我们定义了某个 Annotaion，它的名称是 MyAnnotation，并且 MyAnnotation 被标注为 @Inherited。现在，某个类 Base 使用了MyAnnotation，则 Base 具有了注解 MyAnnotation；现在，Sub 继承了 Base，由于 MyAnnotation 是 @Inherited的(具有继承性)，所以，Sub也具有了注解 MyAnnotation。

示例：

import java.lang.annotation.Target;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Inherited;

/\*\*

\* 自定义的Annotation。

\*/

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Inherited

@interface Inheritable

{

}

@Inheritable

class InheritableFather

{

public InheritableFather() {

// InheritableBase是否具有 Inheritable Annotation

System.out.println("InheritableFather:"+InheritableFather.class.isAnnotationPresent(Inheritable.class));

}

}

/\*\*

\* InheritableSon 类只是继承于 InheritableFather，

\*/

public class InheritableSon extends InheritableFather

{

public InheritableSon() {

super(); // 调用父类的构造函数

// InheritableSon类是否具有 Inheritable Annotation

System.out.println("InheritableSon:"+InheritableSon.class.isAnnotationPresent(Inheritable.class));

}

public static void main(String[] args)

{

InheritableSon is = new InheritableSon();

}

}

运行结果是：

InheritableFather:true

InheritableSon:true

上面的代码如果去掉@Inherited，则运行结果是：

InheritableFather:true

InheritableSon:false

例如SuppressWarnings：

package java.lang.annotation;

@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL\_VARIABLE})

@Retention(RetentionPolicy.SOURCE)

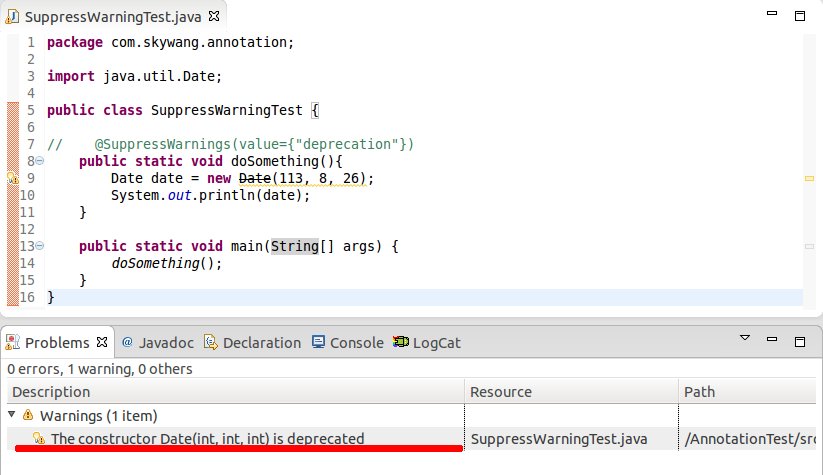
public @interface SuppressWarnings {

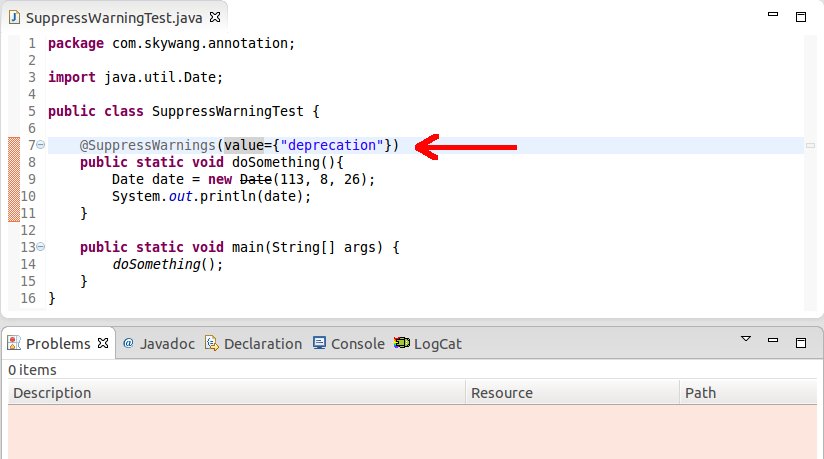
String[] value();

}

@Retention(RetentionPolicy.SOURCE)指定 SuppressWarnings 的策略是 RetentionPolicy.SOURCE，意味着SuppressWarnings 信息仅存在于编译器处理期间，编译器处理完之后 SuppressWarnings 就没有作用了。

@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL\_VARIABLE})指定 SuppressWarnings 的类型同时包括TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL\_VARIABLE。





Annotation 的作用：

（1）例如，@SuppressWarnings, @Deprecated 和 @Override 都具有编译检查作用。例如如果有方法被 @Override 标示，但父类中却没有"被 @Override 标注"的同名方法，则编译器会报错。

（2）在反射中使用Annotation，减少代码量，例如springboot广泛使用Annotation

示例：

import java.lang.annotation.Annotation;

import java.lang.annotation.Target;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Inherited;

import java.lang.reflect.Method;

/\*\*

\* Annotation在反射函数中的使用示例

\*/

@Retention(RetentionPolicy.RUNTIME)

@interface MyAnnotation {

String[] value() default "unknown";

}

/\*\*

\* Person类。它会使用MyAnnotation注解。

\*/

class Person {

/\*\*

\* empty()方法同时被 "@Deprecated" 和 "@MyAnnotation(value={"a","b"})"所标注

\* (01) @Deprecated，意味着empty()方法，不再被建议使用

\* (02) @MyAnnotation, 意味着empty() 方法对应的MyAnnotation的value值是默认值"unknown"

\*/

@MyAnnotation

@Deprecated

public void empty(){

System.out.println("\nempty");

}

/\*\*

\* sombody() 被 @MyAnnotation(value={"girl","boy"}) 所标注，

\* @MyAnnotation(value={"girl","boy"}), 意味着MyAnnotation的value值是{"girl","boy"}

\*/

@MyAnnotation(value={"girl","boy"})

public void somebody(String name, int age){

System.out.println("\nsomebody: "+name+", "+age);

}

}

public class AnnotationTest {

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

// 新建Person

Person person = new Person();

// 获取Person的Class实例

Class<Person> c = Person.class;

// 获取 somebody() 方法的Method实例

Method mSomebody = c.getMethod("somebody", new Class[]{String.class, int.class});

// 执行该方法

mSomebody.invoke(person, new Object[]{"lily", 18});

iteratorAnnotations(mSomebody);

// 获取 somebody() 方法的Method实例

Method mEmpty = c.getMethod("empty", new Class[]{});

// 执行该方法

mEmpty.invoke(person, new Object[]{});

iteratorAnnotations(mEmpty);

}

public static void iteratorAnnotations(Method method) {

// 判断 somebody() 方法是否包含MyAnnotation注解

if(method.isAnnotationPresent(MyAnnotation.class)){

// 获取该方法的MyAnnotation注解实例

MyAnnotation myAnnotation = method.getAnnotation(MyAnnotation.class);

// 获取 myAnnotation的值，并打印出来

String[] values = myAnnotation.value();

for (String str:values)

System.out.printf(str+", ");

System.out.println();

}

// 获取方法上的所有注解，并打印出来

Annotation[] annotations = method.getAnnotations();

for(Annotation annotation : annotations){

System.out.println(annotation);

}

}

}

运行结果：

somebody: lily, 18

girl, boy,

@com.skywang.annotation.MyAnnotation(value=[girl, boy])

empty

unknown,

@com.skywang.annotation.MyAnnotation(value=[unknown])

@java.lang.Deprecated()

## Object

继承Object是由JVM处理的：

编译器仍然按照实际代码进行编译，并不会做额外的处理，即如果一个类没有显式地继承于其他类时，编译后的代码仍然没有父类。然后由虚拟机运行二进制代码时，当遇到没有父类的类时，就会自动将这个类看成是Object类的子类（一般这类语言的默认父类都是Object）

我们在编辑器里（IDE）打点时就能列出Object类下的方法，此时还没轮到编译器和jvm，编辑器就已经知道MyClass类的父类是Object类了，这是因为编辑器为我们做了一些智能处理。

## 移位

由于double，float在二进制中的表现比较特殊，因此不能来进行移位操作。

移位数量大于64取余，不会让数的比特位变成全0的，没意义

左移运算符：<<

右补零，不考虑符号位，可能变负数

右移运算符：>>

对于正数左补0，对于负数左补1

无符号右移运算符：>>>

左补零，不考虑符号位，可能变正数

<https://zhuanlan.zhihu.com/p/30108890>

## HashMap

默认容量16，负载因子0.75，使用链地址法解决哈希冲突。使用自己的哈希函数+数组长度掩码。

自己的哈希函数就是系统的hash值异或系统的hash值右移16位。

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

Java 1.8版本，将hashmap解决hash冲突的方式由链表法改为链表法+红黑树法。

当链表长度大于等于8时，转换成红黑树结构；当红黑树长度小于等于6时，转换为链表。为什么不让红黑树长度小于等于7时就转换为链表呢？因为要避免乒乓效应，以免链表或红黑树长度总是在7和8之间来回变，程序频繁地做链表、红黑树的转换工作。

<https://blog.csdn.net/f1004145107/article/details/105904975/>

<https://blog.csdn.net/longfulong/article/details/78760486>

## ArrayList

ArrayList的默认初始容量是0。在第一次添加元素的时候，会对集合进行扩容，扩容之后，集合容量为10；之后，当向集合中添加元素达到集合的上限(也就是minCapacity大于elementData.length)时，会对集合再次扩容，扩容为原来的3/2，也就是15，再扩容一次，就是22了。

如果初始值设为12，扩容一次是18。

**ArrayList是线程不安全的。如果想保证线程安全，方法是：**

（1）使用Collections.synchronizedList(List)进行包装，其内部有Object mutex = new Object()，并对list的各个方法加synchronized (mutex) {。。。}包装

（2）使用Vector，Vector是线程安全的。将public static List arrayList = new ArrayList();替换为public static List arrayList = new Vector<>();

为什么Vector是线程安全的？因为Vector几乎每个方法都用synchronized修饰，所以线程安全。

（3）使用CopyOnWriteArrayList，这个不详细讲了，有点复杂。

<https://www.cnblogs.com/IT-CPC/p/10897559.html>

## socket

Socket是什么？

socket是应用层与传输层的一个抽象，将复杂的TCP/IP协议隐藏在Socket接口之后，只对应用层暴露简单的接口。

socket是一种特殊的文件，它也有文件描述符，进程可以打开一个socket，并且像处理文件一样对它进行read()和write()操作，而不必关心数据是怎么在网络上传输的。

socket是一个tcp连接的两端。

Socket属于网络的哪一层？

Socket不算是一个协议，它是应用层与传输层间的一个抽象层。它把TCP/IP层复杂的操作抽象为几个简单的接口供应用层调用，以实现进程在网络中通信。

Socket如何唯一标识一个进程？

socket基于tcp协议实现，网络层的ip地址唯一标识一台主机。

Socket是全双工的吗？

基于TCP协议，是全双工的。

通信双方如何进行端口绑定？

通常服务端启动时会绑定一个端口提供服务，而客户端在发起连接请求时会被随机分配一个端口号。（建立TCP连接时，客户端随机端口）。

## 变量命名、参数

如果传递的参数为基本数据类型，参数视为形参。形参的改变不会影响实参的值。

如果传递的参数是引用数据类型，参数视为实参。修改了引用的数据，就会影响实参的引用值。如果只是修改这个引用，不修改引用的数据，就不会影响引用的数据。

变量命名：

对于类的成员变量，类的方法内部可以创建和成员变量一样名字的变量。如果想引用成员变量，前面加上this.就可以了。

对于方法传入的参数变量，方法内部不可以创建与传入的参数变量同样名字的变量，编译器会报错。

## String、StringBuffer、StringBuilder

StringBuffer是线程安全的，因为所有方法都用synchronized修饰了。

StringBuilder与StringBuffer的区别就是所有方法都没有synchronized。

String是不可变对象。

所以修改时，速度StringBuilder > StringBuffer > String

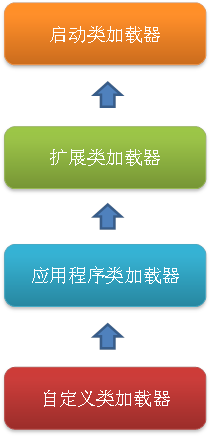
## 类加载器

类加载器使用了双亲委派机制。什么是双亲委派机制？就是当某个类加载器需要加载某个.class文件时，它首先把这个任务委托给他的上级类加载器，递归这个操作，如果上级的类加载器没有加载，自己才会去加载这个类。类加载器的父子关系并不是通过继承关系来实现的，而是使用组合关系来复用父加载器中的代码，源码中就是构造子类加载器时需要传入父类加载器对象。

为什么使用双亲委派机制？因为这样就具备了带有优先级的层次关系，保证了Java程序的稳定运作。例如，类java.lang.Object类存放在JDK\jre\lib下的rt.jar之中，因此无论是哪个类加载器要加载此类，最终都会委派给启动类加载器进行加载，这边保证了Object类在程序中的各种类加载器中都是同一个类，否则可能不是同一个类。双亲委派机制被广泛应用于各种程序中。

<https://blog.csdn.net/ns_code/article/details/17881581>

过程如下图：

其中启动类加载器（BootStrapClassLoader），由c++编写，加载java核心库 java.\*，构造ExtClassLoader和AppClassLoader。由于引导类加载器涉及到虚拟机本地实现细节，开发者无法直接获取到启动类加载器的引用。

扩展类加载器（ExtClassLoader），由java编写，加载扩展库，如classpath中的jre ，javax.\*或者java.ext.dir 指定位置中的类。由sun.misc.Launcher$ExtClassLoader实现，它负责加载JDK\jre\lib\ext目录中，或者由java.ext.dirs系统变量指定的路径中的所有类库（如javax.\*开头的类）。开发者可以直接使用标准扩展类加载器。

应用程序类加载器（AppClassLoader），由java编写，加载程序所在的目录，如user.dir所在位置的class。

自定义类加载器（CustomClassLoader），java编写，用户自定义的类加载器，可加载指定路径的class文件。JVM自带的ClassLoader只是懂得从本地文件系统加载标准的java class文件，如果编写了自己的ClassLoader，便可以做到如下几点：（1）动态地创建符合用户特定需要的定制化构建类，（2）从特定的场所取得java class，例如数据库中和网络中

启动类加载器：它使用C++实现（这里仅限于Hotspot，也就是JDK1.5之后默认的虚拟机，有很多其他的虚拟机是用Java语言实现的），是虚拟机自身的一部分。

所有其他的类加载器：这些类加载器都由Java语言实现，独立于虚拟机之外，并且全部继承自抽象类java.lang.ClassLoader，这些类加载器需要由启动类加载器加载到内存中之后才能去加载其他的类。

来看一下源码：

package java.lang;

……

public abstract class ClassLoader {

……

public Class<?> loadClass(String name) throws ClassNotFoundException {

return loadClass(name, false);

}

……

protected Class<?> loadClass(String name, boolean resolve)

throws ClassNotFoundException

{

//对这个类名进行加锁，避免其他线程重复加载这个类

synchronized (getClassLoadingLock(name)) {

// First, check if the class has already been loaded。首先，检查类是否加载过，加载过直接给结果，没加载过现在加载

Class<?> c = findLoadedClass(name);

if (c == null) {

long t0 = System.nanoTime();

try {

//如果有父类加载器，让父类加载器加载

if (parent != null) {

c = parent.loadClass(name, false);

} else {

//如果没有父类加载器，则直接递归到bootStrapClassloader，

//因为bootStrapClassloader比较特殊，是native方法，无法通过get获取，所以这里调用findBootstrapClassOrNull，交给BootStrapClassLoader去检查类是否加载过，如果加载过给结果，没加载过尝试是否可以加载并返回结果

c = findBootstrapClassOrNull(name);

}

} catch (ClassNotFoundException e) {

// ClassNotFoundException thrown if class not found

// from the non-null parent class loader

}

//如果父类加载器、BootstrapClassLoader 都不能加载此类，则递归回来，尝试自己去加载class

if (c == null) {

// If still not found, then invoke findClass in order

// to find the class.

long t1 = System.nanoTime();

c = findClass(name);

// this is the defining class loader; record the stats

sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);

sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);

sun.misc.PerfCounter.getFindClasses().increment();

}

}

if (resolve) {

resolveClass(c);

}

return c;

}

}

……

}

类加载器关系着类初始化的过程，一般如下：



打个简单的比方，你一个WEB程序，发布到Tomcat里面运行。首先是执行Tomcat org.apache.catalina.startup.Bootstrap类，这时候的类加载器是ClassLoader.getSystemClassLoader()，即BootStrapClassLoader。而我们后面的WEB程序，里面的jar、resources都是由Tomcat内部来加载的，即AppClassLoader，所以你在代码中动态加载jar、资源文件的时候，首先应该是使用Thread.currentThread().getContextClassLoader()。如果你使用Test.class.getClassLoader()，可能会得到BootStrapClassLoader，导致和当前线程所运行的类加载器不一致（因为Java天生的多线程）。

Test.class.getClassLoader()一般用在getResource，因为你想要获取某个资源文件的时候，这个资源文件的位置是相对固定的。

建议去查看一下openfire或者tomcat内部的源码，对ClassLoader会有比较深的理解。

用例子验证上述内容：

@RestController

@RequestMapping(value = "getResource")

public class GetResourceController {

@GetMapping(value = "get")

public ResponseEntity<String> getResource() {

String resourcePath = "lisp/helloworld.lisp";

URL systemResource = ClassLoader.getSystemResource(resourcePath);

System.out.println(">>>" + systemResource);

ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();

System.out.println(">>>>>" + systemClassLoader.toString());

System.out.println(">>>" + systemClassLoader.getResource(resourcePath));

ClassLoader cuurentThreadClassLoader = Thread.currentThread().getContextClassLoader();

System.out.println(">>>>>" + cuurentThreadClassLoader.toString());

System.out.println(">>>" + cuurentThreadClassLoader.getResource(resourcePath));

ClassLoader parentClassLoader = cuurentThreadClassLoader.getParent();

System.out.println("parent:" + parentClassLoader);

ClassLoader thisClassLoader = GetResourceController.class.getClassLoader();

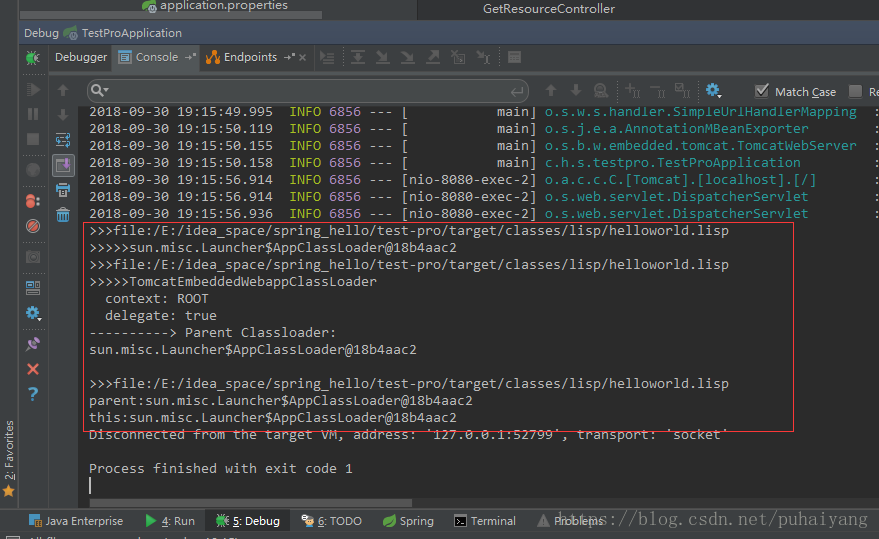
System.out.println("this:" + thisClassLoader);

return new ResponseEntity<String>("ok", HttpStatus.OK);

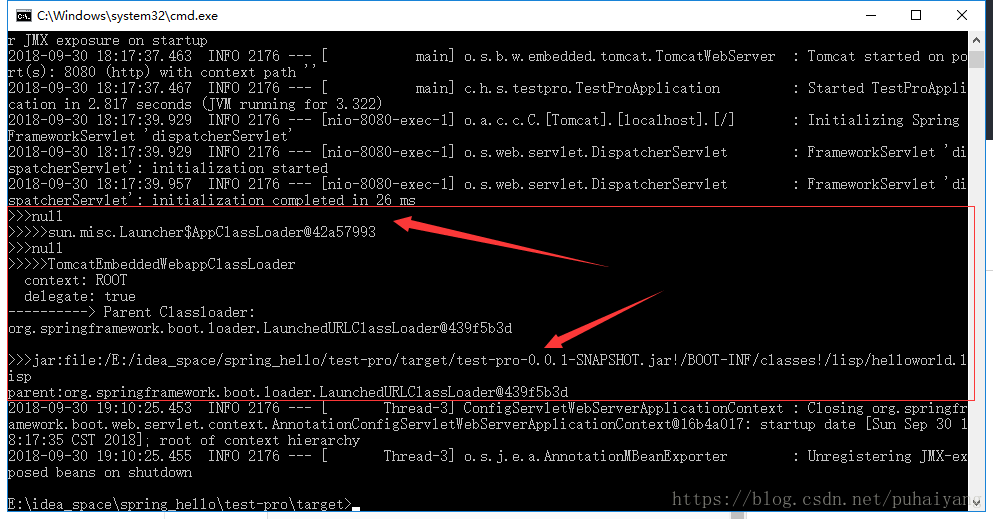
}

}

上述代码在IDEA中运行的结果为：



如果打包成fat JAR，放到Tomcat中运行，结果为：



导致上述结果的原因是：在IDEA开发工具中运行时调用classloader调用的是AppClassLoader，在IDEA环境时可以加载到target目录下的所有文件。而用fat jar方式运行时用的是内置tomcat的classloader去加载的，即上图中的LaunchedURLClassLoader。

如果不想出现上述的classLoader不一致的情况，导致bug，则调用静态资源的classLoader最好用Thread.currentThread().getContextClassLoader()方法来获取。

## 类初始化

静态代码块和静态成员的执行顺序由书写顺序决定，均在main方法前执行。

例如：

package xxx;

public class Son {

static {

System.out.println(1111);

}

static int i = prt();

static int prt(){

System.out.println(222);

return 222;

}

public static void main(String[] args) {

System.out.println(333);

Son son = new Son();

}

}

输出结果是：

1111

222

333

对调静态代码块和静态成员变量：

package xxx;

public class Son {

static int i = prt();

static {

System.out.println(1111);

}

static int prt(){

System.out.println(222);

return 222;

}

public static void main(String[] args) {

System.out.println(333);

Son son = new Son();

}

}

输出结果是：

222

1111

333

如果类继承了别的类，那么初始化的时候，会先初始化父类的静态内容，然后初始化子类的静态内容，然后初始化父类的非静态成员变量，再调用父类的构造函数。调用完父类的构造函数后，回来初始化本类的非静态成员变量，再调用本类的构造函数。这个过程中用到的自定义方法，对于非静态方法，如果被子类覆盖了，就调用子类的方法，对于静态方法，是不能被覆盖的，因为静态方法属于类，不属于对象，所以会调用自己的静态方法。

示例如下：

package xx;

class Father {

int i = prt();

Father() {

System.out.println("Father construc");

say();

}

public static int prt() {

System.out.println("Father int i");

return 10;

}

public void say(){

System.out.println("Father say");

}

}

public class Son extends Father {

int j = prt();

Son() {

System.out.println("Son construct");

say();

}

public static int prt(){

System.out.println("Son int i");

return 20;

}

public void say(){

System.out.println("Son say = " + j);

}

public static void main(String[] args) {

Son son = new Son();

}

}

输出结果是：

Father int i

Father construc

Son say = 0

Son int i

Son construct

Son say = 20

再举一个例子：

public class Main {

public static Star star = new Star();

public int a = 0;

public static void main(String[] args) {

System.out.println(“A”);

new Main();

new Main();

new Star();

}

public Main() {

System.out.println(“B”);

}

{

System.out.println(“C”);

}

static {

System.out.println(“D”);

}

}

public class Star {

static {

System.out.println(“E”);

}

public Star() {

System.out.println(“F”);

}

{

System.out.println(“G”);

}

}

输出结果是：

EGFDACBCBGF

**为什么呢？**因为JVM首先会加载含有main方法的Main.class，完成其静态成员变量初始化、静态代码块执行。Main.class里的静态成员变量需要实例化Star，因此先加载Star.class，完成Star.class的静态成员变量初始化、静态代码块执行，输出E。然后实例化Star，执行Star的代码块，输出G；然后执行Star的构造方法，输出F。然后再回来执行Main.class的静态代码块，输出D。Main.class加载完后，就开始执行其main方法了，输出A。然后new Main就是实例化Main，执行Main的代码块，输出C；然后执行Main的构造方法，输出B。下一个new Main仍旧是实例化Main，执行Main的代码块，输出C；然后执行Main的构造方法，输出B。最后new Star就是实例化Star，执行Star的代码块，输出G；然后执行Star的构造方法，输出F。

**综上所述，只有在类加载的时候，执行类的静态成员变量初始化、静态代码块执行，因为类只加载一次，所以只执行一次。每次实例化类的时候，都会执行一次类的代码块，执行一次构造方法。**

优先级：静态成员变量、静态代码块 > main方法 > 成员变量、代码块 > 构造方法

## 类加载机制

虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型，这就是虚拟机的类加载机制。

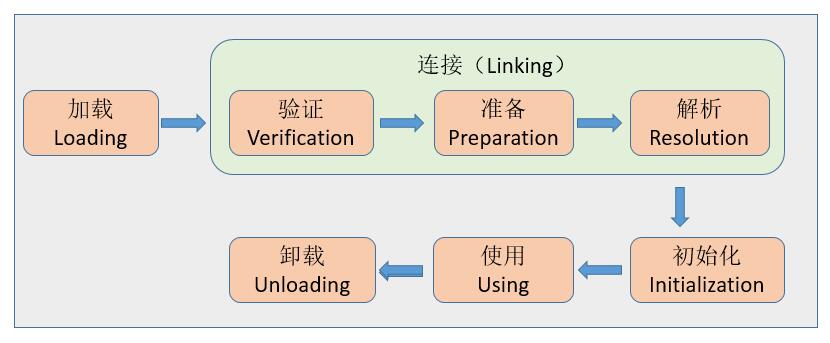
在Java语言里，类型的加载、连接和初始化过程都是在程序**运行期间**完成的，例如import java.util.\*下面包含很多类，但是，在程序运行的时候，虚拟机**只会加载那些我们程序需要的类**。这种策略虽然会令类加载时稍微增加一些性能开销，但是会为Java应用程序提供高度的**灵活性**。

有些语言是在**编译期**完成类的加载。

**静态绑定：**即前期绑定。在程序执行前方法已经被绑定，此时由编译器或其它连接程序实现。针对java，简单的可以理解为程序编译期的绑定。java当中的方法只有final，static，private和构造方法是前期绑定的。

**动态绑定：**即晚期绑定，也叫运行时绑定。在运行时根据具体对象的类型进行绑定。在java中，几乎所有的方法都是后期绑定的。

类加载的过程，其中**解析有可能在初始化之后，因为这样设计可以实现动态绑定**：



### 加载

类的加载是在程序运行期完成的，那什么时候加载类？当然是在类被引用的时候加载类。这里就分主动引用、被动引用。

**主动引用：**

（1）程序启动，执行main方法的时候，触发main方法所在类的初始化。

（2）初始化类的时候，如果发现其父类没有被初始化，先触发其父类的初始化。

（3）遇到new（使用new 关键字实例化一个对象）、get static（读取一个类的静态字段）、put static或者invoke static（设置一个类的静态字段）这4条指令的时候，如果类没有进行过初始化。则需要先触发其初始化。

（4）使用反射进行反射调用的时候，如果类没有初始化，则需要先触发其初始化。

**被动引用：**

（1）通过子类引用父类的静态字段，不会初始化子类。因为静态字段属于父类，只有直接定义静态字段的类才会被触发初始化。例子：

//父类

public class SuperClass {

//静态变量value

public static int value = 666;

//静态块，父类初始化时会调用

static{

System.out.println("父类初始化！");

}

}

//子类

public class SubClass extends SuperClass{

//静态块，子类初始化时会调用

static{

System.out.println("子类初始化！");

}

}

//主类、测试类

public class NotInit {

public static void main(String[] args){

System.out.println(SubClass.value);

}

}

运行结果：

父类初始化！

666

（2）通过类引用常量，不会初始化类。因为常量在编译阶段会存入调用常量的类的常量池中，本质上并没有引用定义这个常量的类。例子：

//常量类

public class ConstClass {

static{

System.out.println("常量类初始化！");

}

public static final String HELLOWORLD = "hello world!";

}

//主类、测试类

public class NotInit {

public static void main(String[] args){

System.out.println(ConstClass.HELLOWORLD);

}

}

运行结果：

Hello world！

（3）通过数组来引用类，不会初始化类，因为是数组new，而类没有被new。例子：

//父类

public class SuperClass {

//静态变量value

public static int value = 666;

//静态块，父类初始化时会调用

static{

System.out.println("父类初始化！");

}

}

//主类、测试类

public class NotInit {

public static void main(String[] args){

SuperClass[] test = new SuperClass[10];

}

}

运行结果：

无任何输出

**在加载阶段虚拟机需要完成以下三件事：**

（1）通过一个类的全限定名称来获取此类的二进制字节流

（2）将这个字节流所代表的静态存储结构转化为**方法区**的运行时**数据结构**

（3）在**Java堆**中生成一个代表这个类的java.lang.Class对象，作为**方法区**这个类的各种**数据的访问入口**

通常来讲，一个类的全限定名称可以从zip、jar包中加载，也可以从网络中获取，也可以在运行的时候生成（这点最明显的技术体现就是反射机制）。

对于类的加载，可以分为数组类型和非数组类型，对于非数组类型可以通过系统的BootStrapClassLoader进行加载，也可以通过自定义的类加载器进行加载。这点是比较灵活的。而对于数组类型，数组类本身不通过类加载器进行加载，而是通过Java**虚拟机直接进行加载**的。

### 验证

验证阶段的目的是为了确保Class字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机的安全。

虚拟机的验证阶段主要完后以下4项验证：**文件格式验证、元数据验证、字节码验证、符号引用验证。**

**文件格式验证：**验证class文件是否符合编写规范。

**元数据验证：**保证不存在不符合Java语言规范（Java语法）的元数据信息。比如这个类是否继承了不允许被继承的类（比如被final修饰的类）。比如类是否覆盖了不该覆盖的字段、方法（比如是否覆盖了父类的final字段）。比如非抽象类，是否实现了父类的接口、抽象方法。

**字节码验证：**这个阶段主要对类的方法体进行校验分析。

**符号引用验证：**符号引用验证主要是对类自身以外的信息进行匹配性校验。比如通过字符串描述的全限定名是否能够找到对应的类。

**在解析阶段，将符号引用转为直接引用。**

如果无法通过符号引用验证那么将会抛出java.lang.IncomingChangeError异常的子类。

**符号引用（Symbolic Reference） ：**

符号引用以一组符号来描述所引用的目标，符号引用可以是任何形式的字面量，只要使用时能无歧义的定位到目标即可（符号字面量，还没有涉及到内存）。符号引用与虚拟机实现的内存布局无关，**引用的目标并不一定已经加载在内存中。各种虚拟机实现的内存布局可以各不相同，但是他们能接受的符号引用必须都是一致的**，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中。

**直接引用（Direct Reference） ：**

直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄（可以理解为内存地址）。直接引用是与虚拟机实现的内存布局相关的，同一个符号引用在不同的虚拟机实例上翻译出来的直接引用一般都不相同，**如果有了直接引用，那引用的目标必定已经在内存中存在**。

### 准备

准备阶段是正式为变量分配内存空间并且设置类变量初始值。内存是在**方法区**中分配的。

需要注意的是，这时候进行内存分配的仅仅是类变量（也就是被static修饰的变量），实例变量是不包括的，实例变量的初始化是在对象实例化的时候进行初始化，而且分配的内存区域是Java堆。这里的初始值也就是在编程中默认值，也就是零值。

例如public static int value = 3 ；value在准备阶段后的初始值是0而不是3，因为此时尚未执行任何的Java方法，而把value赋值为3的putStatic指令是程序被编译后，存放在类构造器clinit()方法之中，把value赋值为3的动作将在初始化阶段才会执行。

**特殊情况：**如果类字段的字段属性表中存在ConstantValue属性，即同时被final和static修饰，那在准备阶段变量就会被初始化为ConstantValue属性所指定的值，例如public static final int value = 123 编译时javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将变量赋值为123。

### 解析

解析阶段是将常量池中的符号引用替换为直接引用的过程。

不同虚拟机实现可以根据需要判断到底是在类被加载器加载的时候对常量池的符号引用进行解析（也就是初始化之前），还是等到一个符号引用被使用之前进行解析（也就是在初始化之后）。虚拟机可以对第一次解析的结果进行缓存（在运行时常量池中记录引用，并把常量标识为一解析状态），这样就避免了一个符号引用的多次解析。

解析动作主要针对**类或接口**、**字段**、**类方法**、**接口方法**四类符号引用进行，分别对应于常量池中的CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info、CONSTANT\_Methodref\_info、CONSTANT\_InterfaceMethodref\_info四种常量类型。

**（1）类或者接口解析：**

把一个类或者接口的符号引用解析为直接引用，需要三个步骤：

A、如果符号引用不是一个数组类型，那么虚拟机会把该符号引用代表的全限定类名称传递给调用这个符号引用的类。这个过程涉及验证过程，可能会触发其他相关类的加载。

B、如果符号引用是一个数组类型，并且该数组的元素类型是对象。我们知道符号引用是存在方法区的常量池中的，该符号引用的描述符会类似”[java/lang/Integer”的形式（具体可以看下面的链接），将会按照上面的规则进行加载，虚拟机将会生成一个代表此数组对象的直接引用。

<https://blog.csdn.net/u014296316/article/details/83066436>

C、如果上面的步骤都没有出现异常，那么该符号引用已经在虚拟机中产生了一个直接引用。但是在解析完成之前需要对符号引用进行验证，主要是确认当前调用这个符号引用的类是否具有访问权限，如果没有访问权限将抛出java.lang.IllegalAccess异常

**（2）字段解析：**

字段解析需要首先对其所属的类进行解析，因为字段是属于类的，只有在正确解析得到其类的直接引用才能继续字段解析。对字段的解析主要包括以下几个步骤：

A、对字段进行解析时，先在本类中查找是否包含有简单名称和字段描述符都与目标相匹配的字段，如果有，则查找结束。

B、如果没有，则会按照继承关系从上往下递归搜索该类所实现的各个接口和它们的父接口，还没有，则按照继承关系从上往下递归搜索其父类，直至查找结束。**记住是先找接口，再找父类。**

C、上面都没有，则解析失败，抛出java.lang.NoSuchFieldError异常。如果最终返回了这个字段的直接引用，就进行权限验证，如果发现不具备对字段的访问权限，将抛出java.lang.IllegalAccessError异常。在实际应用中，虚拟机的编译器实现可能要比上述规范要求的更严格一些。如果有一个同名字段同时出现在该类的接口和父类中，或同时在自己或父类的接口中出现，编译器可能会拒绝编译。

**（3）类方法解析：**

类方法和接口方法的符号引用是分开的。对类方法的解析与对字段解析的搜索步骤差不多，不同之处在于多了判断该方法所处的是类还是接口的步骤，以及是**先找父类，再找父接口**。如果类方法所处的是接口，会抛出java.lang.IncompatibleClassChangeError异常。如果最终返回了直接引用，还需要对该符号引用进行权限验证，如果没有访问权限，就抛出java.lang.IllegalAccessError异常。

**（4）接口方法解析：**

与类方法解析步骤类似，只是接口不会有父类，所以只找父接口就行了。接口的所有方法都是public，所以不存在访问权限问题。

### 初始化

初始化阶段会执行clinit方法。

clinit方法是编译器自动收集类中所有类变量的赋值动作和静态语句块合并生成的。

编译器收集的顺序是由语句在源文件中出现的顺序决定的。静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值，但是不能访问。示例：

public class Test {

static{

i =0; //给变量赋值可以正常编译通过

System.out.println(i); //这句编译器会提示“非法向前引用”。注意是编译期，编译成.class文件的时候。

}

static int i = 1;

}

虚拟机会保证在子类的clinit() 方法执行之前，父类的clinit() 方法已经执行完毕。而构造器方法（init方法）不同，构造器需要显式调用父类构造器方法，才能执行父类构造器方法。

clinit()方法对于类或者接口来说并不是必需的，如果一个类中没有静态语句块也没有对变量的赋值操作，那么**编译器可以不为这个类生成clinit()方法**。

接口中不能使用静态语句块，但仍然有变量赋值的初始化操作，因此接口也会生成clinit()方法。**但是接口与类不同，执行接口的clinit()方法不需要先执行父接口的clini>()方法**。**只有当父接口中定义的变量被使用时，父接口才会被初始化**。另外，**接口的实现类在初始化时也不会执行接口的clinit()方法**。

虚拟机会保证一个类的clinit()方法在多线程环境中被正确地加锁和同步。如果有多个线程去同时初始化一个类，那么只会有一个线程去执行这个类的clinit()方法，其它线程都需要阻塞等待，直到活动线程执行clinit()方法完毕。**如果在一个类的clinit()方法中有耗时很长的操作，那么就可能造成多个进程阻塞**。

<https://blog.csdn.net/ns_code/article/details/17881581>

### AOP

AOP代理主要分为静态代理和动态代理，静态代理的代表为AspectJ；而动态代理则以Spring AOP为代表。

Aspectj是一个特殊的编译器。

AspectJ是静态代理的增强。所谓的静态代理，就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强。

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

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

JDK动态代理通过“反射”来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK动态代理的核心是InvocationHandler接口和Proxy类。如果目标类没有实现接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。

注意，CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

静态代理是最简单也是最容易理解的一种方式，只需要在编码的时候创建手动创建代理类调用即可，缺点也很明显，灵活性太差，代理的代码很难复用，编码结束之后类的增强就没有补充的可能性了。

<https://zhuanlan.zhihu.com/p/126322321>

<https://blog.csdn.net/u012094957/article/details/109464760>

## 设计模式

抽象工厂模式：有很多工厂，但是他们生产的产品，具有相似性，因此可以把工厂的共性抽象出来。例如，苹果电脑工厂和联想电脑工厂，都生产鼠标、键盘等；苹果电脑工厂生产苹果鼠标、键盘，联想电脑生产联想鼠标、键盘。根据这样的抽象，就可以定义抽象工厂方法，定义抽象产品，从而能把这些东西抽象出共性。

//定义抽象工厂

public interface Factory {

Mouse createMouse();

Keyboard createKeyboard();

}

//定义抽象产品

public interface Mouse {

void click();

void scroll();

}

//定义抽象产品

public interface KeyBoard {

void keyPress();

}

这样就定义好了抽象工厂和抽象产品。使用的时候，创建对应的Factory，然后调用Factory的createMouse方法和createKeyBoard方法，创造对应的产品出来，然后使用产品的功能。如果要换工厂，只需要修改Factory的实现就行了，改动只需要改工厂就可以，其他的全部不用改。用户使用无感知，不管是什么牌子的鼠标，都可以使用click方法。示例如下：

Factory fac = new XxxFactory();

Mouse ms = fac.createMouse();

ms.click();

KeyBoard kb = fac.createKeyBoard();

kb.keyPress();

下面把抽象工厂和抽象产品落到实处。

//苹果鼠标

public class AppleMouse implements Mouse {

public void click() {

System.out.println(“苹果鼠标被点击”);

}

public void scroll() {

System.out.println(“苹果鼠标被滚轮”);

}

}

//苹果键盘

public class AppleKeyBoard implements KeyBoard {

public void keyPress() {

System.out.println(“苹果键盘被按键”);

}

}

//苹果工厂

public class AppleFactory implements factory {

public Mouse createMouse() {

return new AppleMouse();

}

public KeyBoard createKeyBoard() {

return new AppleKeyBoard();

}

}

//联想鼠标

public class LenovoMouse implements Mouse {

public void click() {

System.out.println(“联想鼠标被点击”);

}

public void scroll() {

System.out.println(“联想鼠标被滚轮”);

}

}

//联想键盘

public class LenovoKeyBoard implements KeyBoard {

public void keyPress() {

System.out.println(“联想键盘被按键”);

}

}

//联想工厂

public class LenovoFactory implements factory {

public Mouse createMouse() {

return new LenovoMouse();

}

public KeyBoard createKeyBoard() {

return new LenovoKeyBoard();

}

}

使用的时候选择实例化哪个工厂就可以了。

## 反编译

Jad是eclipse提供的，idea也用。

Jd-gui是独立的。jd-gui是一个图形界面的反编译工具，我们可以打开该工具，将class文件或者jar文件拖放到界面即可。

# Unicode和UTF-8

Unicode 是「字符集」，UTF-8 是「编码规则」。字符集为每一个「字符」分配一个唯一的 ID（学名为码位 / 码点 / Code Point），编码规则将「码位」转换为字节序列的规则（编码/解码 可以理解为 加密/解密 的过程）。

举一个例子：It's 知乎日报

unicode字符集是这样的编码表

I 0049

t 0074

' 0027

s 0073

0020

知 77e5

乎 4e4e

日 65e5

报 62a5

每一个字符对应一个十六进制数字。

这个字符串总共占用了18个字节（一个字节等于8bit），但是对比中英文的二进制码，可以发现，英文前9位都是0，浪费硬盘，浪费流量。

UTF-8是这样做的：（1）单字节的字符，字节的第一位设为0，对于英语文本，UTF-8码只占用一个字节，和ASCII码完全相同；（2） n个字节的字符(n>1)，第一个字节的前n位设为1，第n+1位设为0，后面字节的前两位都设为10，这n个字节的其余空位填充该字符unicode码，高位用0补足。

于是，”It's 知乎日报“就变成了

I 01001001

t 01110100

' 00100111

s 01110011

00100000

知 11100111 10011111 10100101

乎 11100100 10111001 10001110

日 11100110 10010111 10100101

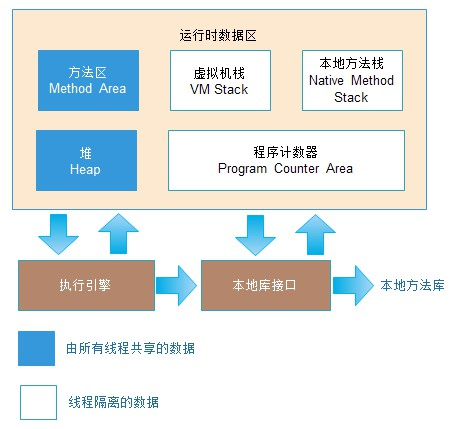
报 11100110 10001010 10100101

整个字符串只用了17个字节，比上边的18个短了一点点。

思考：如果是中文多的，那还是unicode比较省空间。

# JVM

## 运行时分区



线程私有的：程序计数器，虚拟机栈，本地方法栈

线程共享的：堆，方法区，直接内存

**程序计数器：**

程序计数器是一块较小的内存空间，**可以看作是当前线程所执行的字节码的行号指示器**。字节码解释器工作时通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等功能都需要依赖这个计数器来完。

java虚拟机的多线程是通过线程轮流切换并分配CPU的时间片的方式实现的，因此在任何时刻一个处理器（如果是多核处理器，则只是一个核）都只会处理一个线程，为了线程切换后能恢复到正确的执行位置，**每条线程都需要有一个独立的程序计数器**，各线程之间计数器互不影响，独立存储，因此这类内存区域为“线程私有”的内存。

**虚拟机栈：**

虚拟机栈是由一个个栈帧组成，线程在执行一个方法时，便会向栈中放入一个栈帧，每个栈帧中都拥有**局部变量表**、**操作数栈**、**动态链接**、**方法出口信息**。局部变量表主要存放了编译器可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）和对象引用（reference类型，它不同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置）。

Java 虚拟机栈会出现两种异常：StackOverFlowError 和 OutOfMemoryError。

StackOverFlowError：若Java虚拟机栈的内存大小不允许动态扩展，那么**当线程请求栈的深度超过当前Java虚拟机栈的最大深度**的时候，就抛出StackOverFlowError异常。

OutOfMemoryError：若 Java **虚拟机栈的内存大小允许动态扩展，且当线程请求栈时内存用完了，无法再动态扩展了**，此时抛出OutOfMemoryError异常。

操作数栈好像是用来执行数据之间的操作：加减乘除等等的一个内存空间。

动态链接是将符号引用 -> 直接引用。在运行期间将符号引用转化为直接引用的方法被称之为虚方法。

在编译程序代码的时候，栈帧中需要多大的局部变量表，多深的操作数栈都已经完全确定了，并且写入到方法表的Code属性之中，因此一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。

动态链接：

静态类型:编译期间确定的类型(Ocean)

实际类型:运行期间确定的类型(River)

方法的调用者与方法的参数统称为方法的宗量。

class Ocean {

}

class River extends Ocean{

}

class Lake extends Ocean{

}

/\*\*

\* 静态分派

\*/

public class StaticDispatch{

public void getSize(Ocean waterArea){

System.out.println("Ocean is the biggest!");

}

public void getSize(Lake waterArea){

System.out.println("Lake is bigger!");

}

public void getSize(River waterArea){

System.out.println("River is big...");

}

public static void main(String[] args) {

StaticDispatch dispatch = new StaticDispatch();

Ocean river = new River();

Ocean lake = new Lake();

dispatch.getSize(lake);

dispatch.getSize(river);

}

}

运行结果：

Ocean is the biggest!

Ocean is the biggest!

分析：重载时方法的执行依赖的是形参列表。静态类型是在类加载(解析时期)就确定下来的。静态分派可以解释重载。

abstract class Human {

abstract void call();

}

class Father extends Human{

@Override

void call() {

System.out.println("I am the Father!");

}

}

class Mother extends Human{

@Override

void call() {

System.out.println("I am the Mother!");

}

}

public class DynamicDispatch {

public static void main(String[] args) {

Human father = new Father();

Human mother = new Mother();

father.call();

mother.call();

}

}

运行结果：

I am the Father!

I am the Mother!

分析：方法call()的符号引用转换是在运行时期完成的,所以可以说动态分派解释了重载。

“符号引用”的实态：带有类型（tag） / 结构（符号间引用层次）的字符串。

<https://blog.csdn.net/FloatDreamed/article/details/96147409>

<https://blog.csdn.net/u010386612/article/details/80105951>

**本地方法栈：**

和虚拟机栈所发挥的作用非常相似，区别是： 虚拟机栈为虚拟机执行 Java 方法 （也就是字节码）服务，而**本地方法栈则为虚拟机使用到的 Native 方法服务**。

本地方法被执行的时候，在本地方法栈也会创建一个栈帧，用于存放该本地方法的局部变量表、操作数栈、动态链接、出口信息。方法执行完毕后相应的栈帧也会出栈并释放内存空间，也会出现 StackOverFlowError 和 OutOfMemoryError 两种异常。

**方法区：**

方法区与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 Non-Heap（非堆），目的应该是与 Java 堆区分开来。

**堆：**

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

Java 堆是垃圾收集器管理的主要区域，因此也被称作GC堆。从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以Java堆还可以细分为：新生代和老年代：其中新生代又分为：Eden空间、From Survivor、To Survivor空间。进一步划分的目的是更好地回收内存，或者更快地分配内存。从内存分配的角度来看，线程共享的java堆中可能会划分出多个线程私有的分配缓冲区（Thread Local Allocation Buffer，TLAB）。

## 问题

问题：

String str1 = "abc";

String str2 = new String("abc");

sout(str1 == str2);//false

问题：

String s1 = new String("abc"); // 这句话创建了几个对象？

解答：

创建了两个对象。

因为String s1 = new String("abc");// 堆内存的地值值

String s2 = "abc";

System.out.println(s1 == s2);// 输出false,因为一个是堆内存，一个是常量池的内存，故两者是不同的。

System.out.println(s1.equals(s2));// 输出true

Java 基本类型的包装类的大部分都实现了常量池技术，即 Byte、Short、Integer、Long、Character、Boolean；这5种包装类默认创建了数值 [-128，127] 的相应类型的缓存数据，但是超出此范围仍然会去创建新的对象。

两种浮点数类型的包装类 Float、Double 并没有实现常量池技术。

Integer i1 = 33;

Integer i2 = 33;

System.out.println(i1 == i2);// 输出true

Integer i11 = 333;

Integer i22 = 333;

System.out.println(i11 == i22);// 输出false

Double i3 = 1.2;

Double i4 = 1.2;

System.out.println(i3 == i4);// 输出false

## 垃圾回收

有两种算法可以判定对象是否存活：

1.）引用计数算法：给对象中添加一个引用计数器，每当一个地方应用了对象，计数器加1；当引用失效，计数器减1；当计数器为0表示该对象已死、可回收。但是它很难解决两个对象之间相互循环引用的情况。

2.）可达性分析算法：通过一系列称为“GC Roots”的对象作为起点，从这些节点开始向下搜索，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（即对象到GC Roots不可达），则证明此对象已死、可回收。**Java中可以作为GC Roots的对象包括：虚拟机栈中引用的对象、本地方法栈中Native方法引用的对象、方法区静态属性引用的对象、方法区常量引用的对象。**

**在主流的商用程序语言（如我们的Java）的主流实现中，都是通过可达性分析算法来判定对象是否存活的。**

垃圾收集算法

1、标记-清除算法

最基础的算法，分标记和清除两个阶段：首先标记处所需要回收的对象，在标记完成后统一回收所有被标记的对象。

它有两点不足：一个效率问题，标记和清除过程都效率不高；一个是空间问题，标记清除之后会产生大量不连续的内存碎片（类似于我们电脑的磁盘碎片），空间碎片太多导致需要分配大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾回收动作。

2、复制算法

为了解决效率问题，出现了“复制”算法，他将可用内存按容量划分为大小相等的两块，每次只需要使用其中一块。当一块内存用完了，将还存活的对象复制到另一块上面，然后再把刚刚用完的内存空间一次清理掉。这样就解决了内存碎片问题，但是代价就是可以用内容就缩小为原来的一半。

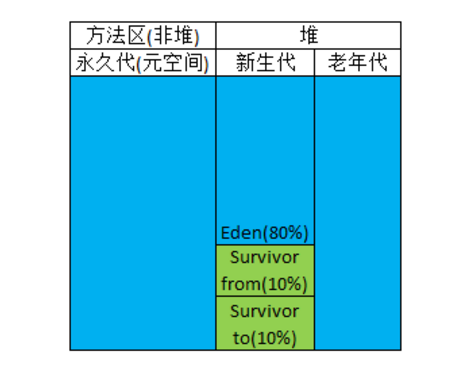
3、标记-整理算法

复制算法在对象存活率较高时就会进行频繁的复制操作，效率将降低。因此又有了标记-整理算法，标记过程同标记-清除算法，但是在后续步骤不是直接对对象进行清理，而是让所有存活的对象都向一侧移动，然后直接清理掉端边界以外的内存。

4、分代收集算法

当前商业虚拟机的GC都是采用分代收集算法，这种算法并没有什么新的思想，而是根据对象存活周期的不同将堆分为：新生代和老年代，方法区称为永久代（在新的版本中已经将永久代废弃，引入了元空间的概念，永久代使用的是JVM内存而元空间直接使用物理内存）。

这样就可以根据各个年代的特点采用不同的收集算法。



新生代中的对象“朝生夕死”，每次GC时都会有大量对象死去，少量存活，使用复制算法。新生代又分为Eden区和Survivor区（Survivor from、Survivor to），大小比例默认为8:1:1。

老年代中的对象因为对象存活率高、没有额外空间进行分配担保，就使用标记-清除或标记-整理算法。

新产生的对象优先进去Eden区，当Eden区满了之后再使用Survivor from，当Survivor from 也满了之后就进行Minor GC（新生代GC），将Eden和Survivor from中存活的对象copy进入Survivor to，然后清空Eden和Survivor from，这个时候原来的Survivor from成了新的Survivor to，原来的Survivor to成了新的Survivor from。复制的时候，如果Survivor to 无法容纳全部存活的对象，则根据老年代的分配担保（类似于银行的贷款担保）将对象copy进去老年代，如果老年代也无法容纳，则进行Full GC（老年代GC）。

大对象直接进入老年代：JVM中有个参数配置-XX:PretenureSizeThreshold，令大于这个设置值的对象直接进入老年代，目的是为了避免在Eden和Survivor区之间发生大量的内存复制。

长期存活的对象进入老年代：JVM给每个对象定义一个对象年龄计数器，如果对象在Eden出生并经过第一次Minor GC后仍然存活，并且能被Survivor容纳，将被移入Survivor并且年龄设定为1。每熬过一次Minor GC，年龄就加1，当他的年龄到一定程度（默认为15岁，可以通过XX:MaxTenuringThreshold来设定），就会移入老年代。但是JVM并不是永远要求年龄必须达到最大年龄才会晋升老年代，如果Survivor 空间中相同年龄（如年龄为x）所有对象大小的总和大于Survivor的一半，年龄大于等于x的所有对象直接进入老年代，无需等到最大年龄要求。

## 调优

为了根据实际情况，让JVM偏向于高吞吐量，或者偏向于低延迟，需要修改JVM的垃圾回收算法。

CMS牺牲了系统的吞吐量来追求收集速度，适合追求垃圾收集速度的服务器上。开启命令：

-XX:+UseConcMarkSweepGC

CMS 处理过程有七个步骤：

1、**初始标记(CMS-initial-mark) ,会导致stw**;

2、并发标记(CMS-concurrent-mark)，**与用户线程同时运行**；

3、预清理（CMS-concurrent-preclean），与用户线程同时运行；

4、可被终止的预清理（CMS-concurrent-abortable-preclean） 与用户线程同时运行；

5、重新标记(CMS-remark) ，会导致swt；

6、并发清除(CMS-concurrent-sweep)，与用户线程同时运行；

7、并发重置状态等待下次CMS的触发(CMS-concurrent-reset)，与用户线程同时运行；

**CMS是基于标记-清除算法的**，CMS只会删除无用对象，不会对内存做压缩，会造成内存碎片，这时候我们需要用到这个参数：

-XX:CMSFullGCsBeforeCompaction=n

意思是说在上一次CMS并发GC执行过后，到底还要再执行多少次full GC才会做压缩。默认是0，也就是在默认配置下每次CMS GC顶不住了而要转入full GC的时候都会做压缩。 如果把CMSFullGCsBeforeCompaction配置为10，就会让上面说的第一个条件变成每隔10次真正的full GC才做一次压缩。

<https://www.jianshu.com/p/86e358afdf17>

为解决CMS算法产生空间碎片和其它一系列的问题缺陷，HotSpot提供了另外一种垃圾回收策略，G1（Garbage First）算法，通过参数-XX:+UseG1GC来启用，该算法在JDK 7u4版本被正式推出。

G1垃圾收集算法主要应用在多CPU大内存的服务中，在满足高吞吐量的同时，尽可能的满足垃圾回收时的暂停时间。

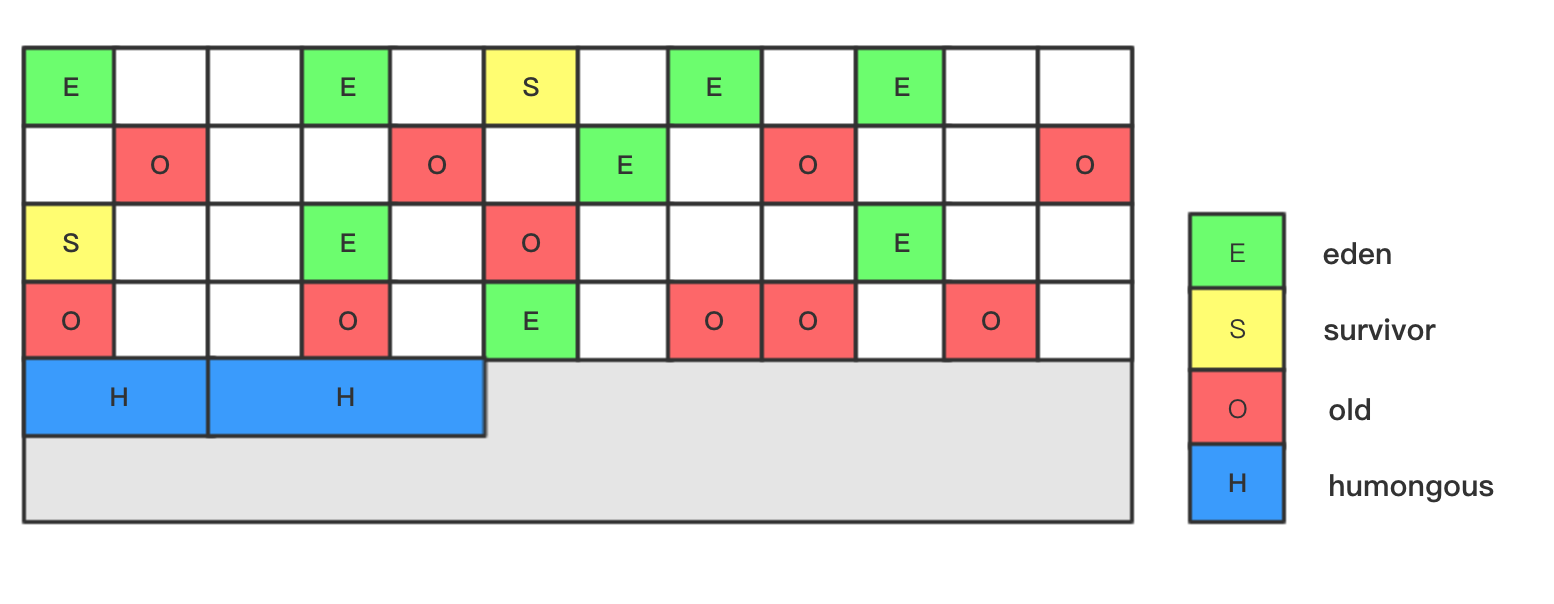
1、垃圾收集线程和应用线程并发执行，和CMS一样

2、空闲内存压缩时避免冗长的暂停时间

3、应用需要更多可预测的GC暂停时间

4、不希望牺牲太多的吞吐性能

5、不需要很大的Java堆



每个Region被标记了E、S、O和H，说明每个Region在运行时都充当了一种角色，其中H是以往算法中没有的，它代表Humongous，这表示这些Region存储的是巨型对象（humongous object，H-obj），当新建对象大小超过Region大小一半时，直接在新的一个或多个连续Region中分配，并标记为H。

堆内存中一个Region的大小可以通过-XX:G1HeapRegionSize参数指定，大小区间只能是1M、2M、4M、8M、16M和32M，总之是2的幂次方，如果G1HeapRegionSize为默认值，则在堆初始化时计算Region的实际大小。

G1中提供了三种模式垃圾回收模式，young gc、mixed gc 和 full gc，在不同的条件下被触发。

Young gc就是S1拷贝到S2，

当越来越多的对象晋升到老年代old region时，为了避免堆内存被耗尽，虚拟机会触发一个混合的垃圾收集器，即mixed gc。除了回收整个young region，还会回收一部分的old region，这里需要注意：是一部分老年代，而不是全部老年代，可以选择哪些old region进行收集，从而可以对垃圾回收的耗时时间进行控制。

如果对象内存分配速度过快，mixed gc来不及回收，导致老年代被填满，就会触发一次full gc，G1的full gc算法就是单线程执行的serial old gc，会导致异常长时间的暂停时间（在JDK9中，Full GC是串行的，JDK10后改为并行执行），需要进行不断的调优，尽可能的避免full gc。

-XX:G1ReservePercent=n

设置作为空闲空间的预留内存百分比，以降低目标空间溢出的风险，默认值是10%

-XX:ConcGCThreads=n

并发GC使用的线程数

-XX:MaxGCPauseMillis=n

最大停顿时间，这是个软目标，JVM将尽可能（但不保证）停顿时间小于这个时间

## GO的垃圾回收

Golang使用的是三色标记法。

**第一步：**stw，然后启动写屏障，新创建的对象会被写屏障保护。

**第二步：**开始标记，从GC root开始标记，采用广度优先策略，被标记的节点置为**灰色**。然后将这些灰色节点引用的节点置为灰色，然后将自己置为**黑色**。一开始所有的节点都是**白色**的，不断这样的标记，最后所有的节点只有白色和黑色两种了，因为灰色的都被标记为黑色了，然后就可以开始回收了。

**第三步：**stw，开始清除白色的节点。

三色标记的一个明显好处是能够让用户程序和垃圾标记并发的进行。

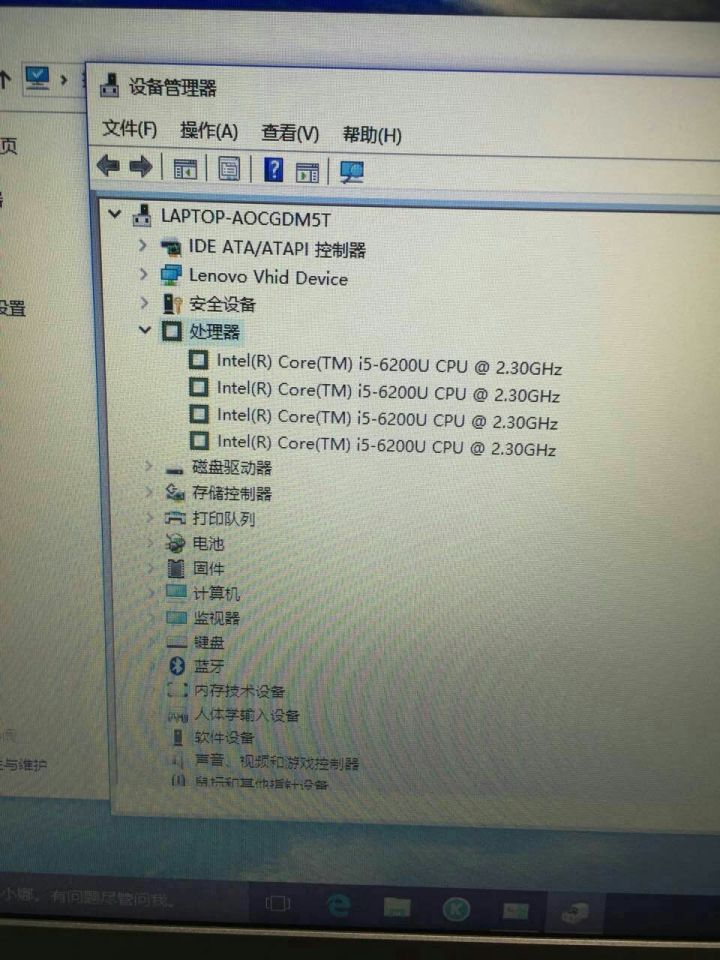
<https://studygolang.com/articles/21688>

<https://www.cnblogs.com/hezhixiong/p/9577199.html>

<https://zhuanlan.zhihu.com/p/74853110>

# CPU

双核4线程显示为4核。有的cpu采用了超线程技术，因为单核单线程cpu都还有一些空闲时间，于是就有了让单核cpu跑两线程的超线程。所以双核4线程不如四核4线程。设备管理器显示的是支持的线程数。



## 加锁原语

如果没有硬件提供的原子操作，只有软件上层是不可能设计出原子操作的。

操作系统之所以能构建出锁之类的原子操作，就是因为硬件已经为我们提供了一些原子操作：中断禁止和启用（ir E/D）、内存加载和存入（L/S）、测试与设置（test&set）、比较与设置（CAS）。

比如禁止中断这个操作是一个硬件步骤，中间无法插入别的操作。中断启用、内存加载、内存存入等均为一个硬件步骤，中间无法插入别的操作。

在这些硬件原子操作之上，我们便可以构建出软件的原子操作：锁、挂起与唤醒、信号量等。

为什么不将中断的禁止和启用函数提供出来由用户直接按自己的需要构建原子操作呢。这种做法理论上是可行的，但是却是危险的，将操作系统赖以工作的基础机制交给用户管理，万一用户水平有限，没有正确的在禁止中断后进行启用，对系统的破坏将是灾难性的。

操作系统通过硬件原语构造了自己的原语，语言通过操作系统原语搭建起了自己的同步机制，剩下的便是我们应用层程序员通过语言提供的同步机制来构建多线程的应用世界了。但值得注意的是，java中的synchronized不是调用的操作系统层级的锁原语，因为目前操作系统还没有提供多核环境下的锁原语，**多核环境的同步是基于共享内存、总线锁、test&set、内存加载和载入等多核环境下的原语实现的**。

**中断保护：**相比另一种方式更加简便，但是只适用于单核环境。因为如果是多核环境，需要发出信号使其它CPU也禁止中断，这将不再是原子操作，而且也让多核心在一定程度上失去了各个核心的独立性（多核的初衷就是让它们可以独立执行）。就算我们这样做了，也将付出极大的代价（保证各个核心的中断是一个原子操作），因此不提倡使用。

**test&set原语：**实现相对复杂，且在多核环境下也可以工作。因为即使是多核，各个核心也在使用共享内存，而该指令针对的就是内存单元。在多核环境下，test&set原语会结合总线锁来保证同一时间只有一个核心可以访问共享内存，从而保证该原语在多核环境下的原子性。

硬件的指令集为我们提供了两种实现思路：

（1）不要打断我的执行（基于中断保护，适用于单核环境，多核环境下不打断你其它核心也可以执行同步代码块）。

（2）可以打断我的执行，但我退出上锁部分前，其它线程不能执行上锁的部分（基于test&set指令，以来共享内存。单核多核均适用，实现较复杂）。

目前操作系统还没有为多核环境提供锁操作，因为代价较大。而像JAVA中提供的对同步代码区的锁操作（synchronized）在多核环境下也是基于共享内存的，即对象的monitor，这也是效率较低的原因（当然更大的原因是需要改变线程状态，即需要进行系统调用进行用户态核心态的切换来阻塞和唤醒线程），在使用前应当进行合理的设计。

<https://www.cnblogs.com/niuyourou/p/11917919.html>

## 进程间通信

（1）socket：基于TCP/IP，使用socket()函数创建socket，服务端调用bind()函数和listen()函数，调用accept()函数接收客户端的请求内容，客户端使用connect()函数发起连接，客户端、服务端调用read()函数和write()函数，完成之后调用close()函数。Socket是全双工的。

Socket s = new Socket("127.0.0.1",9999);

s.getOutputStream();

s.getInputStream();

（2）共享内存：优点是直接访问内存，数据只拷贝一次，速度快；缺点是没有同步机制，需要借助其他手段保证同步。

创建共享内存int shmget(key\_t key, size\_t size, int shmflg)：其中key是由ftok生成的标识，标识系统的唯一IPC资源。Size是申请内存的大小。Shmflg决定是否创建新的共享内存。创建成功返回共享内存标识符，失败返回-1。

获得共享内存void \*shmat(int shmid, const void \*shmaddr, int shmflg)：shmid是shmget返回的共享内存标识符。Shmaddr是从共享内存的第几个地址开始，一般取0。Shmflg决定是只读，还是读写。如果执行成功内核将shmid\_ds结构中的shm\_nattch计数器加1，并返回共享内存的引用，如果失败返回-1。

断开与共享内存的关联int shmdt(const void \*shmaddr)：shmaddr是shmat返回的地址。如果执行成功内核将shmid\_ds结构体中的 shm\_nattch计数器减1，然后返回0。执行失败返回-1。

销毁共享内存int shmctl(int shmid, int cmd, struct shmid\_ds \*buf)：shmid是shmget返回的共享内存标识符。Cmd决定是否删除共享内存。Buf设置为null即可。成功返回0，失败返回-1。

使用示例：<https://blog.csdn.net/ypt523/article/details/79958188>

（3）信号量：可以用来实现共享内存的同步。信号量是一个计数器。具体的去百度查吧。

（4）消息队列：就是消息的链表，存储在内核中，由消息队列标识符标识。

创建或打开消息队列int msgget(key\_t key, int msgflg)：key是消息队列关联的标识符。Msgflg决定存取权限。创建成功返回消息队列标识符，失败返回-1。

消息队列发送int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg)：msqid是msgget返回的消息队列标识符。Msgp是要发的消息，是指向消息的指针，需要用户自己定义一个结构体，系统没有提供。Msgsz是消息的长度。Msgsnd是标志位，设为0即可。执行成功返回0，执行失败返回-1。

消息队列接收ssize\_t msgrcv(int msqid, void \*msgp, size\_t msgsz, long msgtyp,int msgflg)：msqid是msgget返回的消息队列标识符。Msgp是接收消息的结构体对象，msgsz是消息的长度，需要跟发送时指定的长度一致。Msgtyp如果取0，返回消息队列里最早的一个消息，一般取0。Msgsnd是标志位，设为0即可。

删除或断开消息队列int msgctl(int msqid, int cmd, struct msqid\_ds \*buf)：msqid是msgget返回的消息队列标识符。Cmd取IPC\_RMID，表示立即断开连接。Buf设置为0即可。成功返回0，失败返回-1。

<https://blog.csdn.net/weixin_39956356/article/details/86652957>

（5）FIFO（有名管道）：first in，first out。它是一个单向（半双工）数据流，不同于管道的是，每个FIFO有一个路径名与之关联。FIFO可以被无亲缘关系的进程访问。FIFO是Linux基础文件类型中的一种（是一种伪文件），FIFO文件在磁盘上没有数据块，仅仅用来标识内核中一条通道（可以理解为内核中的一块内存）。各进程可以打开这个文件进行read/write，实际上是在读写内核通道，这样就实现了进程间通信。FIFO的名字存在于文件系统中，内容存放在内存中。FIFO先进先出后进后出，不管有多少写的，都是依次写进去的，而且读了就不再在管道里了，不读就一直在管道里，直到爆满。如果是双向的通信，FIFO还得加同步机制，麻烦的要死， 还不如选用其它的通信手段。

创建int mkfifo(const char \*pathname, mode\_t mode)：pathname是一个普通的路径名，它是该FIFO的名字。mode参数指定文件权限。成功返回0，失败返回-1。

打开rst = open(FIFO\_PATH, O\_RDWR)：创建了FIFO之后，就可以像打开文件一样使用open打开它。常见的文件I/O函数都可用于fifo，如close、read、write、unlink等。

<https://blog.csdn.net/longjiang321/article/details/105104549>

（6）无名管道：是半双工的。只能在具有公共祖先的两个进程之间使用，通常一个管道由一个进程创建，在进程调用fork之后，这个管道就能在父进程和子进程之间使用了。

创建int pipe(int fd[2])：成功返回0，失败返回-1。

## Select/Poll/Epoll/Kqueue

select时间复杂度O(n)。它仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个），只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以select具有O(n)的无差别轮询复杂度，同时处理的流越多，无差别轮询时间就越长。

fd数量被限制，即能监听端口的大小有限。具体数目可以cat /proc/sys/fs/file-max察看。32位机默认是1024个。64位机默认是2048。

当套接字比较多的时候，每次select()都要通过遍历FD\_SETSIZE个Socket来完成调度,不管哪个Socket是活跃的,都遍历一遍。这会浪费很多CPU时间。如果能给套接字注册某个回调函数，当他们活跃时，自动完成相关操作，那就避免了轮询，这正是epoll与kqueue做的。

poll时间复杂度O(n)。poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态， 但是它没有最大连接数的限制，原因是它是基于链表来存储的。

epoll时间复杂度O(1)。epoll可以理解为event poll，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实际上是事件驱动（每个事件关联上fd）的，此时我们对这些流的操作都是有意义的。

其中epoll是Linux所特有。

epoll有EPOLLLT和EPOLLET两种触发模式，LT是默认的模式，ET是“高速”模式。LT模式下，只要这个fd还有数据可读，每次 epoll\_wait都会返回它的事件，提醒用户程序去操作，而在ET（边缘触发）模式中，它只会提示一次，直到下次再有数据流入之前都不会再提示了，无论fd中是否还有数据可读。所以在ET模式下，read一个fd的时候一定要把它的buffer读光。

通过epoll\_ctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epoll\_wait便可以收到通知。

epoll只有epoll\_create,epoll\_ctl,epoll\_wait 3个系统调用。

因为epoll内核中实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者的线性下降的性能问题，但是所有socket都很活跃的情况下，可能会有性能问题。

Select、poll：内核需要将消息传递到用户空间，都需要内核拷贝动作

Epoll：通过内核和用户空间共享一块内存来实现的。

在select/poll时代，服务器进程每次都把这100万个连接告诉操作系统(从用户态复制句柄数据结构到内核态)，让操作系统内核去查询这些套接字上是否有事件发生，轮询完后，再将句柄数据复制到用户态，让服务器应用程序轮询处理已发生的网络事件，这一过程资源消耗较大，因此，select/poll一般只能处理几千的并发连接。

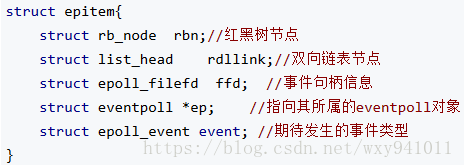
如果没有I/O事件产生，我们的程序就会阻塞在select处。但是依然有个问题，我们从select那里仅仅知道了，有I/O事件发生了，但却并不知道是那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流。

当某一进程调用epoll\_create方法时，Linux内核会创建一个eventpoll结构体，这个结构体中有两个成员与epoll的使用方式密切相关。eventpoll结构体如下所示：



而所有通过epoll\_ctl添加到epoll中的事件都会与设备(网卡)驱动程序建立回调关系，也就是说，当相应的事件发生时会调用这个回调方法。这个回调方法在内核中叫ep\_poll\_callback,它会将发生的事件添加到rdlist双链表中。

对于每一个事件，都会建立一个epitem结构体。



当调用epoll\_wait检查是否有事件发生时，只需要检查eventpoll对象中的rdlist双链表中是否有epitem元素即可。如果rdlist不为空，则把发生的事件复制到用户态，同时将事件数量返回给用户。如果rdlist为空就sleep，等到timeout时间到后即使链表没数据也返回。

当我们执行epoll\_ctl时，除了把socket放到epoll文件系统里，file对象对应的红黑树上之外，还会给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪list链表里。所以，当一个socket上有数据到了，内核在把网卡上的数据copy到内核中后，就来把socket插入到准备就绪链表里了。

LT, ET这件事怎么做到的呢？当一个socket句柄上有事件时，内核会把该句柄插入上面所说的准备就绪list链表，这时我们调用epoll\_wait，会把准备就绪的socket拷贝到用户态内存，然后清空准备就绪list链表，最后，epoll\_wait干了件事，就是检查这些socket，如果不是ET模式（就是LT模式的句柄了），并且这些socket上确实有未处理的事件时，又把该句柄放回到刚刚清空的准备就绪链表了。所以，LT的句柄，只要它上面还有事件，epoll\_wait每次都会返回这个句柄。（从上面这段，可以看出，LT还有个回放的过程，低效了）

<https://blog.csdn.net/wxy941011/article/details/80274233>

<https://www.cnblogs.com/allenwas3/p/8473614.html>

<https://blog.csdn.net/ligupeng7929/article/details/93672312>

## JS事件循环

Event Loop 包含两类：一类是基于 Browsing Context ，一种是基于 Worker ，二者是独立运行的。 下面本文用一个例子，着重讲解下基于 Browsing Context 的事件循环机制。

来看下面这段 JavaScript 代码：

console.log('script start');

setTimeout(function() {

console.log('setTimeout');

}, 0);

Promise.resolve().then(function() {

console.log('promise1');

}).then(function() {

console.log('promise2');

});

console.log('script end');

输出是：

script start,

script end,

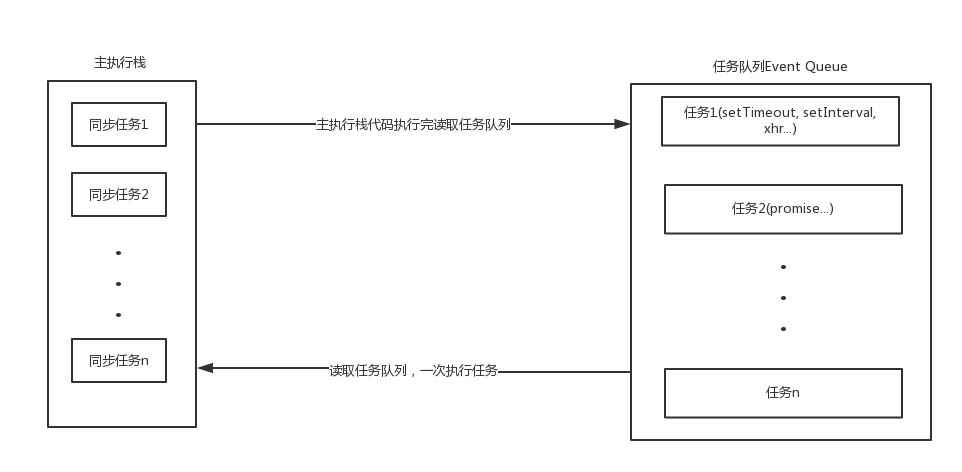
promise1,

promise2,

setTimeout

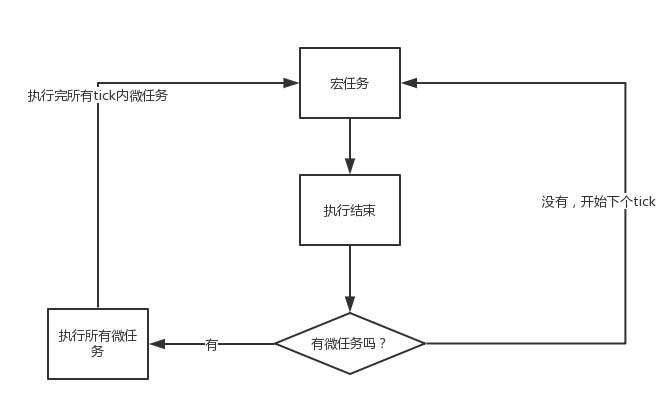
为什么呢？因为任务队列

所有的任务可以分为同步任务和异步任务，同步任务，顾名思义，就是立即执行的任务，同步任务一般会直接进入到主线程中执行；而异步任务，就是异步执行的任务，比如ajax网络请求，setTimeout 定时函数等都属于异步任务，异步任务会通过任务队列( Event Queue )的机制来进行协调。具体的可以用下面的图来大致说明



同步和异步任务分别进入不同的执行环境，同步的进入主线程，即主执行栈，异步的进入 Event Queue 。主线程内的任务执行完毕为空，会去 Event Queue 读取对应的任务，推入主线程执行。 上述过程的不断重复就是我们说的 Event Loop (事件循环)。

在事件循环中，每进行一次循环操作称为tick。



范中规定，task分为两大类, 分别是 Macro Task （宏任务）和 Micro Task（微任务）, 并且每个宏任务结束后, 都要清空所有的微任务。

macro task 主要包含：script( 整体代码)、setTimeout、setInterval、I/O、UI 交互事件、setImmediate(Node.js 环境)

Micro task主要包含：Promise、MutaionObserver、process.nextTick(Node.js 环境)

现在再来看JavaScript代码的输出结果：

（1）整体 script 作为第一个宏任务进入主线程，遇到 console.log，输出 script start

（2）遇到 setTimeout，其回调函数被分发到宏任务 Event Queue 中

（3）遇到 Promise，其 then函数被分到到微任务 Event Queue 中,记为 then1，之后又遇到了 then 函数，将其分到微任务 Event Queue 中，记为 then2

（4）遇到 console.log，输出 script end

（5）执行微任务，首先执行then1，输出 promise1, 然后执行 then2，输出 promise2，这样就清空了所有微任务

（6）执行 setTimeout 任务，输出 setTimeout

至此，输出的顺序是：script start, script end, promise1, promise2, setTimeout

通过以上的演示分析，再来试试下面的题：

console.log('script start');

setTimeout(function() {

console.log('timeout1');

}, 10);

new Promise(resolve => {

console.log('promise1');

resolve();

setTimeout(() => console.log('timeout2'), 10);

}).then(function() {

console.log('then1')

})

console.log('script end');

输出是：

script start,

promise1,

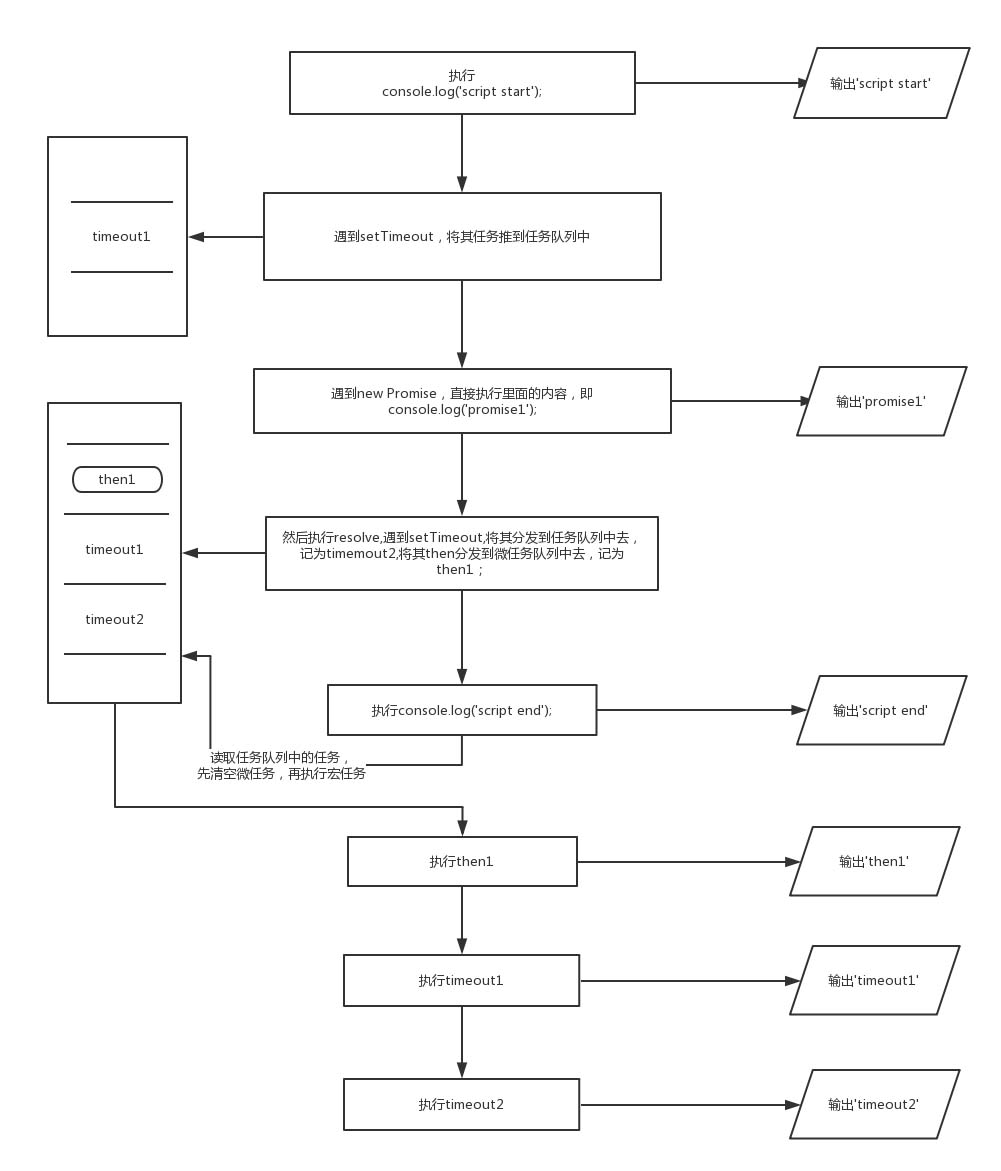
script end,

then1,

timeout1,

timeout2

其中的要点是：执行到 promise，new promise 中的代码立即执行，输出 promise1, 然后执行 resolve ,遇到 setTimeout ,将其分发到任务队列中去，记为 timemout2, 将其 then 分发到微任务队列中去，记为 then1。下面给出执行的流程图：



<https://www.cnblogs.com/yugege/p/9598265.html>

再来一题

for (var i = 0; i < 5; i++) {

setTimeout(function() {

console.log(new Date, i);

}, 1000);

}

console.log(new Date, i);

这个输出会是什么呢？来分析下：

当i=0,1,2,3,4时，执行栈执行循环体里面的代码，发现是setTimeout，将其出栈之后把延时执行的函数交给Timer模块进行处理，进入宏任务的Event Queue。

当i=5的时候，不满足条件，因此for循环结束,console.log(new Date, i)入栈，此时的i已经变成了5。因此输出5。

然后1s过去，开始执行Event Queue里的5个回调函数。由于此时的i已经变成了5，因此几乎同时输出5个5。

结果就是：

5

5

5

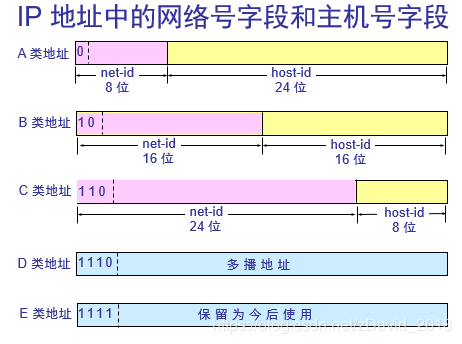
5

5

5

# IP和子网

## 分类



A类网络号：1~126

B类网络号：128~191

C类网络号：192~223

D类网络号：224~239

E类网络号：240~255

## 私有IP

在现在的网络中，IP地址分为公网IP和私有IP地址。公网IP是在Internet使用的IP地址，而私有IP地址是在局域网中使用的IP地址。 由于我们目前使用的IP V4协议的限制，现在IP地址的数量是有限的。这样，我们就不能为居于网中的每一台计算机分配一个公网IP。所以，在局域网中的每台计算机就只能使用私有IP地址了

私有IP地址的范围有：

10.0.0.0-10.255.255.255

172.16.0.0—172.31.255.255

192.168.0.0-192.168.255.255

## 子网

例题：一个C类网络IP地址使用掩码255.255.255.224，每个子网可以容纳（  ）

A.32台主机      B.30台主机       C.16台主机       D.28台主机

解析：子网掩码是255.255.255.224换算成二进制就是11111111.11111111.11111111.11100000,其中前面为1的表示是网络位，后面为0的表示主机位，所以前27位是网络号，后5位是主机号。由此可以看到主机位为5,子网内IP为2^5=32。但是要去掉1个全0和1个全1地址，因为主机位全0表示本网络，全1留作广播地址，这两种情况下子网是没有可用主机地址的。所以有效IP地址为32-2=30个

6.采用子网掩码255.255.255.240可以将网络219.21.33.0划分为（  ）

A.30个子网     B.14个子网      C.12个子网      D.16个子网

解析：

219.21.33.0为一个C类IP地址，划分子网就要划分主机位

子网掩码中最后的240 = 1111 0000（二进制）

划分子网的话 2^4 = 16  -2（全0 全1 不可用）  = 14

所以最后的结果为14个

# Redis

## 数据类型

只有五种基本数据类型：String、Hash、List、Set、SortedSet(zSet)

底层数据结构：SDS、ziplist、hashtable（dict）、linkedlist、intset、skiplist

将一个对象存储在 hash 类型中会占用更少的内存，并且可以更方便的存取整个对象。

**底层实现：**

1、String的底层是SDS（Simple dynamic string），结构如下：

struct sdshdr{

//记录buf数组中已使用字节的数量

//等于 SDS 保存字符串的长度

int len;

//记录 buf 数组中未使用字节的数量

int free;

//字节数组，用于保存字符串

char buf[];

}

关系是：**已使用+free=len**。

2、List的底层是linkedlist和ziplist。当List的元素个数和单个元素的长度较小时，redis会使用ziplist存储，减少内存的占用，其他情况使用linkedlist。具体的说，当list中只包含少量的数据（数量小于512），并且每个数据要么是小整数值或者是长度比较短的字符串（长度小于64字节），那么Redis就会使用ziplist作为list的底层数据实现。那么如果list中的数据不满足以上两个条件的时候，就会将数据存储到linkedlist实现中。**当然512和64字节这两个参数**可以在配置文件中修改list-max-ziplist-value和list-max-ziplist-entries参数。

3、Hash的底层存储可以使用ziplist（压缩列表）和hashtable（dict）。当hash对象可以同时满足以下两个条件时，哈希对象使用ziplist编码。

（1）哈希对象保存的所有键值对的键和值的字符串长度都小于64字节

（2）哈希对象保存的键值对数量小于512个

*因为在数据量小的时候ziplist的查询效率接近于O(1)，与hash效率相似，ziplist是一整块连续内存，实质是个数组，不利于插入删除和查找。ziplist唯一的优势：以字节为单位，通过压缩变长编码的方式节省大量存储空间，当需要使用时，数据可以从磁盘中快速导入内存中处理，而数据在内存中的操作速度是极快的，通过节省存储空间的方式节省了时间。*

4、Set的底层使用了intset和hashtable两种数据结构存储的，intset我们可以理解为数组，hashtable就是普通的哈希表（key为set的值，value为null）。

使用intset存储必须满足下面两个条件，否则使用hashtable（dict），条件如下：

（1）结合对象保存的所有元素都是整数值

（2）集合对象保存的元素数量不超过512个

5、有序集合对象的编码可以是ziplist或者skiplist。同时满足以下条件时使用ziplist编码：

（1）元素数量小于**128个**

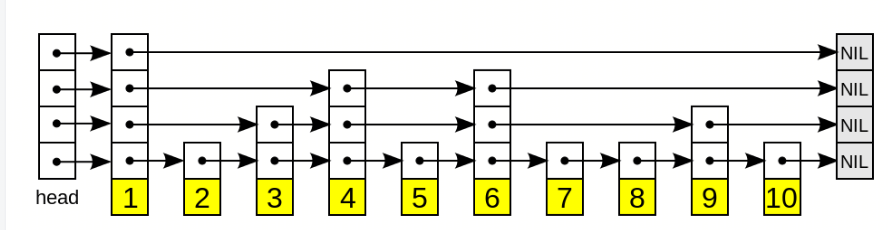
（2）所有member的长度都小于**64字节**

以上两个条件的上限值可通过zset-max-ziplist-entries和zset-max-ziplist-value来修改。

**Skiplist:**

平衡树的插入和删除操作可能引发子树的调整，逻辑复杂，而skiplist的插入和删除只需要修改相邻节点的指针，操作简单又快速。

从内存占用上来说，skiplist比平衡树更灵活一些。一般来说，平衡树每个节点包含2个指针（分别指向左右子树），而skiplist每个节点包含的指针数目平均为1/(1-p)，具体取决于参数p的大小。如果像Redis里的实现一样，取p=1/4，那么平均每个节点包含1.33个指针，比平衡树更有优势。



**Intset:**

intset实质就是一个有序数组，可以看到添加删除元素都比较耗时，查找元素是O(logN)时间复杂度，不适合大规模的数据。

## 对比

redis是单线程。多线程的是Memcached。但是Memcached单个key最大1MB，Redis可以是512MB。

MemCached不支持数据持久化。MemCached数据结构单一。

## 持久化

RDB内存快照记录的是某一个时刻的内存数据，因此能够快速恢复；AOF和RDB混合使用能够使得宕机后数据快速恢复，又能够避免AOF日志文件过大。这个特性从redis4.0开始支持。

## epoll

redis与客户端的连接，使用epoll来进行io多路复用，效率高。

Redis 的瓶颈并不在 CPU，而在内存和网络，所以使用多线程也不会提高太多的性能。

Redis 6的多线程部分只是用来处理网络数据的读写和协议解析，执行命令仍然是单线程顺序执行。

为什么使用单线程：单线程机制使得 Redis 内部实现的复杂度大大降低，Hash 的惰性 Rehash、Lpush 等等 “线程不安全” 的命令都可以无锁进行。

## 缓存问题

### 缓存穿透

缓存穿透是指查询一个一定不存在的数据，由于缓存不命中时需要从数据库查询，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，造成缓存穿透。

比如，我们有一张数据库表，ID都是从1开始的(正数)。但是可能有黑客想把我的数据库搞垮，每次请求的ID都是负数。这会导致我的缓存就没用了，请求全部都找数据库去了，但数据库也没有这个值啊，所以每次都返回空出去。

缓存穿透如果发生了，也可能把我们的数据库搞垮，导致整个服务瘫痪。

解决办法：1.布隆过滤。最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉。

### 缓存雪崩

如果缓存集中在一段时间内失效，发生大量的缓存穿透，所有的查询都落在数据库上，造成了缓存雪崩。

这个没有完美解决办法，但可以分析用户行为，尽量让失效时间点均匀分布。大多数系统设计者考虑用加锁或者队列的方式保证缓存的单线程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。

解决办法：

0. 这个没有完美解决办法，但可以分析用户行为，尽量让失效时间点均匀分布。

1.数据预热。可以通过缓存reload机制，预先去更新缓存，在即将发生大并发访问前手动触发加载缓存不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。

2.缓存永远不过期。

3.在缓存的时候给过期时间加上一个随机值，这样就会大幅度的减少缓存在同一时间过期。

4.对于“Redis挂掉了，请求全部走数据库”这种情况，我们可以有以下的思路：实现Redis的高可用。主从架构+Sentinel（哨兵） 或者Redis Cluster（集群），尽量避免Redis挂掉这种情况发生。

## Redis集群问题

### 不支持多数据库

Redis Cluster不支持多数据库，不支持“select x; ”这个选择数据库的操作。只有select 0可以成功。

为什么不支持多数据库，因为实际中很少用到多数据库。

### 故障转移机制

TAKEOVER：这种手动故障转移的方式比较暴力，slave直接提升自己的epoch为最大的epoch。并把自己变成master。这样在消息交互过程中，旧master能**发现自己的epoch小于该slave**，同时两者负责的slot一致，它会把自己降级为slave。

某个master节点一段时间没收到**心跳响应**，则集群内的master会把该节点标记为pfail，类似sentinel的sdown。集群间的节点会交换相互的认识，超过一半master认为该异常master宕机，则这些master把异常master**标记为fail**，类似sentinel的odown。fail消息会被master广播出来。group的slave收到fail消息后开始竞选成为master。竞选的方式跟sentinel选主的方式类似，都是使用了raft协议，slave会从其他的master拉取选票，**票数最多的slave被选为新的master**，新master会马上给集群内的其他节点发送pong消息，告知自己角色的提升。其他slave接着开始复制新master。等旧master上线后，发现新master的epoch高于自己，通过gossip消息交互，把自己变成了slave。大致就是这么个流程。自动故障转移的方式跟sentinel很像。

## 作为简单消息队列使用

（1）基于List的 LPUSH+BRPOP 的实现

（2）PUB/SUB，订阅/发布模式

（1）基于List的 LPUSH+BRPOP 的实现

典型的命令为：

LPUSH，插入消息队列

BRPOP，从队列中取出消息，阻塞模式

**该模式的优点：**

1.实现简单。

2.Reids支持持久化消息，意味着消息不会丢失，可以重复查看(注意不是消费，只看不用，LRANGE类的指令)。

3.可以保证顺序，保证使用LPUSH命令，可以保证消息的顺序性。

4.使用RPUSH，可以将消息放在队列的开头，达到优先消息的目的，可以实现简易的消息优先队列。

**同时也有缺点：**

1.做消费确认ACK比较麻烦，就是不能保证消费者在读取之后，未处理后的宕机问题。导致消息意外丢失。通常需要自己维护一个Pending列表，保证消息的处理确认。

2.不能重复消费，一旦消费就会被删除。

3.不能做广播模式，例如典型的Pub/Discribe模式。

4.不支持分组消费，需要自己在业务逻辑层解决。

（2）PUB/SUB，订阅/发布模式

SUBSCRIBE，用于订阅信道

PUBLISH，向信道发送消息

UNSUBSCRIBE，取消订阅

生产者和消费者通过相同的一个信道(Channel)进行交互。信道其实也就是队列。通常会有多个消费者。多个消费者订阅同一个信道，当生产者向信道发布消息时，该信道会立即将消息逐一发布给每个消费者。可见，该信道对于消费者是发散的信道，每个消费者都可以得到相同的消息。

**优点是：**

1.一个消息可以发布到多个消费者

2.多信道订阅，消费者可以同时订阅多个信道，从而接收多类消息

3.消息即时发送，消息不用等待消费者读取，消费者会自动接收到信道发布的消息

**缺点是：**

1.消息一旦发布，不能接收。换句话就是发布时若客户端不在线，则消息丢失，不能寻回

2.若消费者客户端出现消息积压，到一定程度，会被强制断开，导致消息意外丢失。通常发生在消息的生产远大于消费速度时

3.不能保证每个消费者接收的时间是一致的

## 一致性哈希

首先求出服务器的哈希值，映射到到0～2^32的圆上。

然后用同样的方法求出数据的哈希值，映射到到0～2^32的圆上。

数据值根据映射，顺时针查找，找到的第一台服务器，就是数据应该存储的位置。

添加服务器，用同样的方法求出哈希值，映射到0～2^32的圆上。产生的影响是，这台新添加的服务器顺时针往下的下一台服务器，能承接到的数据变少，被这台新添加的服务器截流了。可以把原来属于添加服务器的数据，全部搬到这个添加服务器上，这样查找的时候就不用了调到下一个服务器上去找了。

删除服务器，产生的影响是，原本要存在删除的服务器上的数据，都要去找删除服务器顺时针往下的下一台服务器去存储。

**一致性哈希的好处：**添加、删除服务器，只影响局部的数据存储，不影响其他地方的数据存储。这样就能平滑的扩容、缩容了。

**一致性哈希的坏处：**（1）数据倾斜，即有的节点数据多，有的节点数据少。（2）一致性哈希需要将排好序的桶组成一个链表，然后一路找下去，k 个桶查询时间复杂度是 O(k)，所以通常情况下的哈希还是用不一致的实现。当然也可以用路由表、跳转表去优化这个O(k)的时间复杂度，但是多了一步也麻烦。

一致性哈希应用：DHT(Distributed Hash Table，分布式哈希表)网络，就是一致性哈希的应用，也就是P2P网络。

Redis没有使用一致性哈希，因为一致性哈希算法对于数据分布、节点位置的控制不好。

Redis使用了哈希槽，分成了16384个哈希槽，然后使用crc(key)%16384计算应该在哪个槽位，然后根据人为配置的“槽位-redis实例”映射表，映射过去。所以槽位的转移，需要人来配置。这个槽是一个虚拟的槽，并不是真正存在的。

<https://www.cnblogs.com/williamjie/p/9477852.html>

<https://zhuanlan.zhihu.com/p/24440059>

# MySQL

## 事务的四个特征

事务的四个特征是ACID。分别为原子性（ Atomicity ）、一致性（ Consistency ）、隔离性（ Isolation ）和持续性（ Durability ）。

1 、原子性。事务是数据库的逻辑工作单位，事务中包含的各操作要么都做，要么都不做

2 、一致性。事 务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。因此当数据库只包含成功事务提交的结果时，就说数据库处于一致性状态。如果数据库系统 运行中发生故障，有些事务尚未完成就被迫中断，这些未完成事务对数据库所做的修改有一部分已写入物理数据库，这时数据库就处于一种不正确的状态，或者说是 不一致的状态。

3 、隔离性。一个事务的执行不能其它事务干扰。即一个事务内部的操作及使用的数据对其它并发事务是隔离的，并发执行的各个事务之间不能互相干扰。

4 、持续性。也称永久性，指一个事务一旦提交，它对数据库中的数据的改变就应该是永久性的。接下来的其它操作或故障不应该对其执行结果有任何影响。

Savepoint：应该是mysql 5.5以后有的，对于一段比较长的事务，可以定义一个存档点，不至于让事务全部回滚，浪费资源。

具体案例演示可以看：<https://blog.csdn.net/u014745069/article/details/103434169>

<https://my.oschina.net/liangxiao/blog/3076502>

有了savepoint，可以方便的去理解spring的事务嵌套，事务传播机制

<https://blog.csdn.net/nhlbengbeng/article/details/87797781>

<https://www.cnblogs.com/baizhanshi/p/10425467.html>

## 隔离级别

**读未提交（脏读）：**在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。比如事务1更改了字段A，但未提交，此时事务2来读字段A，读取到一个脏值，后来事务1失败回撤。**写不阻塞读，速度快。（没写完可以读）**

**读已提交（不可重复读）：**在一个事务中，可以读取到其他事务已经提交的数据变化，这种读取也就叫做不可重复读，因为两次同样的查询可能会得到不一样的结果。**写阻塞读，速度慢。（没写完不让读）**

在可重复读中，该sql第一次读取到数据后，就将这些数据加锁，其它事务无法修改这些数据，就可以实现可重复 读了。但这种方法却无法锁住insert的数据，所以当事务A先前读取了数据，或者修改了全部数据，事务B还是可以insert数据提交，这时事务A就会 发现莫名其妙多了一条之前没有的数据，这就是幻读，不能通过行锁来避免。

**可重复读（幻读）：**这是MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读 （Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。InnoDB和Falcon存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）机制解决了该问题。**读不阻塞写，速度快。（没读完可以写）**

**串行化：**这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。**读阻塞写，速度慢。（没读完不让写）**

隔离级别实验，可参考：

<https://www.cnblogs.com/jian-gao/p/10795407.html>

## 锁

for update是在数据库中上锁用的，可以为数据库中的行上一个排它锁。当一个事务的操作未完成时候，其他事务可以读取但是不能写入或更新。for update必须在begin与commit之间才生效。

使用示例：select \* from table where xxx **for update;**

for update可能是行锁、可能是表锁、也可能是间隙锁。

例1：SELECT \* FROM foods WHERE id=1 FOR UPDATE; **这个是行锁，id是主键或者索引，只锁住id=1这样一个行，如果没有id=1的记录，则可能有间隙锁。**

例2：SELECT \* FROM foods WHERE name=’羊驼’ FOR UPDATE; **这个是表锁，name不是主键也不是索引，因此需要锁住整个表，如果没有name=’羊驼’的记录，也会加表锁，这点跟行锁不一样。**

例3：SELECT \* FROM foods WHERE id<>’3’ FOR UPDATE; **这个是表锁，因为没有指明是哪个id，需要mysql去判断，所以mysql就加个表锁。**

例4：SELECT \* FROM foods WHERE id LIKE ‘3’ FOR UPDATE; **这个是表锁，因为没有指明是哪个id，需要mysql去判断，所以mysql就加个表锁。当然有时候mysql的优化器认为这个挺明确的，会加行锁，但是我们要考虑最坏的情况，就是表锁。**

例5：SELECT \* FROM foods WHERE id BETWEEN 5 AND 7 FOR UPDATE; **这个是间隙锁+行锁，特定的区间里记录都被锁住了，包括不存在的记录。**

记录被加了for update后，其他事务再来操作加锁的记录，就会陷入**阻塞状态**，直到当前事务释放锁。

<https://zhuanlan.zhihu.com/p/117476959>

InnoDB行锁是通过锁住索引BTREE上的索引项来实现的，Oracle是通过在数据中对相应数据行加锁来实现的，这一点MySQL和Oracle不同。InnoDB这种行锁实现意味者，只有通过索引条件检索数据，InnoDB才会使用行级锁，否则，InnoDB将使用表锁。这就是为什么例1是行锁，例2就是表锁。

如果直接操作的是主键，则直接锁主键；如果操作的是非主键索引，则会先锁住这个非主键索引，然后把非主键索引指向的主键索引锁住。

总结：

**无索引**：表锁

**有索引，指明锁哪一条数据**：行锁

**有索引，锁几条数据不明确**：可能是行锁，也可能是表锁，取决于mysql优化器。Mysql优化器认为即使锁索引，也差不多锁住了大半张表，那还不如锁全表来的简单。

Innodb在RR和RC隔离下的加锁实例分析

例子：select \* from meng\_hinata where id = 10 for update

**组合一：**id列是主键，RC隔离级别

在主键id=10列加上X锁

**组合二：**id列是二级唯一索引，RC隔离级别

在唯一索引id=10列上加X锁，在主键索引上对应列加X锁

**组合三：**id列是二级非唯一索引，RC隔离级别

在二级索引上所有id=10列加上X锁，这些列对应的主键索引列加上X锁

**组合四：**id列上没有索引，RC隔离级别

在聚簇索引上扫描，所有列上加X锁，此处有个优化，不满足的列在加锁后，判断不满足即可释放锁，违背二阶段加锁

**组合五：**id列是主键，RR隔离级别

在主键id=10列上加X锁

**组合六：**id列是二级唯一索引，RR隔离级别

在唯一索引id=10列上加X锁，在主键索引上对应列加X锁

**组合七：**id列是二级非唯一索引，RR隔离级别

在二级索引上查找id=10列，找到则加上X锁和GAP锁，然后对应的聚簇索引列加上X锁（next key锁），最后一个不满足的列只会加上GAP锁。GAP锁是前一个列到当前列，前后开区间的。

**组合八：**id列上没有索引，RR隔离级别

在聚簇索引上扫描，所有列加上X锁和GAP锁

<https://cloud.tencent.com/developer/article/1442426>

### 间隙锁

读已提交级别，不需要防止幻读，也就不需要间隙锁。因此读已提交级别时，间隙锁失效。

普通索引的间隙，优先以普通索引排序，然后再根据主键索引排序。具体可以去看索引的知识点。

Mysql默认使用间隙锁，如果要禁用间隙锁，需要修改my.cnf，加入一条配置：

[mysqld]

innodb\_locks\_unsafe\_for\_binlog = 1

然后重启mysql使其生效。

什么时候会出现间隙锁？或者说间隙锁是用来解决什么问题的？

当我们进行范围查询和操作，并请求排它锁时，InnoDB会给符合条件的记录加锁，对于在范围内但是尚不存在的记录，叫做间隙，InnoDB会对这个间隙加锁。间隙锁就是为了解决事务A在操作某一范围的数据时，事务B在这个范围里增加了不存在的数据，然后事务A更新这一范围的数据时，事务B增加的数据也被A更新了，但是我们开启事务A的时候，并不想更新事务B的数据，事务B的数据也不想被事务A更新，事务A和事务B之间不满足隔离性。为了解决这样的问题，就需要间隙锁。

**所以进行范围查询和操作时，会出现间隙锁。**

行锁（record锁）+ 间隙锁（Gap锁）=临键锁（next key锁）。

举个例子，有表t\_student，其中id为主键，name为非唯一索引。表的数据如下，其中id为2的记录因为某些原因被删掉了，留下了间隙：

|  |  |  |  |
| --- | --- | --- | --- |
| id | name | sex | address |
| 1 | zhaoyi | 0 | beijin |
| 3 | sunsan | 1 | shanghai |
| 4 | lisi | 0 | guangzhou |
| 5 | zhouwu | 0 | shenzhen |
| 6 | wuliu | 1 | hangzhou |

我们要对某一范围的学生数据进行操作：select id,name from t\_student where id > 0 and id < 5 for update;

于是标黄的数据会被加锁，但是id=2这个间隙因为没有数据所以没有加锁：

|  |  |  |  |
| --- | --- | --- | --- |
| id | name | sex | address |
| 1 | zhaoyi | 0 | beijin |
| 3 | sunsan | 1 | shanghai |
| 4 | lisi | 0 | guangzhou |
| 5 | zhouwu | 0 | shenzhen |
| 6 | wuliu | 1 | hangzhou |

场景一，事务A开启事务，查范围内的数据条数，随后事务B在id=2这个地方补上了数据，再随后事务A再来查范围内的数据条数，结果后面一次查询，发现数据多了1条，这就出现了幻读（幻影行）。

|  |  |  |
| --- | --- | --- |
| 时间 | 事务A | 事务B |
| T1 | select count(1) from t\_student where id > 0 and id < 5; |  |
| T2 |  | insert into t\_student values(2,'qianer',1,'nanjing'); |
| T3 |  | commit; |
| T4 | select count(1) from t\_student where id > 0 and id < 5; |  |
| T5 | commit; |  |

**但是实际上，mysql通过MVCC机制，让事务A只能读到快照，读不到事务B的数据修改，所以不会出现数据多1条的情况，也就没有幻读。但是事务A读到的是过期的值，不是当前数据库最新的值，不是最真实的值。所以是变相的解决幻读，实际上幻读仍然存在，只是快照读时没有幻读，当前读时还是会有幻读。但是用for update升级为当前读时会加锁，升级为串行化，所以当前读也没有了幻读。**如果事务A用for update加个间隙锁，事务B就要等事务A释放锁之后才能插数据，这样事务A读到的就是最新的值，最真实的值，事务A提交后，才允许其他事务修改数据。

场景二，事务A开启事务，删除范围内的数据，随后事务B在id=2这个地方插入了数据，然后事务A提交，最后发现把事务B插入的数据删了，但是事务A没想删事务B的数据，但是实际上删了，也就是两个事务不满足隔离性要求。

|  |  |  |
| --- | --- | --- |
| 时间 | 事务A | 事务B |
| T1 | delete from t\_student where id > 0 and id < 5; |  |
| T2 |  | insert into t\_student values(2,'qianer',1,'nanjing'); |
| T3 |  | commit; |
| T4 | commit; |  |

但是实际上，在事务A执行范围delete的时候，mysql给这个范围内加上了next key锁，在事务B执行insert的时候，被间隙锁拦住了，陷入阻塞，只有事务A提交事务后，事务B才能拿到锁，插入数据。**这样间隙锁就解决了事务A和事务B互相干扰的问题，解决了幻影行的问题。**

场景三，事务A开启事务，查范围内的数据条数，随后事务B在id=2这个地方补上了数据，并提交修改。然后事务A根据查询的结果，删除范围内的数据，结果多删了一条，把事务B的数据删了。

|  |  |  |
| --- | --- | --- |
| 时间 | 事务A | 事务B |
| T1 | select count(1) from t\_student where id > 0 and id < 5; |  |
| T2 |  | insert into t\_student values(2,'qianer',1,'nanjing'); |
| T3 |  | commit; |
| T4 | delete from t\_student where id > 0 and id < 5; |  |
| T5 | commit; |  |

**实际上，这就是幻读了，多删了一行。因为mysql可重复读级别，普通读是快照读，读到的是过期的数据，不是最新的数据。当真正去写数据的时候，必须是当前读，必须在最新的数据上操作，就导致了幻读，即第一次读到的，和更新时读到的，不一致，多出来一行。**

**为了解决幻读，需要用for update把快照读升级为当前读，给数据加间隙锁，才能解决幻影行。**

间隙锁的范围，示例如下：

create table t(

a int ,

b int ,

primary key(a),

key(b)

)engine=innodb;

INSERT INTO `t`(`a`, `b`) VALUES (1, 1);

INSERT INTO `t`(`a`, `b`) VALUES (3, 3);

INSERT INTO `t`(`a`, `b`) VALUES (5, 5);

INSERT INTO `t`(`a`, `b`) VALUES (9, 30);

INSERT INTO `t`(`a`, `b`) VALUES (11, 10);

|  |  |
| --- | --- |
| 事务A | 事务B |
| begin; |  |
| select \* from t where b = 10 for update; |  |
|  | update t set a = 4 where b = 5; 成功 |
|  | update t set a = 5 where b = 5; 成功 |
|  | update t set a = 6 where b = 5; 被锁阻塞 |
|  | insert into t(a, b) values(10, 9); 被锁阻塞 |
|  | insert into t(a, b) values(10, 10); 被锁阻塞 |
|  | insert into t(a, b) values(12, 10); 被锁阻塞 |
|  | insert into t(a, b) values(10, 29); 被锁阻塞 |
|  | insert into t(a, b) values(8, 30); 被锁阻塞 |
|  | update t set a = 8 where b = 30; 被锁阻塞 |
|  | update t set a = 10 where b = 30; 成功 |
|  | update t set a = 9 where b = 30; 成功 |

由此看出，间隙锁的范围是：**(前一个key， 当前key)和(当前key， 下一个key)，**只要insert 、update、delete、select for update这些当前读落在间隙锁的范围内，就会因为获取不到锁被阻塞。

哪些索引会产生间隙锁呢？（1）非唯一索引，索引命中，或者范围命中，都可以；（2）主键、唯一索引。in关键字是精确匹配，不产生间隙锁，只产生行锁。Between A and B是范围查询，间隙锁落在(A,B)和(B, next key)区间，跟非唯一索引不太一样。如果记录不存在，会在该记录前第一个存在的记录和该记录后第一个存在的记录之间，加上间隙锁。

<https://www.jianshu.com/p/42e60848b3a6>

<https://zhuanlan.zhihu.com/p/48269420>

**多索引时，测试间隙锁的行为如下：**

CREATE TABLE `gap\_lock` (

`id` bigint(20) NOT NULL,

`number` int(11) NOT NULL,

`val` int(11) NOT NULL,

PRIMARY KEY (`id`) USING BTREE,

KEY `idx\_number` (`number`) USING BTREE,

KEY `idx\_val` (`val`) USING BTREE

) ENGINE=InnoDB DEFAULT CHARSET=utf8 ROW\_FORMAT=DYNAMIC;

INSERT INTO `gap\_lock`(`id`, `number`, `val`) VALUES (1, 1, 4);

INSERT INTO `gap\_lock`(`id`, `number`, `val`) VALUES (3, 1, 3);

INSERT INTO `gap\_lock`(`id`, `number`, `val`) VALUES (6, 6, 6);

INSERT INTO `gap\_lock`(`id`, `number`, `val`) VALUES (10, 11, 8);

INSERT INTO `gap\_lock`(`id`, `number`, `val`) VALUES (12, 10, 9);

INSERT INTO `gap\_lock`(`id`, `number`, `val`) VALUES (15, 6, 5);

INSERT INTO `gap\_lock`(`id`, `number`, `val`) VALUES (18, 1, 6);

select \* from gap\_lock where val = 6 and number = 6 for update;

**number索引产生间隙锁(18, 1, x)到(12, 10, x)，以及number=6的主键行锁。**

**(15, x, x)不可以操作，因为(15, 6, 5)被number索引加了id=15的行锁。**

**val索引产生间隙锁(15, x, 5)到(18, x, 6)，是命中记录的左一个键和右一个键之间，这个与number索引不一样，以及val = 6的主键行锁。**

**(16, 11, 5)，(16, 16, 5)** **不可以操作，因为在间隙锁内。**

**(18, 1, 6)不可以操作，因为被val索引加了id=18的行锁。**

**(12, 10, 9)可以操作。**(14, 11, 5)，(19, 12, 6) ，（30, 30, 6），(9, 11, 8)，(11, 11, 7)可以操作。

如果number索引上字段不重复，mysql优化器可能会不走val索引，最后只加了number的间隙锁。

如果val索引上字段不重复，mysql优化器可能会不走number索引，最后只加了val的间隙锁。

number索引、val索引上字段都重复，就像上面的例子那样，mysql优化器会先走number索引，加number间隙锁，然后走val索引，val的间隙锁范围会发生变异。

### 死锁

Mysql有两种处理死锁的方式：

**等待直到超时（innodb\_lock\_wait\_timeout=50s）**

**发起死锁检测， 主动回滚一条事务， 让其他事务继续执行（innodb\_deadlock\_detect=on）**

由于性能原因， 一般都是使用死锁检测，不然总是超时重试，重复请求操作，效率低。

innodb\_lock\_wait\_timeout可以在配置文件[mysqld]里面设置innodb\_lock\_wait\_timeout=120，**不设默认是50**

也可以使用指令SET GLOBAL innodb\_lock\_wait\_timeout = 120设置

使用指令SET innodb\_lock\_wait\_timeout = 120则只在当前会话生效

锁等待超时不会回滚事务，只会回滚当前语句，如果想锁等待超时回滚整个事务，需要设置**innodb\_rollback\_on\_timeout=on，默认是off**

**死锁检测默认是开启**的，如果想关掉，可以在配置文件[mysqld]里面设置innodb\_deadlock\_detect=off

查看死锁日志：

**show engine innodb status \G;**

在打印出来的信息中找到“LATEST DETECTED DEADLOCK”一节的内容

查看事务的状态：

**SELECT \* FROM information\_schema.INNODB\_TRX\G;**

查看锁等待：

**SELECT \* FROM sys.INNODB\_LOCK\_WAITS\G;**

查看锁等待的原因：

**SELECT \* FROM INNODB\_LOCKS\G;**

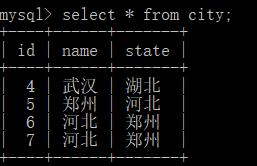
**SELECT \* FROM performance\_schema.DATA\_LOCKS\G; (对于mysql8)**

检测到死锁之后，选择插入更新或者删除的行数最少的事务回滚，基于 INFORMATION\_SCHEMA.INNODB\_TRX 表中的 trx\_weight 字段来判断。**代价最小原则。锁住的行数越多，trx\_weight值越大**。如果一个事务改变了非事务表，那么它的权重比只改变事务表事务的权重大。

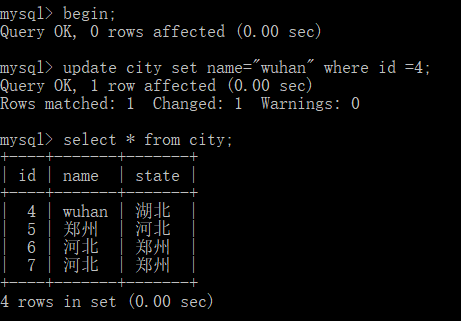
<https://blog.csdn.net/wmq880204/article/details/52505040>

## MVVC

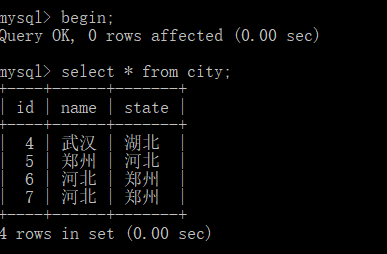
来看一个例子，初始元素数据表如图：



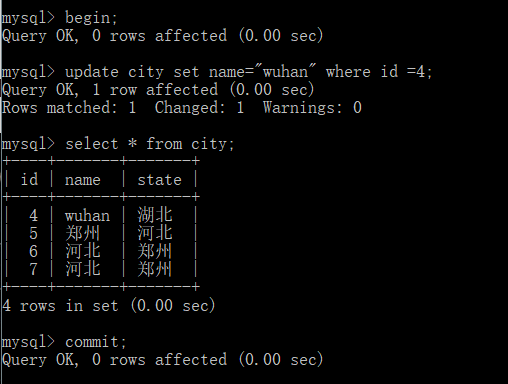
开启事物1 将id为4的那条记录的武汉改成wuhan：



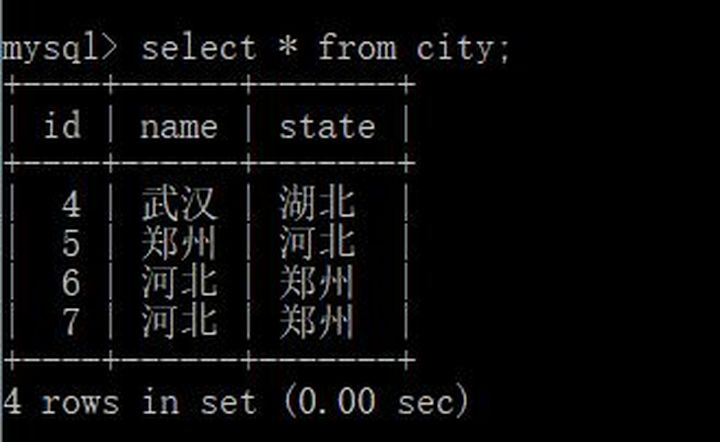
开启事物2，此时事物2中查询不到事物1中的修改。



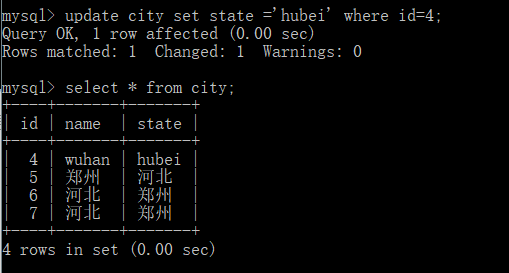
事物1提交：



此时事物2中任然看不到事物1中的修改



接下来在事物2中修改id为4的记录中的湖北改为hubei，结果发现武汉，湖北都被改成拼音了



这个现象算是幻觉吧，但是它是幻读吗？答案是不是，因为幻读的定义是出现幻影行，这个没有出现幻影行，所以不是幻读。这个算是不满足读已提交，因为这些select都是快照读，不是当前读。如果要转为当前读，需要加for update。**快照读的好处就是并发性能好，读不会阻塞，但是处理不好会丢失已提交的更新。当前读的好处是不会丢失已提交的更新，但是会阻塞其他事务更新，实际上是进入了串行化级别。**

**所以使用mysql的时候，一定要小心，可重复读级别存在丢失更新的风险。**

MVVC、快照读，解决了可重复读的问题，但是丧失了读已提交。要读已提交，需要用for update把快照读变为当前读，锁住数据不让别人动，这就是悲观锁。

MVCC只工作在REPEATABLE READ和READ COMMITED隔离级别下。SERIABLABLE隔离级别不需要MVCC。

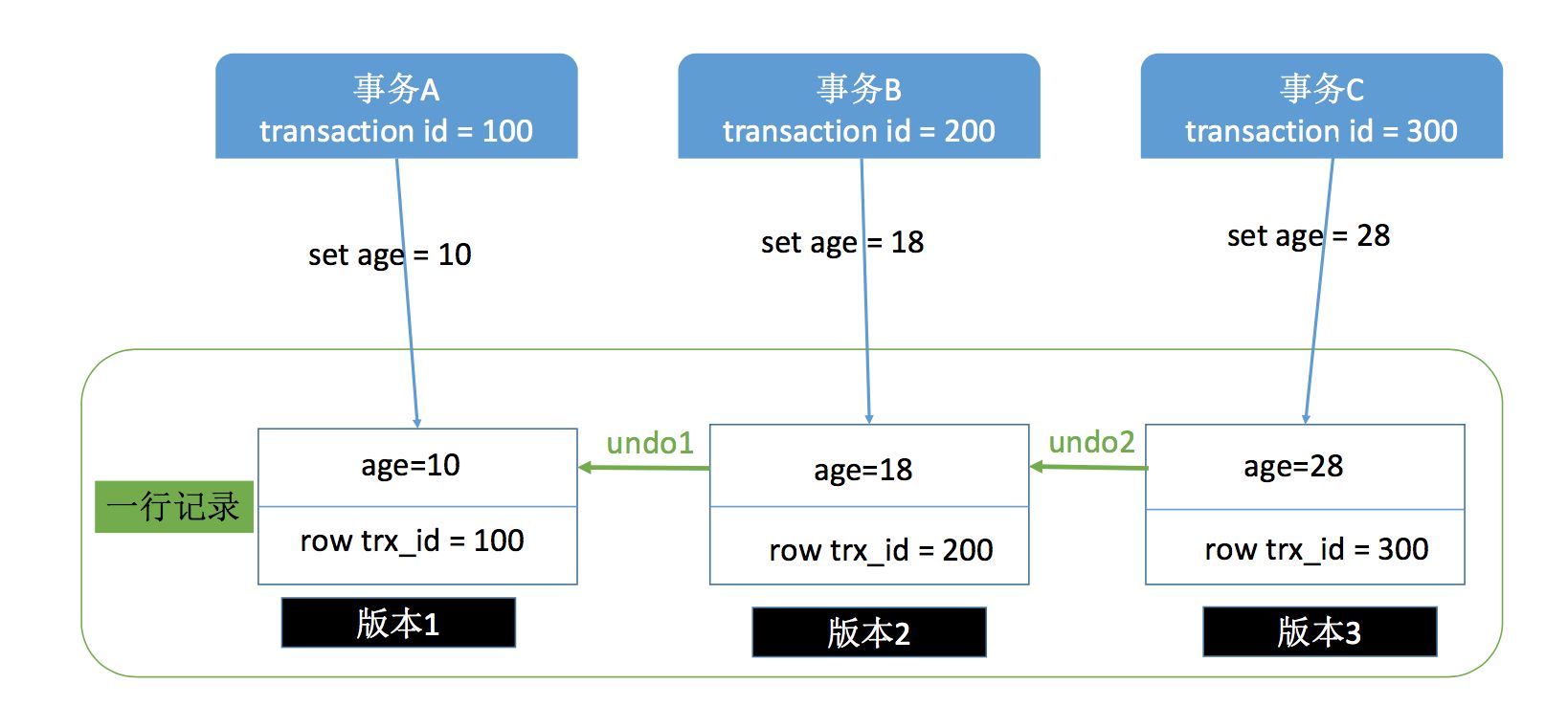
### 快照读

快照读是一致性读，就是可重复读。

当前事务版本号是在begin后第一次操作时生成，不是在begin时生成。事务版本号是InnoDB 的事务系统分配的，单项递增，所以后来的事务其版本号一定更大。Insert、update、delete操作默认开启一个事务，所以是会有事务版本号的。

如果查询的数据没有被加行锁，那么读取当前事务版本号之前或与当前事务版本号一样的数据，对于当前事务版本号之后的数据，读不到。那事务怎么读取当前事务版本号之前的数据呢？用当前数据行的创建版本号、删除版本号，去undo log里回溯查找，计算得到。

如果查询的数据被加了行锁，那么会去undo log里读之前版本的数据。



InnoDB给每行数据都增加了两个隐藏字段，一个记录创建的版本号，一个记录删除的版本号。每次数据被更新的时候，数据行的创建版本号被设置为insert、update、delete的事务版本号，数据行的删除版本号被设置为之前的创建版本号，并且将如何变回之前的数据行的方法放在undo log里。

<https://blog.csdn.net/qq_32573109/article/details/98610368>

<https://blog.csdn.net/lzw2016/article/details/89420391>

### 当前读

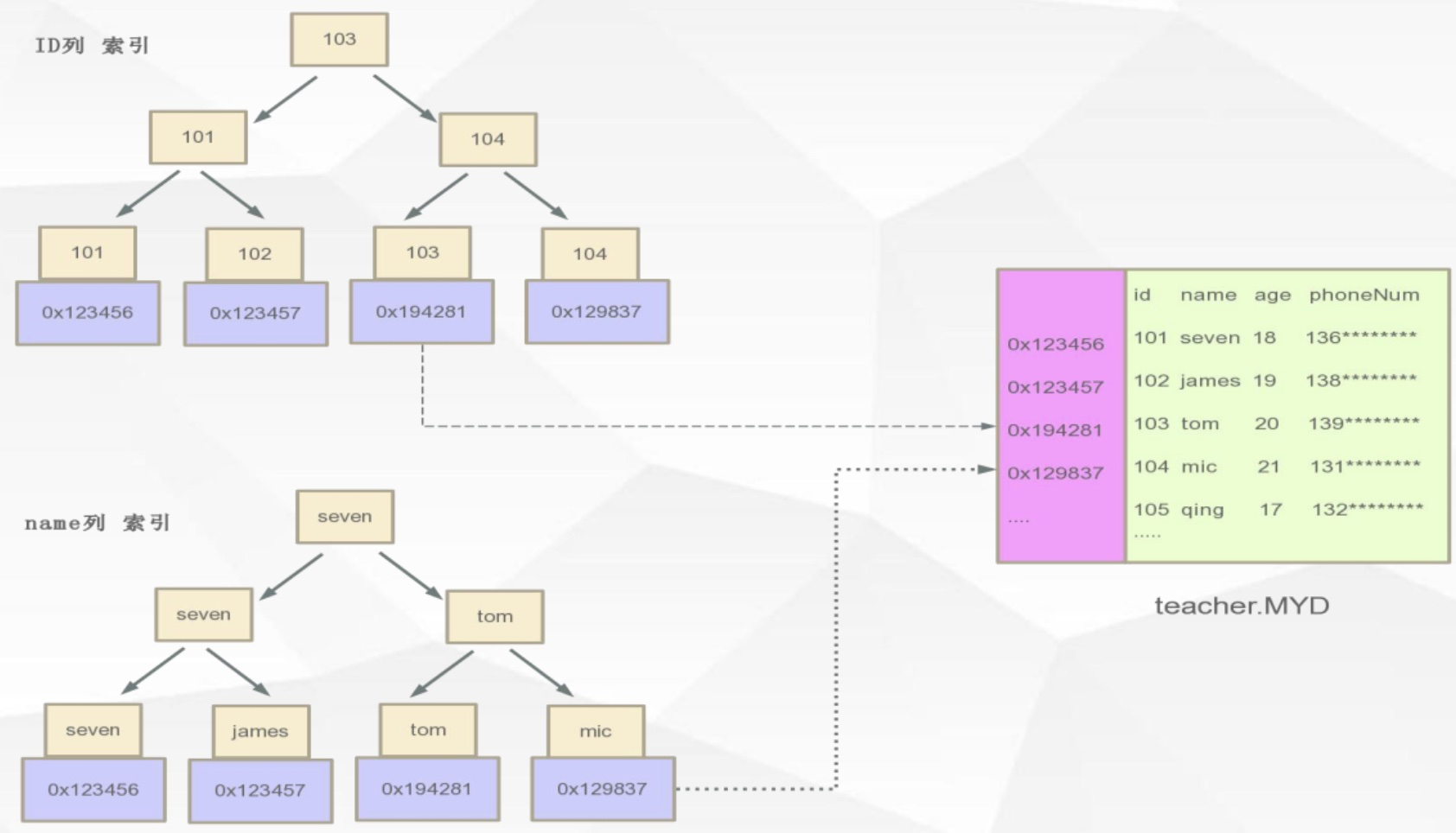
Insert、update、delete为了不丢失更新，不使用快照读，而是在当前数据行上加锁，在最新的数据行上进行更新，这叫当前读。加锁之后，别的事务就不能做当前读了，因为当前读需要先加锁。

## 索引

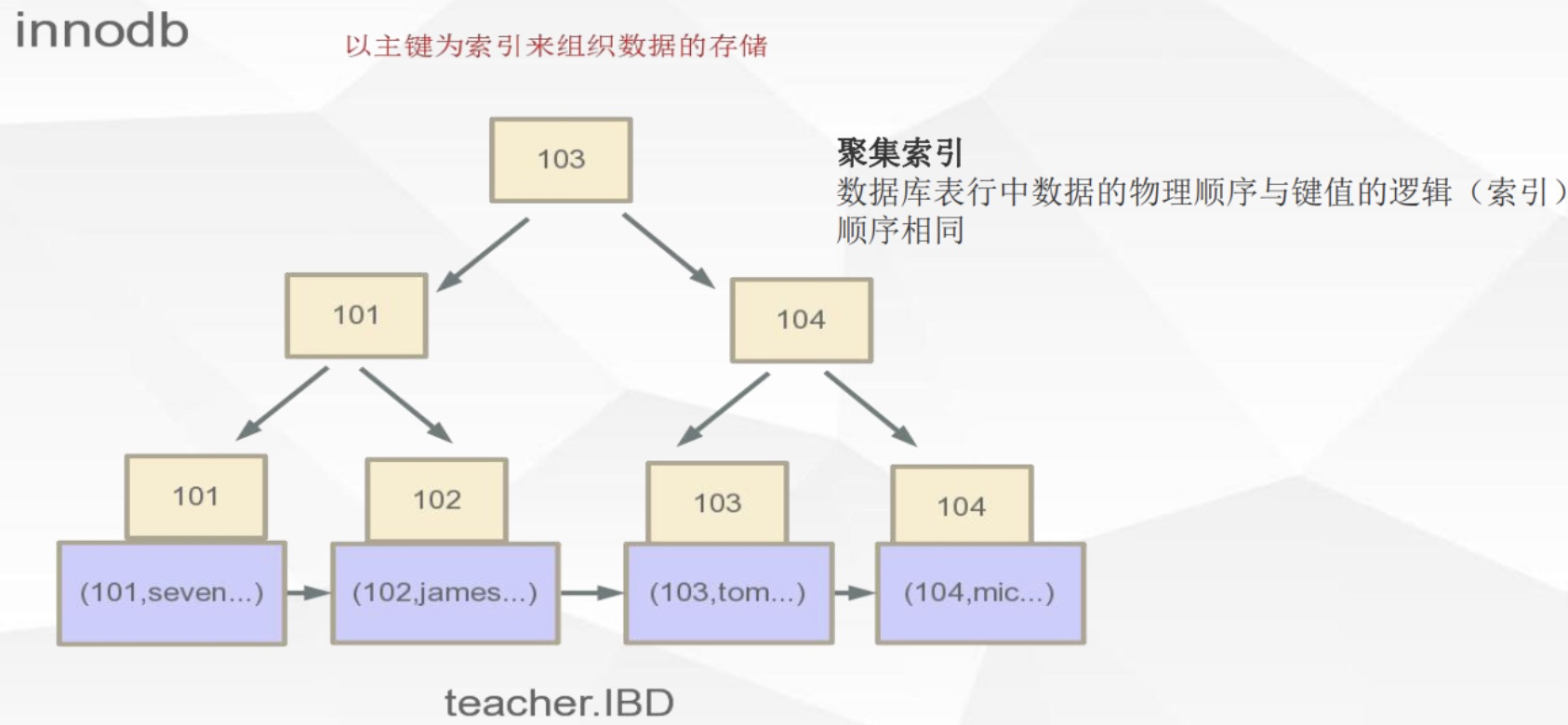
MyISAM和InnoDB存储引擎：只支持BTREE索引， 也就是说默认使用BTREE，不能够更换。Memory存储引擎同时支持hash和BTREE。

非要说InnoDB支持hash索引的话，要指出这个hash索引是InnoDB存储引擎根据表的使用情况自动为表生成hash索引，不可以人为干预。

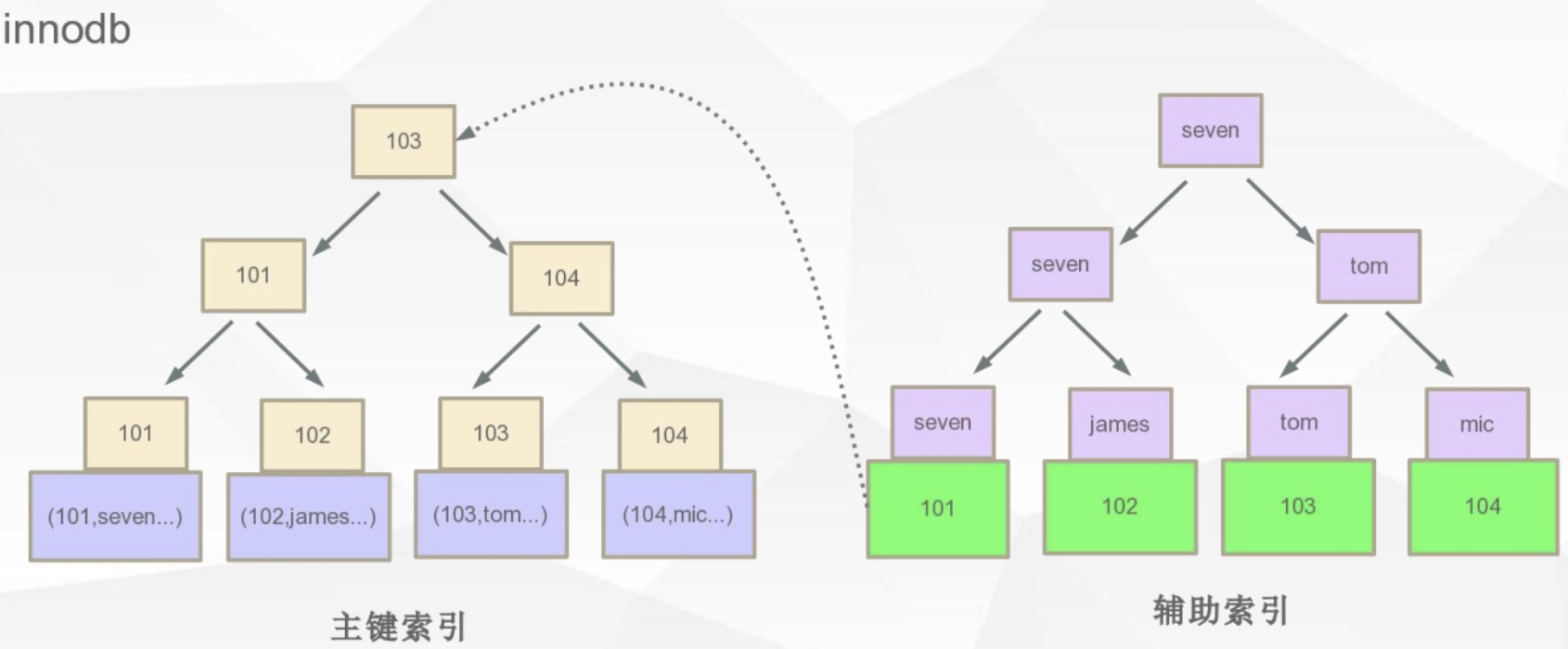
MyISAM的索引是非聚集索引，即索引和数据分开存储，索引里存放数据的地址。表定义存储在.frm文件中，数据存储在.myd文件中，索引存储在.myi文件中。如下图所示：



InnoDB的索引是聚集索引，即索引和数据一起存储，存储在.ibd文件中。数据就在索引的叶子节点中，也就是BTREE的叶子节点中。



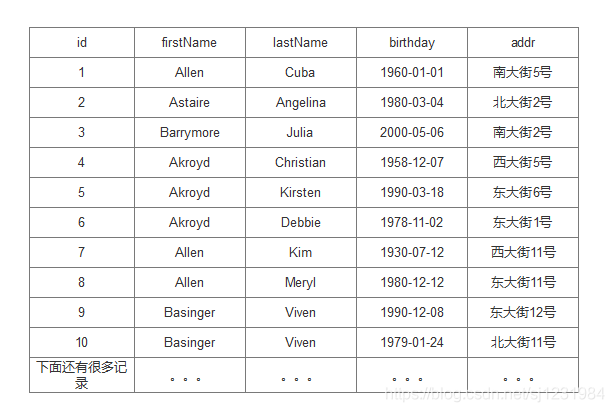
对于非主键索引，其BTREE维护的是对应的主键，其叶子节点存的是主键。非主键索引也是存储在.ibd文件中，不单独存储。



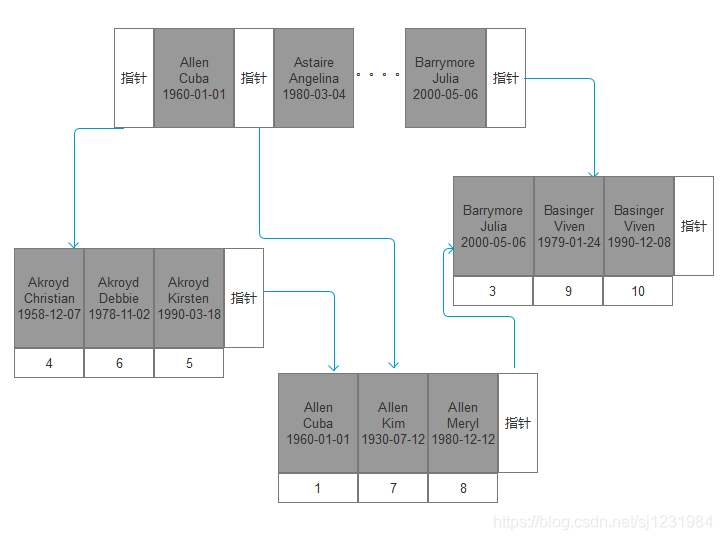
实际使用中，我们最好使用自增主键，来作为表的主键。如果我们不设置自增主键，MySQL会选择唯一键作为主键，创建BTREE。如果连唯一键都没有，MySQL会生成一个6位的row\_id来作为主键。

对于多字段的索引，也是一颗BTREE，这个BTREE的非叶子节点存储了多字段的信息。

例如我们有如下的一张表，我们对firstName,lastName,birthday这三列建立一个复合索引，即key(firstName,lastName,birthday)。



那么这个复合索引的BTREE大致如下，叶子节点存的是主键的值：



需要额外注意的一点是，对于非主键索引BTREE，例如非唯一索引BTREE，其叶子节点中存储的主键，是由小到大有序的，而不是无序的。

优点 提高查询效率

缺点 降低了更新效率

### 覆盖索引

比如创建的索引：create index idx\_name\_phoneNum on users(name,phoneNum);

对于这个查询语句：select name,phoneNum from user where name=?，会直接从索引树中返回，而不会再去查完整数据了，这样就减少了硬盘读，提高了效率。索引覆盖了查询内容，因此叫覆盖索引。

### 回表

如果索引不是覆盖索引，不是主键索引，那么查完索引树，还要回到主键索引的树上，根据主键查一遍表，这就是回表了。

如果索引是覆盖索引，就不需要回表了，查询就快一倍了。

**所以为什么要建立多列索引呢？因为多列索引如果能形成覆盖索引，就不需要回表查询了，就快了。**

<https://www.jianshu.com/p/8991cbca3854>

B+Tree索引和Hash索引区别？

1、哈希索引适合等值查询，但是无法进行范围查询

2、哈希索引没办法利用索引完成排序

3、哈希索引不支持多列联合索引的最左匹配规则

4、如果有大量重复键值的情况下，哈希索引的效率会很低，因为存在哈希碰撞问题

问：排查的时候，有什么手段可以知道有没有走索引查询呢？

答：可以通过explain查看sql语句的执行计划，通过执行计划来分析索引使用情况

(0) select \* from mytable where a=3 and b=5 and c=4;

abc三个索引都在where条件里面用到了，而且都发挥了作用

(1) select \* from mytable where c=4 and b=6 and a=3;

这条语句列出来只想说明 mysql没有那么笨，where里面的条件顺序在查询之前会被mysql自动优化，效果跟上一句一样

(2) select \* from mytable where a=3 and c=7;

a用到索引，b没有用，所以c是没有用到索引效果的

(3) select \* from mytable where a=3 and b>7 and c=3;

a用到了，b也用到了，c没有用到，这个地方b是范围值，也算断点，只不过自身用到了索引

(4) select \* from mytable where b=3 and c=4;

因为a索引没有使用，所以这里 bc都没有用上索引效果

(5) select \* from mytable where a>4 and b=7 and c=9;

a用到了 b没有使用，c没有使用

(6) select \* from mytable where a=3 order by b;

a用到了索引，b在结果排序中也用到了索引的效果，前面说了，a下面任意一段的b是排好序的

(7) select \* from mytable where a=3 order by c;

a用到了索引，但是这个地方c没有发挥排序效果，因为中间断点了，使用 explain 可以看到 filesort

(8) select \* from mytable where b=3 order by a;

b没有用到索引，排序中a也没有发挥索引效果

面试官:"那自增主键达到最大值了，用完了怎么办？"

你:"这问题没遇到过，因为自增主键我们用int类型，一般达不到最大值，我们就分库分表了，所以不曾遇见过！"

问题8:字段为什么要定义为NOT NULL?

1. 索引性能不好
2. 查询会出现一些不可预料的结果。select count(name) from table\_2; 你会发现结果为2，但是实际上是有四条数据的！因为有两个name为null。

问题6:时间字段用什么类型?

一、timestamp，该类型是四个字节的整数，它能表示的时间范围为1970-01-01 08:00:01到2038-01-19 11:14:07。2038年以后的时间，是无法用timestamp类型存储的。

二、datetime，datetime储存占用8个字节，它存储的时间范围为1000-01-01 00:00:00 ~ 9999-12-31 23:59:59。

mysql版本不同而导致的索引长度限制不同

在MySQL5.5版本，引入了innodb\_large\_prefix，用来禁用大型前缀索引，以便与不支持大索引键前缀的早期版本的InnoDB兼容

开启innodb\_large\_prefix可以使单索引的长度限制达到3072字节（但是联合索引总长度限制还是3072字节），禁用时单索引的长度限制为767字节

在MySQL5.5版本与MySQL5.6版本，innodb\_large\_prefix是默认关闭的，在MySQL5.7及以上版本则默认开启

在MySQL8.0版本中，innodb\_large\_prefix已被移除

这就是我在自己机器（MySQL8.0）上可以创建1024字符（utf8字符集下表示3072字节）长的索引，而在服务器（MySQL5.5）上不行的原因

## ISAM

ISAM不支持事务，但是查询更快，对于大量的查询性能更好。但是插入的时候会加表锁，效率就很低，而INNODB加行锁。

ISAM存储空间较小，可以被压缩。

## 日志

在MariaDB/MySQL中，主要有5种日志文件：

**错误日志：**“主机名.err”文件，记录启动信息、停止信息、运行过程中的错误信息。

**查询日志：**“主机名.log”文件，记录建立的客户端连接和执行的几乎所有语句。默认关闭，建议关闭。

**慢查询日志：**“主机名-slow.err”文件，记录所有执行时间超过long\_query\_time的所有查询或不使用索引的查询。哪个查询先完成先记录哪个。可以使用“mysqldumpslow 主机名-slow.log”将相同的慢查询归类显示。慢查询在SQL语句调优的时候非常有用，应该将它启用起来。

**二进制日志：**记录所有更改数据的语句，可用于数据复制。MariaDB/MySQL默认没有启动二进制日志。二进制日志包含了引起或可能引起数据库改变(如delete语句但没有匹配行)的事件信息，但绝不会包括select和show这样的查询语句。二进制目的是为了恢复定点数据库和主从复制，所以出于安全和功能考虑，极不建议将二进制日志和datadir放在同一磁盘上。

**中继日志：**主从复制时使用的日志。

<https://www.cnblogs.com/f-ck-need-u/p/9001061.html>

InnoDB特有的两种日志：

undo.log和redo.log写入时机，参考：

<https://blog.csdn.net/lzw2016/article/details/89420391>

<https://zhuanlan.zhihu.com/p/150105821>

## 自增

MySQL 官方文档中的解释是：在非 NO\_AUTO\_VALUE\_ON\_ZERO模式下，给自增的列赋值为 0，都会被替换为自增序列的下一个值；当该自增列值指定 NOT NULL 时赋值 NULL，也会被替换。当插入其他值时，自增序列的值会被替换为当前列中最大值的下一个值（比如当前自增为10，你插入一个id为20的列，那么自增就从21开始了，自增跳过了10到20这段，这样可以防止自增冲突）。

## 数据库范式

英文Normal form，简称NF。一般来说，数据库只需满足第三范式(3NF）就行了。设计关系数据库时，遵从不同的规范要求，这些不同的规范要求被称为不同的范式，各种范式呈递次规范，越高的范式数据库冗余越小。

第一范式（1NF）：是指在关系模型中，所有的域都应该是原子性的，即数据库表的每一列都是不可分割的原子数据项，而不能是集合，数组，记录等非原子数据项。即实体中的某个属性有多个值时，必须拆分为不同的属性。

比如phoneNumber = "手机号:座机号"，把该放2个字段的属性挤到1个字段里了，就是反1NF的。

第二范式（2NF）：要求实体的属性完全依赖于主关键字。第二范式是在第一范式的基础上建立起来的，即满足第二范式必须先满足第一范式，第二范式要求数据库的每个实例或行必须可以被唯一的区分，即表中要有一列属性可以将实体完全区分，这个属性就是主键，即每一个属性完全依赖于主键。

例如，某张表的主键由“学号”与“学院 ID”确定，那么“学院名称”就不应该出现在这张表里，而是应该放到“学院”表中，否则就出现了数据冗余，也不利于保持数据一致性。

第三范式（3NF）：在2NF基础上，任何非主属性不依赖于其它非主属性（在2NF基础上消除传递依赖）。满足第三范式必须先满足第二范式，第三范式要求一个数据库表中不包含已在其他表中已包含的非主关键字信息， 例如 存在一个课程表，课程表中有课程号(Cno),课程名(Cname),学分(Ccredit)，那么在学生信息表中就没必要再把课程名，学分再存储到学生表中，这样会造成数据的冗余，第三范式就是属性不依赖与其他非主属性。

副键与副键之间，不能存在依赖关系。

定义：一个数据库表中不包含已在其它表中已包含的非主关键字信息。

说人话：不得存在传递式依赖，比如对于一张数据库，里面的元素有son, person, father, grand-father，依赖关系是son -> person, person -> father, father -> grand-father，明显有一个链表式的传递，3NF中禁止此类依赖的出现。修改示范：依赖关系修改为son -> person表，son -> father表，son -> grand-father表

巴斯-科德范式（BCNF）：在3NF基础上，任何非主属性不能对主键子集依赖（在3NF基础上消除对主码子集的依赖）

满足BCNF范式的条件如下：

1、所有的非主属性对每一个码都是完全函数依赖 （暗含 主关键字里面可能有多个码可以将实体区分）

2、所有的主属性对每一个不包含它的码也是完全函数依赖（即所选码与未选择的码之间也是完全函数依赖的）

3、没有任何属性完全函数依赖于非码的任何一组属性（即非主属性之间不能函数依赖）

解释：

例如关系模式 S(Sno,Sname,Sdept,Sage) 假设 Sname具有唯一性

解释条件1：非主属性 （Sdept,Sage） 不仅依赖于Sno,而且依赖于Sname,因为不仅可以通过学号知道学生的信息，还可以通过姓名知道学生的信息。

解释条件2：Sno 与Sname之间也是完全函数依赖关系

解释条件3：没有任何一个属性函数依赖于Sdept和Sage

## 连接

LEFT JOIN：以作为准，右侧可空

RIGHT JOIN：以右为准，左侧可空

INNER JOIN：左右都不为空

OUTER JOIN：左右都可以为空，但不都为空

Mysql没有full join，可以使用union合并查询结果

示例：**SELECT** column\_name(s) **FROM** table1 **INNER** **JOIN** table2 **ON** table1.column\_name=table2.column\_name;

UNION

若需要返回重复的结果，则应该使用union all来代替union

例如制造结果集，其中a是字段名：

**SELECT** 1 a **UNION** **SELECT** 2 **UNION** **SELECT** 3 **UNION** **SELECT** 4 **UNION** **SELECT** 5;

高级用法：update join

示例：**UPDATE** T1, T2 [**INNER JOIN** | **LEFT JOIN**] T1 **ON** T1.C1 = T2. C1 **SET** T1.C2 = T2.C2, T2.C3 = expr **WHERE** condition

如果您学习过了UPDATE语句教程，您可能会注意到使用以下语法更新数据交叉表的另一种方法：

**UPDATE** T1, T2 **SET** T1.c2 = T2.c2, T2.c3 = expr **WHERE** T1.c1 = T2.c1 **AND** condition

在这个UPDATE语句与具有隐式INNER JOIN子句的UPDATE JOIN工作相同。这意味着可以如下重写上述语句：

**UPDATE** T1,T2 **INNER JOIN** T2 **ON** T1.C1 = T2.C1 **SET** T1.C2 = T2.C2, T2.C3 = expr **WHERE** condition

示例：有员工编号与部门ID的表 employee（employee\_id, department\_id），部门ID与部门名称与部门人数的表department(department\_id, department\_name, department\_employee\_num)，要求根据employee表更新department表的department\_employee\_num。那么SQL语句可以这样写

**UPDATE** department **INNER** **JOIN** (**SELECT** count(\*) **AS** num, department\_id **FROM** employee **GROUP** **BY** department\_id) emp **ON** department.department\_id = emp.department\_id **SET** department.department\_employee\_num = emp.num

考点：

（1）表名为nsc，字段为name、subject、score，找出每门课最高得分是谁

**SELECT** nsc.name **FROM** nsc **INNER** **JOIN** (**SELECT** subject, **MAX**(score) **AS** maxscore **FROM** nsc **GROUP** **BY** subject) maxtbl **ON** nsc.subject = maxtbl.subject **AND** nsc.score = maxtbl.score

（2）表名为nsc，字段为name、subject、score，找出所有科目分数大于60的人

**SELECT** name60.name **FROM** (**SELECT** name, **COUNT**(\*) **AS** cnt **FROM** nsc **WHERE** score > 60 **GROUP** **BY** name) name60 **INNER** **JOIN** (**SELECT** name, **COUNT**(\*) **AS** cnt **FROM** nsc **GROUP** **BY** name) namecnt **ON** name60.name = namecnt.name **AND** name60.cnt = namecnt.cnt

## 子查询

子查询可以**分为相关子查询和非相关子查询**。

非相关子查询是独立于外部查询的子查询，子查询总共执行一次，执行完毕后将值传递给外部查询。

相关子查询的执行依赖于外部查询的数据，外部查询执行一行，子查询就执行一次。

因此非相关子查询比相关子查询效率高，但相关子查询能解决一些特定的问题。

举例：

有图书表（学科分类、图书名、价格）

（1）查询图书表中大于图书价格平均值的图书信息

SELECT \* FROM 图书表 WHERE 价格 > (SELECT AVG(价格) FROM 图书表)

以上就是一个非相关子查询，子查询总共执行一次。

（2）查询图书表中大于该学科分类图书价格平均值的图书信息

SELECT \* FROM 图书表 AS books WHERE 价格 > (SELECT AVG(价格) FROM 图书表 AS b WHERE books.学科分类 = b.学科分类)

以上就是一个相关子查询，外部查询会对每一行执行where条件判断，因此子查询就每一行都执行一次，返回该学科分类图书价格平均值。

面试题实战：

有姓名表（省份、姓名、身份证号），查询出各个省份重名数量前10的人名以及对应的省份，假设没有并列前10的人名

SELECT tablea.省份, tablea.姓名, tablea.cnt FROM (SELECT 省份, 姓名, COUNT(\*) AS cnt FROM 姓名表 GROUP BY省份, 姓名) AS tablea WHERE (SELECT COUNT(\*) FROM (SELECT 省份, 姓名, COUNT(\*) AS cnt FROM 姓名表 GROUP BY省份, 姓名) AS tableb WHERE tableb.cnt > tablea.cnt AND tableb.省份 = tablea.省份) < 10 ORDER BY tablea.省份, tablea.cnt desc;

<https://www.cnblogs.com/tangself/archive/2010/01/23/1654623.html>

<https://www.cnblogs.com/buwuliao/p/13268246.html>

<https://www.cnblogs.com/houqijun/p/6085603.html>

## WITH用法

上面子查询的面试题，sql读起来非常费劲，而且where里面子查询套子查询，多了很多不必要的重复，sql执行效率也非常低。为此，需要用with改写上述sql语句：

WITH tablea AS (SELECT 省份, 姓名, COUNT(\*) AS cnt FROM 姓名表 GROUP BY省份, 姓名)

SELECT省份, 姓名, cnt FROM tablea WHERE (SELECT COUNT(\*) FROM tablea AS tableb WHERE tableb.cnt > tablea.cnt AND tableb.省份 = tablea.省份) < 10 ORDER BY省份, cnt desc;

WITH的用法是从mysql 8.x版本开始支持的，mysql 5.x版本不支持。

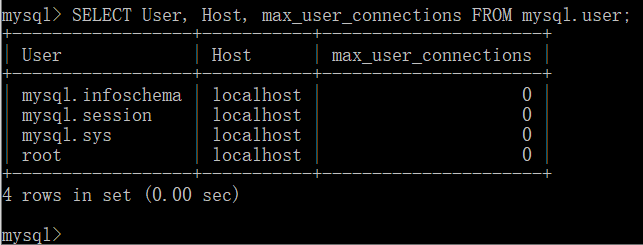
<https://blog.csdn.net/dongying1751/article/details/102457754>

<https://blog.csdn.net/u010520724/article/details/106498870/>

## 权限

查看mysql所有的用户信息

SELECT User, Host, max\_user\_connections FROM mysql.user;



**Mysql8不支持grant all privileges on \*.\* to 'root'@'%' identified by '密码' with grant option;这样的写法了。**

所以要这样先新建用户：

create user 'admin'@'192.168.1.%' identified by 'a123456';

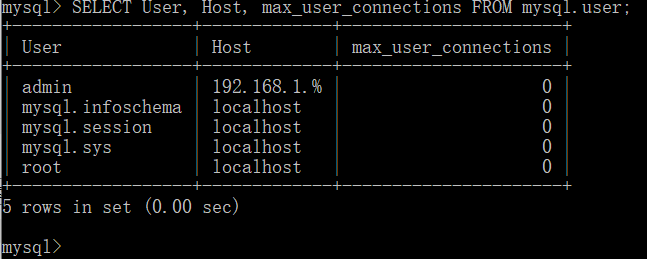
然后赋予权限

grant all privileges on demo.\* to 'admin'@'192.168.1.%';

然后刷新权限

flush privileges;

这样就有了



**如果要修改admin用户的Host，可以直接update，如下：**

update mysql.user set host='%' where user = 'admin';

或者

update mysql.user set host='192.168.2.%' where user = 'admin';

然后刷新权限

flush privileges;

**如果要修改admin用户的密码，可以alter，如下：**

alter user 'admin'@'192.168.2.%' identified by '123asd';

**最后删除用户，如下：**

drop user 'admin'@'192.168.2.%';

# DISTINCT性能

当数据量超过100万时，使用distinct去重的效率极其的差，估计需要一个小时。

这时不能使用distinct，而要使用group by，它只需要60秒。

# MyBatis

<https://www.cnblogs.com/yinjw/p/11757349.html>

1、通常一个Xml映射文件，都会写一个Dao接口与之对应，请问，这个Dao接口的工作原理是什么？Dao接口里的方法，参数不同时，方法能重载吗？

Dao接口里的方法，是不能重载的，因为是全限名+方法名的保存和寻找策略。

Dao接口的工作原理是JDK动态代理，Mybatis运行时会使用JDK动态代理为Dao接口生成代理proxy对象，代理对象proxy会拦截接口方法，转而执行MappedStatement所代表的sql，然后将sql执行结果返回。

2、#{}和${}的区别是什么？

#{}是**预编译处理**，${}是字符串替换。

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

Mybatis在处理${}时，就是把${}替换成变量的值。

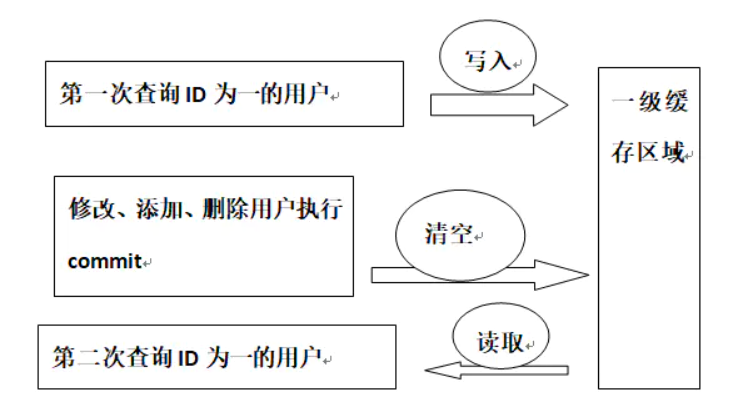
使用#{}可以有效的防止SQL注入，提高系统安全性。

3、Mybatis 中一级缓存与二级缓存的区别？

缓存：合理使用缓存是优化中最常见的方法之一，将从数据库中查询出来的数据放入缓存中，下次使用时不必从数据库查询，而是直接从缓存中读取，避免频繁操作数据库，减轻数据库的压力，同时提高系统性能。

一级缓存是SqlSession级别的缓存：

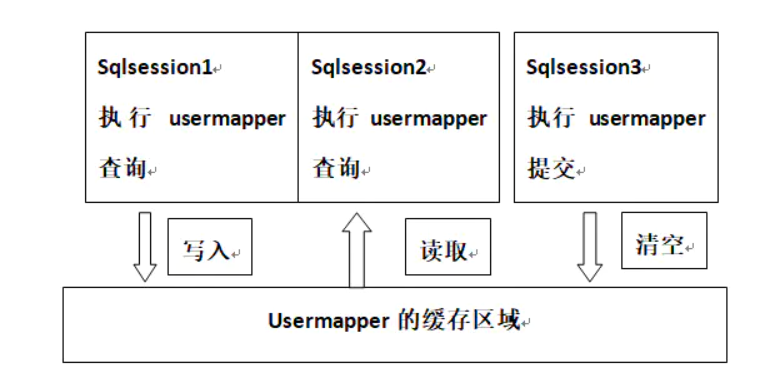
Mybatis对缓存提供支持，但是在没有配置的默认情况下，它只开启一级缓存。一级缓存在操作数据库时需要构造sqlSession对象，在对象中有一个数据结构用于存储缓存数据。不同的sqlSession之间的缓存数据区域是互相不影响的。也就是他只能作用在同一个sqlSession中，不同的sqlSession中的缓存是互相不能读取的。



一级缓存工作原理.png

二级缓存是mapper级别的缓存：

MyBatis的二级缓存是mapper级别的缓存，它可以提高对数据库查询的效率，以提高应用的性能。多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。



二级缓存工作原理.png

开启二级缓存:

A.mybatis.xml配置文件中加入：

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

B.在需要开启二级缓存的mapper.xml中加入caceh标签

<cache/>

## 一级缓存避坑

MyBatis一级缓存（MyBaits 称其为 Local Cache）无法关闭，但是有两种级别可选：

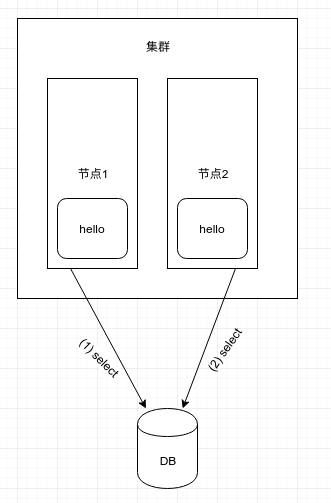
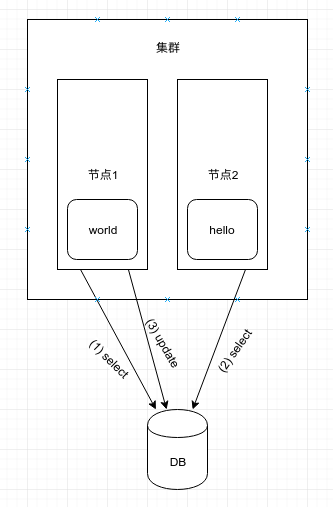
（1）SESSION, //session 级别的缓存

（2）STATEMENT //statement 级别的缓存

session 级别的缓存策略有一个坑，在服务集群时就会出现问题。

假设现在有一个服务集群，有两个节点。

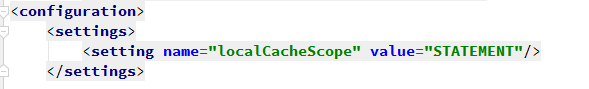
首先，两个节点都进行了同样的查询，两个节点都有自己的一级缓存，后续同样的查询，两个节点将不再查询数据库。

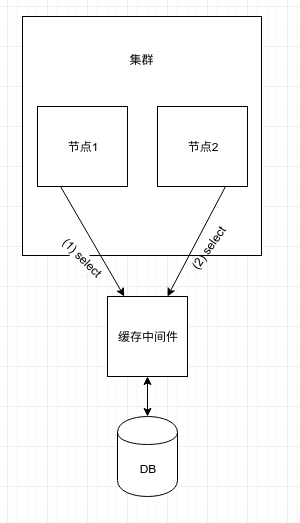
如果此时节点 1 执行了 update 语句，那么节点 1 的一级缓存会被刷新，而节点 2 的一级缓存不会改变。

避坑： 为了避免这个问题，可以将一级缓存的级别设为 statement 级别的，这样每次查询结束都会清掉一级缓存。

在 MyBatis 的配置文件中，添加以下配置：



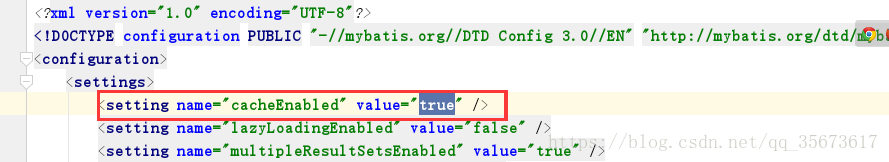
缓存是不可能不要缓存的，这个时候，就需要使用缓存中间件了，由缓存中间件管理缓存。



mybatis和spring结合使用的时候，将原本的DefaultSqlSession替换成了SqlSessionTemplate，并且在SqlSessionTemplate将sqlSession替换成了代理对象，当我们执行sqlSession.selectList方法的时候会调用到SqlSessionInterceptor的invoke方法， 在invoke方法的fianlly中调用了SqlSessionUtils.closeSqlSession(sqlSession, SqlSessionTemplate.this.sqlSessionFactory)将我们的session关闭了。原生的mybatis之所以没有关闭session是因为它把session暴露给我们了，而和spring结合使用的时候并没有提供暴露session的方法，所以只能在这里关，而一旦session关闭了，那一级缓存自然也就失效了。

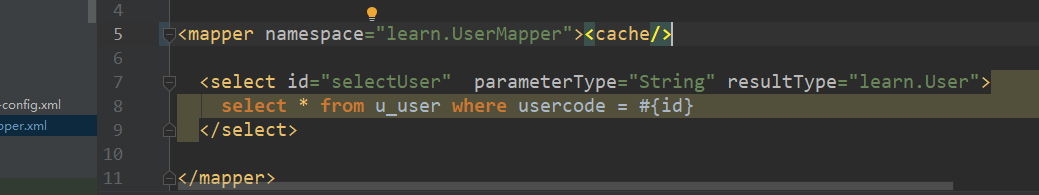
## 二级缓存

二级缓存总开关cacheEnabled默认是true。如果想要关掉，需要显示的设置cacheEnabled为false，如下：



如果不用二级缓存，为了提高性能，应该将总配置文件中的cacheEnabled改为fasle，这样我们执行器可以不用CachingExecutor，能够提高点效率。

如果要想某个Mapper.xml使用到二级缓存，需要设置cacheEnabled为true，或者不设置让其默认为true。然后在<mapper>标签下加一个<cache/>标签。如果不加，该mapper就不使用二级缓存。如果针对要查询的statement不使用缓存，可以在<select>节点中配置useCache="false"，不配置的话，默认useCache="true"。



<select id="findOrderListResultMap" resultMap="ordersUserMap" useCache="false">

为什么不加<cache/>，就无法使用二级缓存呢？因为mybatis代码里这样写的：

package org.apache.ibatis.executor;

……

public class CachingExecutor implements Executor {

……

public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws SQLException {

//这里判断Statement是否开启了cache，开了的话，这里就不是空的了。

Cache cache = ms.getCache();

if (cache != null) {

this.flushCacheIfRequired(ms);

if (ms.isUseCache() && resultHandler == null) {

this.ensureNoOutParams(ms, parameterObject, boundSql);

List<E> list = (List)this.tcm.getObject(cache, key);

if (list == null) {

list = this.delegate.query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);

this.tcm.putObject(cache, key, list);

}

return list;

}

}

return this.delegate.query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);

}

……

}

Mybatis的二级缓存是和命名空间绑定的，所以通常情况下每一个Mapper映射文件都有自己的二级缓存，不同的mapper的二级缓存互不影响。这样的设计一不注意就会引起脏读，从而导致数据一致性的问题。引起脏读的操作通常发生在多表关联操作中，比如在两个不同的mapper中都涉及到同一个表的增删改查操作，当其中一个mapper对这张表进行查询操作，此时另一个mapper进行了更新操作刷新缓存，然后第一个mapper又查询了一次，那么这次查询出的数据是脏数据。出现脏读的原因是他们的操作的缓存并不是同一个。

脏读的避免:

mapper中的操作以单表操作为主，避免在关联操作中使用mapper

在关联操作的mapper中使用参照缓存

## 同类比较

Spring data jpa是跟hibernate放在一起使用的，所以jpa就算是hibernate的一部分。

Mybatis与hibernate的区别，这里就不细说了，网上翻翻吧。

## MyBatis核心

### MybatisAutoConfiguration

MybatisAutoConfiguration 是spring boot 下 mybatis 默认的配置类，只要开启了注释了 @EnableAutoConfiguration 就可以了，spring boot 会默认执行。在spring boot 启动的过程中 @SpringBootApplication 中组合了 EnableAutoConfiguration ，属于spring boot 自动配置和启动过程。MybatisAutoConfiguration在发现springboot中没有创建SqlSessionFactory的时候，执行自己的sqlSessionFactory(DataSource dataSource)方法，具体内容如下：

package org.mybatis.spring.boot.autoconfigure;

……

@Configuration

@ConditionalOnClass({SqlSessionFactory.class, SqlSessionFactoryBean.class})

@ConditionalOnBean({DataSource.class})

@EnableConfigurationProperties({MybatisProperties.class})

@AutoConfigureAfter({DataSourceAutoConfiguration.class})

public class MybatisAutoConfiguration {

……

@Bean

@ConditionalOnMissingBean

public SqlSessionFactory sqlSessionFactory(DataSource dataSource) throws Exception {

//创建SqlSessionFactoryBean

SqlSessionFactoryBean factory = new SqlSessionFactoryBean();

//设置数据源

factory.setDataSource(dataSource);

//使用SpringBootVFS作为扫描器

factory.setVfs(SpringBootVFS.class);

if (StringUtils.hasText(this.properties.getConfigLocation())) {

factory.setConfigLocation(this.resourceLoader.getResource(this.properties.getConfigLocation()));

}

//从properties中获取Configuration

factory.setConfiguration(this.properties.getConfiguration());

if (!ObjectUtils.isEmpty(this.interceptors)) {

factory.setPlugins(this.interceptors);

}

if (this.databaseIdProvider != null) {

factory.setDatabaseIdProvider(this.databaseIdProvider);

}

if (StringUtils.hasLength(this.properties.getTypeAliasesPackage())) {

factory.setTypeAliasesPackage(this.properties.getTypeAliasesPackage());

}

if (StringUtils.hasLength(this.properties.getTypeHandlersPackage())) {

factory.setTypeHandlersPackage(this.properties.getTypeHandlersPackage());

}

if (!ObjectUtils.isEmpty(this.properties.resolveMapperLocations())) {

factory.setMapperLocations(this.properties.resolveMapperLocations());

}

//从SqlSessionFactoryBean中获取SqlSessionFactory。追踪方法调用，发现最后返回的是DefaultSqlSessionFactory对象。

return factory.getObject();

}

@Bean

@ConditionalOnMissingBean

public SqlSessionTemplate sqlSessionTemplate(SqlSessionFactory sqlSessionFactory) {

ExecutorType executorType = this.properties.getExecutorType();

return executorType != null ? new SqlSessionTemplate(sqlSessionFactory, executorType) : new SqlSessionTemplate(sqlSessionFactory);

}

@Configuration

@Import({MybatisAutoConfiguration.AutoConfiguredMapperScannerRegistrar.class})

@ConditionalOnMissingBean({MapperFactoryBean.class})

public static class MapperScannerRegistrarNotFoundConfiguration {

public MapperScannerRegistrarNotFoundConfiguration() {

}

@PostConstruct

public void afterPropertiesSet() {

MybatisAutoConfiguration.log.debug(String.format("No %s found.", MapperFactoryBean.class.getName()));

}

}

public static class AutoConfiguredMapperScannerRegistrar implements BeanFactoryAware, ImportBeanDefinitionRegistrar, ResourceLoaderAware {

private BeanFactory beanFactory;

private ResourceLoader resourceLoader;

public AutoConfiguredMapperScannerRegistrar() {

}

public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {

MybatisAutoConfiguration.log.debug("Searching for mappers annotated with @Mapper'");

ClassPathMapperScanner scanner = new ClassPathMapperScanner(registry);

try {

if (this.resourceLoader != null) {

scanner.setResourceLoader(this.resourceLoader);

}

List<String> pkgs = AutoConfigurationPackages.get(this.beanFactory);

Iterator var5 = pkgs.iterator();

while(var5.hasNext()) {

String pkg = (String)var5.next();

MybatisAutoConfiguration.log.debug("Using auto-configuration base package '" + pkg + "'");

}

scanner.setAnnotationClass(Mapper.class);

scanner.registerFilters();

scanner.doScan(StringUtils.toStringArray(pkgs));

} catch (IllegalStateException var7) {

MybatisAutoConfiguration.log.debug("Could not determine auto-configuration package, automatic mapper scanning disabled.");

}

}

public void setBeanFactory(BeanFactory beanFactory) throws BeansException {

this.beanFactory = beanFactory;

}

public void setResourceLoader(ResourceLoader resourceLoader) {

this.resourceLoader = resourceLoader;

}

}

}

### SqlSessionFactoryBean

MybatisAutoConfiguration执行自己的sqlSessionFactory(DataSource dataSource)方法时，创建了SqlSessionFactoryBean对象，然后向SqlSessionFactoryBean添加各种参数，之后调用了SqlSessionFactoryBean的getObject方法，最终调用了buildSqlSessionFactory方法，返回了一个SqlSessionFactory对象。具体过程如下：

package org.mybatis.spring;

……

public class SqlSessionFactoryBean implements FactoryBean<SqlSessionFactory>, InitializingBean, ApplicationListener<ApplicationEvent> { ……

private SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();

……

public void afterPropertiesSet() throws Exception {

Assert.notNull(this.dataSource, "Property 'dataSource' is required");

Assert.notNull(this.sqlSessionFactoryBuilder, "Property 'sqlSessionFactoryBuilder' is required");

Assert.state(this.configuration == null && this.configLocation == null || this.configuration == null || this.configLocation == null, "Property 'configuration' and 'configLocation' can not specified with together");

this.sqlSessionFactory = this.buildSqlSessionFactory();

}

protected SqlSessionFactory buildSqlSessionFactory() throws IOException {

XMLConfigBuilder xmlConfigBuilder = null;

Configuration configuration;

if (this.configuration != null) {

configuration = this.configuration;

if (configuration.getVariables() == null) {

configuration.setVariables(this.configurationProperties);

} else if (this.configurationProperties != null) {

configuration.getVariables().putAll(this.configurationProperties);

}

} else if (this.configLocation != null) {

xmlConfigBuilder = new XMLConfigBuilder(this.configLocation.getInputStream(), (String)null, this.configurationProperties);

configuration = xmlConfigBuilder.getConfiguration();

} else {

if (LOGGER.isDebugEnabled()) {

LOGGER.debug("Property `configuration` or 'configLocation' not specified, using default MyBatis Configuration");

}

configuration = new Configuration();

configuration.setVariables(this.configurationProperties);

}

if (this.objectFactory != null) {

configuration.setObjectFactory(this.objectFactory);

}

if (this.objectWrapperFactory != null) {

configuration.setObjectWrapperFactory(this.objectWrapperFactory);

}

if (this.vfs != null) {

configuration.setVfsImpl(this.vfs);

}

String[] typeHandlersPackageArray;

String[] var4;

int var5;

int var6;

String packageToScan;

if (StringUtils.hasLength(this.typeAliasesPackage)) {

typeHandlersPackageArray = StringUtils.tokenizeToStringArray(this.typeAliasesPackage, ",; \t\n");

var4 = typeHandlersPackageArray;

var5 = typeHandlersPackageArray.length;

for(var6 = 0; var6 < var5; ++var6) {

packageToScan = var4[var6];

configuration.getTypeAliasRegistry().registerAliases(packageToScan, this.typeAliasesSuperType == null ? Object.class : this.typeAliasesSuperType);

if (LOGGER.isDebugEnabled()) {

LOGGER.debug("Scanned package: '" + packageToScan + "' for aliases");

}

}

}

int var27;

if (!ObjectUtils.isEmpty(this.typeAliases)) {

Class[] var25 = this.typeAliases;

var27 = var25.length;

for(var5 = 0; var5 < var27; ++var5) {

Class<?> typeAlias = var25[var5];

configuration.getTypeAliasRegistry().registerAlias(typeAlias);

if (LOGGER.isDebugEnabled()) {

LOGGER.debug("Registered type alias: '" + typeAlias + "'");

}

}

}

if (!ObjectUtils.isEmpty(this.plugins)) {

Interceptor[] var26 = this.plugins;

var27 = var26.length;

for(var5 = 0; var5 < var27; ++var5) {

Interceptor plugin = var26[var5];

configuration.addInterceptor(plugin);

if (LOGGER.isDebugEnabled()) {

LOGGER.debug("Registered plugin: '" + plugin + "'");

}

}

}

if (StringUtils.hasLength(this.typeHandlersPackage)) {

typeHandlersPackageArray = StringUtils.tokenizeToStringArray(this.typeHandlersPackage, ",; \t\n");

var4 = typeHandlersPackageArray;

var5 = typeHandlersPackageArray.length;

for(var6 = 0; var6 < var5; ++var6) {

packageToScan = var4[var6];

configuration.getTypeHandlerRegistry().register(packageToScan);

if (LOGGER.isDebugEnabled()) {

LOGGER.debug("Scanned package: '" + packageToScan + "' for type handlers");

}

}

}

if (!ObjectUtils.isEmpty(this.typeHandlers)) {

TypeHandler[] var28 = this.typeHandlers;

var27 = var28.length;

for(var5 = 0; var5 < var27; ++var5) {

TypeHandler<?> typeHandler = var28[var5];

configuration.getTypeHandlerRegistry().register(typeHandler);

if (LOGGER.isDebugEnabled()) {

LOGGER.debug("Registered type handler: '" + typeHandler + "'");

}

}

}

if (this.databaseIdProvider != null) {

try {

configuration.setDatabaseId(this.databaseIdProvider.getDatabaseId(this.dataSource));

} catch (SQLException var24) {

throw new NestedIOException("Failed getting a databaseId", var24);

}

}

if (this.cache != null) {

configuration.addCache(this.cache);

}

if (xmlConfigBuilder != null) {

try {

xmlConfigBuilder.parse();

if (LOGGER.isDebugEnabled()) {

LOGGER.debug("Parsed configuration file: '" + this.configLocation + "'");

}

} catch (Exception var22) {

throw new NestedIOException("Failed to parse config resource: " + this.configLocation, var22);

} finally {

ErrorContext.instance().reset();

}

}

if (this.transactionFactory == null) {

this.transactionFactory = new SpringManagedTransactionFactory();

}

configuration.setEnvironment(new Environment(this.environment, this.transactionFactory, this.dataSource));

if (!ObjectUtils.isEmpty(this.mapperLocations)) {

Resource[] var29 = this.mapperLocations;

var27 = var29.length;

for(var5 = 0; var5 < var27; ++var5) {

Resource mapperLocation = var29[var5];

if (mapperLocation != null) {

try {

XMLMapperBuilder xmlMapperBuilder = new XMLMapperBuilder(mapperLocation.getInputStream(), configuration, mapperLocation.toString(), configuration.getSqlFragments());

xmlMapperBuilder.parse();

} catch (Exception var20) {

throw new NestedIOException("Failed to parse mapping resource: '" + mapperLocation + "'", var20);

} finally {

ErrorContext.instance().reset();

}

if (LOGGER.isDebugEnabled()) {

LOGGER.debug("Parsed mapper file: '" + mapperLocation + "'");

}

}

}

} else if (LOGGER.isDebugEnabled()) {

LOGGER.debug("Property 'mapperLocations' was not specified or no matching resources found");

}

return this.sqlSessionFactoryBuilder.build(configuration);

}

public SqlSessionFactory getObject() throws Exception {

if (this.sqlSessionFactory == null) {

this.afterPropertiesSet();

}

return this.sqlSessionFactory;

}

……

}

### SqlSessionFactoryBuilder

SqlSessionFactoryBean的sqlSessionFactoryBuilder默认使用SqlSessionFactoryBuilder，而且SqlSessionFactoryBean配置好Configuration后，调用了SqlSessionFactoryBuilder的build(Configuration config)方法，这个方法创建并返回了DefaultSqlSessionFactory对象，如下：

package org.apache.ibatis.session;

……

public class SqlSessionFactoryBuilder {

……

public SqlSessionFactory build(Configuration config) {

return new DefaultSqlSessionFactory(config);

}

……

}

### DefaultSqlSessionFactory

DefaultSqlSessionFactory根据被配置的configuration，当执行sql的时候，会调用其中的openSession方法，生成Executor，再生成SqlSession。

package org.apache.ibatis.session.defaults;

……

public class DefaultSqlSessionFactory implements SqlSessionFactory {

private final Configuration configuration;

public DefaultSqlSessionFactory(Configuration configuration) {

this.configuration = configuration;

}

……

//执行sql的时候，会调用这些openSession方法。

public SqlSession openSession(ExecutorType execType) {

return this.openSessionFromDataSource(execType, (TransactionIsolationLevel)null, false);

}

……

private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsolationLevel level, boolean autoCommit) {

Transaction tx = null;

DefaultSqlSession var8;

try {

Environment environment = this.configuration.getEnvironment();

TransactionFactory transactionFactory = this.getTransactionFactoryFromEnvironment(environment);

tx = transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);

Executor executor = this.configuration.newExecutor(tx, execType);

var8 = new DefaultSqlSession(this.configuration, executor, autoCommit);

} catch (Exception var12) {

this.closeTransaction(tx);

throw ExceptionFactory.wrapException("Error opening session. Cause: " + var12, var12);

} finally {

ErrorContext.instance().reset();

}

return var8;

}

private SqlSession openSessionFromConnection(ExecutorType execType, Connection connection) {

DefaultSqlSession var8;

try {

boolean autoCommit;

try {

autoCommit = connection.getAutoCommit();

} catch (SQLException var13) {

autoCommit = true;

}

Environment environment = this.configuration.getEnvironment();

TransactionFactory transactionFactory = this.getTransactionFactoryFromEnvironment(environment);

Transaction tx = transactionFactory.newTransaction(connection);

Executor executor = this.configuration.newExecutor(tx, execType);

var8 = new DefaultSqlSession(this.configuration, executor, autoCommit);

} catch (Exception var14) {

throw ExceptionFactory.wrapException("Error opening session. Cause: " + var14, var14);

} finally {

ErrorContext.instance().reset();

}

return var8;

}

private TransactionFactory getTransactionFactoryFromEnvironment(Environment environment) {

return (TransactionFactory)(environment != null && environment.getTransactionFactory() != null ? environment.getTransactionFactory() : new ManagedTransactionFactory());

}

private void closeTransaction(Transaction tx) {

if (tx != null) {

try {

tx.close();

} catch (SQLException var3) {

;

}

}

}

}

### DefaultSqlSession

DefaultSqlSession实现了SqlSession接口，构造的时候需要传入configuration、executor、autoCommit参数。执行sql的时候从configuration里获取MappedStatement，告诉executor用什么方法、哪个MappedStatement、什么查询参数、查多少行、用什么ResultHandler，然后交给executor去执行。

DefaultSqlSession在内部构造了一个StrictMap，覆盖了HashMap的get方法，如果get失败，就抛绑定异常。

DefaultSqlSession把sql参数用wrapCollection方法包起来，根据类型放入StrictMap中，键名对应为collection、list、array。

package org.apache.ibatis.session.defaults;

……

public class DefaultSqlSession implements SqlSession {

public DefaultSqlSession(Configuration configuration, Executor executor, boolean autoCommit) {

this.configuration = configuration;

this.executor = executor;

this.dirty = false;

this.autoCommit = autoCommit;

}

……

public <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds) {

List var5;

try {

MappedStatement ms = this.configuration.getMappedStatement(statement);

var5 = this.executor.query(ms, this.wrapCollection(parameter), rowBounds, Executor.NO\_RESULT\_HANDLER);

} catch (Exception var9) {

throw ExceptionFactory.wrapException("Error querying database. Cause: " + var9, var9);

} finally {

ErrorContext.instance().reset();

}

return var5;

}

public void select(String statement, Object parameter, RowBounds rowBounds, ResultHandler handler) {

try {

MappedStatement ms = this.configuration.getMappedStatement(statement);

this.executor.query(ms, this.wrapCollection(parameter), rowBounds, handler);

} catch (Exception var9) {

throw ExceptionFactory.wrapException("Error querying database. Cause: " + var9, var9);

} finally {

ErrorContext.instance().reset();

}

}

public int update(String statement, Object parameter) {

int var4;

try {

this.dirty = true;

MappedStatement ms = this.configuration.getMappedStatement(statement);

var4 = this.executor.update(ms, this.wrapCollection(parameter));

} catch (Exception var8) {

throw ExceptionFactory.wrapException("Error updating database. Cause: " + var8, var8);

} finally {

ErrorContext.instance().reset();

}

return var4;

}

public void commit(boolean force) {

try {

this.executor.commit(this.isCommitOrRollbackRequired(force));

this.dirty = false;

} catch (Exception var6) {

throw ExceptionFactory.wrapException("Error committing transaction. Cause: " + var6, var6);

} finally {

ErrorContext.instance().reset();

}

}

public void close() {

try {

this.executor.close(this.isCommitOrRollbackRequired(false));

this.closeCursors();

this.dirty = false;

} finally {

ErrorContext.instance().reset();

}

}

private Object wrapCollection(Object object) {

DefaultSqlSession.StrictMap map;

if (object instanceof Collection) {

map = new DefaultSqlSession.StrictMap();

map.put("collection", object);

if (object instanceof List) {

map.put("list", object);

}

return map;

} else if (object != null && object.getClass().isArray()) {

map = new DefaultSqlSession.StrictMap();

map.put("array", object);

return map;

} else {

return object;

}

}

public static class StrictMap<V> extends HashMap<String, V> {

private static final long serialVersionUID = -5741767162221585340L;

public StrictMap() {

}

public V get(Object key) {

if (!super.containsKey(key)) {

throw new BindingException("Parameter '" + key + "' not found. Available parameters are " + this.keySet());

} else {

return super.get(key);

}

}

}

}

### Executor

DefaultSqlSessionFactory是从Configuration里获取Executor的，来看一下：

package org.apache.ibatis.session;

……

public class Configuration {

public Configuration() {

……

this.cacheEnabled = true;

……

}

……

public Executor newExecutor(Transaction transaction, ExecutorType executorType) {

executorType = executorType == null ? this.defaultExecutorType : executorType;

executorType = executorType == null ? ExecutorType.SIMPLE : executorType;

Object executor;

if (ExecutorType.BATCH == executorType) {

executor = new BatchExecutor(this, transaction);

} else if (ExecutorType.REUSE == executorType) {

executor = new ReuseExecutor(this, transaction);

} else {

executor = new SimpleExecutor(this, transaction);

}

if (this.cacheEnabled) {

executor = new CachingExecutor((Executor)executor);

}

Executor executor = (Executor)this.interceptorChain.pluginAll(executor);

return executor;

}

……

}

默认是开启一级缓存的，也没办法关掉一级缓存。二级缓存的开关是cacheEnabled，默认为true。因此上面的代码会返回一个CachingExecutor，由它来管理二级缓存。

### MapperRegistry

Mybatis的一个核心是MapperRegistry，它在里面有一个private final Map<Class<?>, MapperProxyFactory<?>> knownMappers = new HashMap()，这个knownMappers缓存了xml配置的Bean和@Mapper配置的Bean，通过类名获取对应的MapperProxyFactory对象。调用addMapper方法时，通过创建MapperProxyFactory的newInstance方法对应的MapperProxyFactory对象。调用getMapper方法时，从knownMappers中取出MapperProxyFactory对象。

package org.apache.ibatis.binding;

import java.util.Collection;

import java.util.Collections;

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

import java.util.Set;

import org.apache.ibatis.builder.annotation.MapperAnnotationBuilder;

import org.apache.ibatis.io.ResolverUtil;

import org.apache.ibatis.io.ResolverUtil.IsA;

import org.apache.ibatis.session.Configuration;

import org.apache.ibatis.session.SqlSession;

public class MapperRegistry {

private final Configuration config;

private final Map<Class<?>, MapperProxyFactory<?>> knownMappers = new HashMap();

public MapperRegistry(Configuration config) {

this.config = config;

}

public <T> T getMapper(Class<T> type, SqlSession sqlSession) {

MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory)this.knownMappers.get(type);

if (mapperProxyFactory == null) {

throw new BindingException("Type " + type + " is not known to the MapperRegistry.");

} else {

try {

return mapperProxyFactory.newInstance(sqlSession);

} catch (Exception var5) {

throw new BindingException("Error getting mapper instance. Cause: " + var5, var5);

}

}

}

public <T> boolean hasMapper(Class<T> type) {

return this.knownMappers.containsKey(type);

}

public <T> void addMapper(Class<T> type) {

if (type.isInterface()) {

if (this.hasMapper(type)) {

throw new BindingException("Type " + type + " is already known to the MapperRegistry.");

}

boolean loadCompleted = false;

try {

this.knownMappers.put(type, new MapperProxyFactory(type));

MapperAnnotationBuilder parser = new MapperAnnotationBuilder(this.config, type);

parser.parse();

loadCompleted = true;

} finally {

if (!loadCompleted) {

this.knownMappers.remove(type);

}

}

}

}

public Collection<Class<?>> getMappers() {

return Collections.unmodifiableCollection(this.knownMappers.keySet());

}

public void addMappers(String packageName, Class<?> superType) {

ResolverUtil<Class<?>> resolverUtil = new ResolverUtil();

resolverUtil.find(new IsA(superType), packageName);

Set<Class<? extends Class<?>>> mapperSet = resolverUtil.getClasses();

Iterator i$ = mapperSet.iterator();

while(i$.hasNext()) {

Class<?> mapperClass = (Class)i$.next();

this.addMapper(mapperClass);

}

}

public void addMappers(String packageName) {

this.addMappers(packageName, Object.class);

}

}

### MapperProxyFactory

MapperProxyFactory很简单，就是new MapperProxyFactory的时候指定Mapper的mapperInterface，即我们声明的方法。外部调用MapperProxyFactory的newInstance方法时，它就根据传入的sqlSession和之前传入的mapperInterface创建一个MapperProxy，这个MapperProxy使用MapperProxyFactory的methodCache。

package org.apache.ibatis.binding;

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

import java.util.Map;

import java.util.concurrent.ConcurrentHashMap;

import org.apache.ibatis.session.SqlSession;

public class MapperProxyFactory<T> {

private final Class<T> mapperInterface;

private final Map<Method, MapperMethod> methodCache = new ConcurrentHashMap();

public MapperProxyFactory(Class<T> mapperInterface) {

this.mapperInterface = mapperInterface;

}

public Class<T> getMapperInterface() {

return this.mapperInterface;

}

public Map<Method, MapperMethod> getMethodCache() {

return this.methodCache;

}

protected T newInstance(MapperProxy<T> mapperProxy) {

return Proxy.newProxyInstance(this.mapperInterface.getClassLoader(), new Class[]{this.mapperInterface}, mapperProxy);

}

public T newInstance(SqlSession sqlSession) {

MapperProxy<T> mapperProxy = new MapperProxy(sqlSession, this.mapperInterface, this.methodCache);

return this.newInstance(mapperProxy);

}

}

### MapperProxy

MapperProxy被MapperProxyFactory创建出来的时候，被设置了sqlSession、mapperInterface、methodCache，当Mapper Bean被执行的时候，动态代理就会调用MapperProxy的invoke方法，invoke方法里判断调用的是Object方法还是Mapper Interface方法，是Object方法就执行MapperProxy的Object方法。是Mapper Interface方法就创建MapperMethod对象，然后把MapperMethod对象缓存在methodCache中，下次调用直接从缓存里取，最后调用MapperMethod的execute方法。

package org.apache.ibatis.binding;

import java.io.Serializable;

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.util.Map;

import org.apache.ibatis.reflection.ExceptionUtil;

import org.apache.ibatis.session.SqlSession;

public class MapperProxy<T> implements InvocationHandler, Serializable {

private static final long serialVersionUID = -6424540398559729838L;

private final SqlSession sqlSession;

private final Class<T> mapperInterface;

private final Map<Method, MapperMethod> methodCache;

public MapperProxy(SqlSession sqlSession, Class<T> mapperInterface, Map<Method, MapperMethod> methodCache) {

this.sqlSession = sqlSession;

this.mapperInterface = mapperInterface;

this.methodCache = methodCache;

}

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

if (Object.class.equals(method.getDeclaringClass())) {

try {

return method.invoke(this, args);

} catch (Throwable var5) {

throw ExceptionUtil.unwrapThrowable(var5);

}

} else {

MapperMethod mapperMethod = this.cachedMapperMethod(method);

return mapperMethod.execute(this.sqlSession, args);

}

}

private MapperMethod cachedMapperMethod(Method method) {

MapperMethod mapperMethod = (MapperMethod)this.methodCache.get(method);

if (mapperMethod == null) {

mapperMethod = new MapperMethod(this.mapperInterface, method, this.sqlSession.getConfiguration());

this.methodCache.put(method, mapperMethod);

}

return mapperMethod;

}

}

### MapperMethod

创建MapperMethod的时候，给了它mapperInterface、method、config，即是哪个Mapper Interface、Mapper Interface的哪个method、SQLSession的Configuration，MapperMethod通过这三个参数构造MapperMethod.SqlCommand和MapperMethod.MethodSignature，作为自己的command和method。

package org.apache.ibatis.binding;

……

public class MapperMethod {

private final MapperMethod.SqlCommand command;

private final MapperMethod.MethodSignature method;

public MapperMethod(Class<?> mapperInterface, Method method, Configuration config) {

this.command = new MapperMethod.SqlCommand(config, mapperInterface, method);

this.method = new MapperMethod.MethodSignature(config, mapperInterface, method);

}

……

}

MapperMethod.SqlCommand从configuration里获取MappedStatement，MappedStatement里有type（INSERT, UPDATE, DELETE, SELECT, FLUSH）、name。

public static class SqlCommand {

private final String name;

private final SqlCommandType type;

public SqlCommand(Configuration configuration, Class<?> mapperInterface, Method method) {

String statementName = mapperInterface.getName() + "." + method.getName();

MappedStatement ms = null;

if (configuration.hasStatement(statementName)) {

ms = configuration.getMappedStatement(statementName);

} else if (!mapperInterface.equals(method.getDeclaringClass())) {

String parentStatementName = method.getDeclaringClass().getName() + "." + method.getName();

if (configuration.hasStatement(parentStatementName)) {

ms = configuration.getMappedStatement(parentStatementName);

}

}

if (ms == null) {

if (method.getAnnotation(Flush.class) == null) {

throw new BindingException("Invalid bound statement (not found): " + statementName);

}

this.name = null;

this.type = SqlCommandType.FLUSH;

} else {

this.name = ms.getId();

this.type = ms.getSqlCommandType();

if (this.type == SqlCommandType.UNKNOWN) {

throw new BindingException("Unknown execution method for: " + this.name);

}

}

}

public String getName() {

return this.name;

}

public SqlCommandType getType() {

return this.type;

}

}

mapper中的namespace是用来绑定dao接口的，即面向接口编程。当你的namespace绑定接口后，你可以不用写接口实现类，mybatis会通过该绑定自动帮你找到对应要执行的SQL语句在同一次请求中不允许出现相同名称的方法、类和常量，但是在某些特殊的应用中必须要使用相同名称的方法、类和常量，需要把他们放到不同的空间里，这个空间就是命名空间。命名空间主要是为了解决命名冲突问题确保方法名称的唯一性，如果两个xml文件中的方法名一样，那么就用namespace区分。

总结：

namespace就是为了解决项目中名称重复的问题

因为在大型项目中，你可能会引用别的代码或者子项目等等，而你不知道它里面是否有与你定义的名称一模一样的类名、方法名、常量名，或者你已经词穷，不知道该用什么词来定义词意相同的但实际上并不是同一个类、方法或常量的名字此时就可以用namespace来避免这样的尴尬。

MySQL的FLUSH句法：FLUSH flush\_option [,flush\_option]

如果你想要清除一些MySQL使用内部缓存，你应该使用FLUSH命令。为了执行FLUSH，你必须有reload权限。

**Flush HOSTS**：这个用的最多，经常碰见。主要是用来清空主机缓存表。如果你的某些主机改变IP数字，或如果你得到错误消息Host ... isblocked，你应该清空主机表。当在连接MySQL服务器时，对一台给定的主机有多于 max\_connect\_errors个错误连续不断地发生，MySQL为了安全的需要将会阻止该主机进一步的连接请求。清空主机表允许主机再尝试连接。

**Flush LOGS：**关闭当前的二进制日志文件并创建一个新文件，新的二进制日志文件的名字在当前的二进制文件的编号上加1。

**Flush PRIVILEGES：**这个也是经常使用的，每当重新赋权后，为了以防万一，让新权限立即生效，一般都执行一把，目地是从数据库授权表中重新装载权限到缓存中。

**Flush TABLES：**关闭所有打开的表，同时该操作将会清空查询缓存中的内容。FLUSH TABLES WITH READ LOCK：关闭所有打开的表，同时对于所有数据库中的表都加一个读锁，直到显示地执行unlock tables，该操作常常用于数据备份的时候。解锁的语句就是unlock tables。

**Flush MASTER：**删除所有的二进制日志索引文件中的二进制日志文件，重置二进制日志文件的索引文件为空，创建一个新的二进制日志文件,不过这个已经不推荐使用，改成reset master 了。

**Flush SLAVE：**类似于重置复制吧，让从数据库忘记主数据库的复制位置，同时也会删除已经下载下来的relay log,与Master一样，已经不推荐使用，改成Reset Slave了。

一般来讲，Flush操作都会记录在二进制日志文件中，但是FLUSH LOGS、FLUSH MASTER、FLUSH SLAVE、FLUSH TABLES WITH READ LOCK不会记录，因此上述操作如果记录在二进制日志文件中话，会对从数据库造成影响。

# Zookeeper

## 节点状态

服务器具有四种状态，分别是LOOKING、FOLLOWING、LEADING、OBSERVING。

Leader挂后，余下的非Observer服务器都会讲自己的服务器状态变更为LOOKING，然后开始进入Leader选举过程。

## 节点个数

Zookeeper 集群节点为什么要部署成奇数

无论奇偶数都可以选举leader。例如5台zookeeper节点机器最多宕掉2台，还可以继续使用，因为剩下3台大于5/2。至于为什么最好为奇数个节点？这样是为了以最大容错服务器个数的条件下，能节省资源。比如，最大容错为2的情况下，对应的zookeeper服务数，奇数为5，而偶数为6，也就是6个zookeeper服务的情况下最多能宕掉2个服务，所以从节约资源的角度看，没必要部署6（偶数）个zookeeper服务节点。

## 如何避免脑裂问题

假设某个leader假死，其余的followers选举出了一个新的leader。这时，旧的leader复活并且仍然认为自己是leader，这个时候它向其他followers发出写请求也是会被拒绝的。因为每当新leader产生时，会生成一个epoch标号(标识当前属于那个leader的统治时期)，这个epoch是递增的，followers如果确认了新的leader存在，知道其epoch，就会拒绝epoch小于现任leader epoch的所有请求。那有没有follower不知道新的leader存在呢，有可能，但肯定不是大多数，否则新leader无法产生。Zookeeper的写也遵循quorum机制，因此，得不到大多数支持的写是无效的，旧leader即使各种认为自己是leader，依然没有什么作用。

## 因果一致性

Zookeeper是否满足因果一致性，需要看客户端的编程方式。

不满足因果一致性的做法

1. A进程向Zookeeper的/z写入一个数据，成功返回

2. A进程通知B进程，A已经修改了/z的数据

3. B读取Zookeeper的/z的数据

4. 由于B连接的Zookeeper的服务器有可能还没有得到A写入数据的更新，那么B将读不到A写入的数据

满足因果一致性的做法

1. B进程监听Zookeeper上/z的数据变化

2. A进程向Zookeeper的/z写入一个数据，成功返回前，Zookeeper需要调用注册在/z上的监听器，Leader将数据变化的通知告诉B

3. B进程的事件响应方法得到响应后，去取变化的数据，那么B一定能够得到变化的值

4. 这里的因果一致性提现在Leader和B之间的因果一致性，也就是是Leader通知了数据有变化

第二种事件监听机制也是对Zookeeper进行正确编程应该使用的方法，所以，Zookeeper是满足因果一致性的

## Zookeeper不是强一致性

强一致性指的是你在一个副本节点修改了数据，那么在其他副本节点都能立刻读到最新修改的数据。

# Kafka

## 如何保证数据不丢失？

**对于生产者：**

需要修改配置“producer.type=sync、request.required.acks=1”，让至少一个Follower跟上Leader的数据同步，如果没有一个Follower数据同步成功，就说写入失败，让生产系统重新尝试写入。这样leader宕机，触发leader切换时，就不会丢失数据。但是写入性能就差了。

Leader 会跟踪与其保持同步的 Replica 列表，该列表称为 ISR（即 in-sync Replica）。如果一个 Follower 宕机，或者落后太多，Leader 将把它从 ISR 中移除。

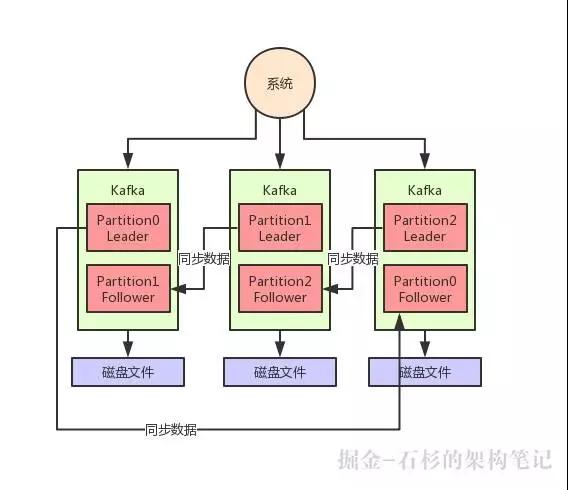
ack机制：在kafka发送数据的时候，每次发送消息都会有一个确认反馈机制，确保消息正常的能够被收到。

**对于消费者：**

kafka自己记录了每次消费的offset数值。如果消费者消费了数据，但是不commit，这样offset不会加1，下次继续消费的时候，还是消费之前的offset，这样就不会丢失数据。消费完了，commit offset，这样就不会重复消费。

## 如何保证高可用？

保存一份副本冗余在其他机器上。



## 如何保证顺序？

kafka只能保证**同一个partition内消息有序，不能保证不同partition之间有序**，因此业务只能充分利用单个partition内有序这个特点。

如果Kafka要保证多个partition有序，不仅broker保存的数据要保持顺序，消费时也要按序消费。假设partition1堵了，为了有序，那partition2以及后续的分区也不能被消费，这种情况下，Kafka 就退化成了单一队列，毫无并发性可言，极大降低系统性能。

Kafka 中发送1条消息的时候，可以指定(topic, partition, key) 3个参数。partiton 和 key 是可选的。如果你指定了 partition，那就是所有消息发往同1个 partition，就是有序的。并且在消费端，Kafka 保证，1个 partition 只能被1个 consumer 消费。或者你指定 key（比如 order id），具有同1个 key 的所有消息，会发往同1个 partition。也是有序的。

**全局有序：**

（1）全局使用一个生产者

（2）全局使用一个消费者（消费者内部自己维护有序，比如有些场景可以将消息分为n个queue，每个queue一个线程）

（3）全局使用一个分区（当然不同的表可以使用不同的分区或者topic实现隔离与扩展）

**局部有序：**

局部有序是指在某个业务功能场景下保证消息的发送和接收顺序是一致的。如：订单场景，要求订单的创建、付款、发货、收货、完成消息在同一订单下是有序发生的，即消费者在接收消息时需要保证在接收到订单发货前一定收到了订单创建和付款消息。

**可以在producer往kafka插入数据时控制，同一key分发到同一partition上面。但是会导致某个partition消息很多。**

**严格有序：**

对于一个有着先后顺序的消息A、B，正常情况下应该是A先发送完成后再发送B，但是在异常情况下，在A发送失败的情况下，B发送成功，而A由于重试机制在B发送完成之后重试发送成功了。

针对以上的问题，严格的顺序消费还需要以下参数支持：**max.in.flight.requests.per.connection=1**，保证在后一条消息发送前，前一条的消息状态已经是可知的。

## 如何保证消息不重复？

需要以下参数支持：**max.in.flight.requests.per.connection=1**，保证在后一条消息发送前，前一条的消息状态已经是可知的。

但是如果Producer发送数据给broker后，遇到网络问题而造成通信中断，丢失broker返回的ack，那Producer就会觉得消息没写入成功。为了解决这样的问题，Producer生成一种类似于主键的东西，发生故障时幂等性的重试多次，这样就做到了消息只写入一次。

具体实现为：Kafka引入Producer ID（即PID）和Sequence Number。对于每个PID，该Producer发送消息的每个<Topic, Partition>都对应一个单调递增的Sequence Number。同样，Broker端也会为每个<PID, Topic, Partition>维护一个序号，并且每Commit一条消息时将其对应序号递增。对于接收的每条消息，如果其序号比Broker维护的序号）大一，则Broker会接受它，否则将其丢弃。该机制就是乐观锁。

（1）如果消息序号比Broker维护的序号差值比一大，说明中间有数据尚未写入，即乱序，此时Broker拒绝该消息，Producer抛出InvalidSequenceNumber

（2）如果消息序号小于等于Broker维护的序号，说明该消息已被保存，即为重复消息，Broker直接丢弃该消息，Producer抛出DuplicateSequenceNumber

Kafka会为每一个Consumer Group保留一些metadata信息——当前消费的消息的position，也即offset。这个offset由Consumer控制。正常情况下Consumer会在消费完一条消息后递增该offset。当然，Consumer也可将offset设成一个较小的值，重新消费一些消息。因为offset由Consumer控制，所以Kafka broker是无状态的，它不需要标记哪些消息被哪些消费过。

同一Topic的一条消息只能被同一个Consumer Group内的一个Consumer消费。但多个Consumer Group可同时消费这一消息。

这是Kafka用来实现一个Topic消息的广播（发给所有的Consumer）和单播（发给某一个Consumer）的手段。一个Topic可以对应多个Consumer Group。如果需要实现广播，只要每个Consumer有一个独立的Group就可以了。要实现单播只要所有的Consumer在同一个Group里。

Consumer在从broker读取消息后，可以选择commit，该操作会在Zookeeper中保存该Consumer在该Partition中读取的消息的offset。该Consumer下一次再读该Partition时会从下一条开始读取。如未commit，下一次读取的开始位置会跟上一次commit之后的开始位置相同。当然可以将Consumer设置为autocommit，即Consumer一旦读到数据立即自动commit。

<https://www.cnblogs.com/frankdeng/p/9310684.html>

## Topic和Partition

Topic在逻辑上可以被认为是一个queue，每条消费都必须指定它的Topic，可以简单理解为必须指明把这条消息放进哪个queue里。为了使得Kafka的吞吐率可以线性提高，物理上把Topic分成一个或多个Partition，每个Partition在物理上对应一个文件夹，该文件夹下存储这个Partition的所有消息和索引文件。创建一个topic时，同时可以指定Partition数目，Partition数越多，其吞吐量也越大，但是需要的资源也越多，同时也会导致更高的不可用性，kafka在接收到生产者发送的消息之后，会根据均衡策略将消息存储到不同的Partition中。因为每条消息都被append到该Partition中，属于顺序写磁盘，因此效率非常高（经验证，顺序写磁盘效率比随机写内存还要高，这是Kafka高吞吐率的一个很重要的保证）。

可以在$KAFKA\_HOME/config/server.properties中通过配置项num.partitions来指定新建Topic的默认Partition数量，也可在创建Topic时通过参数指定，同时也可以在Topic创建之后通过Kafka提供的工具修改。

## Kafka事务

<https://www.cnblogs.com/middleware/p/9477133.html>

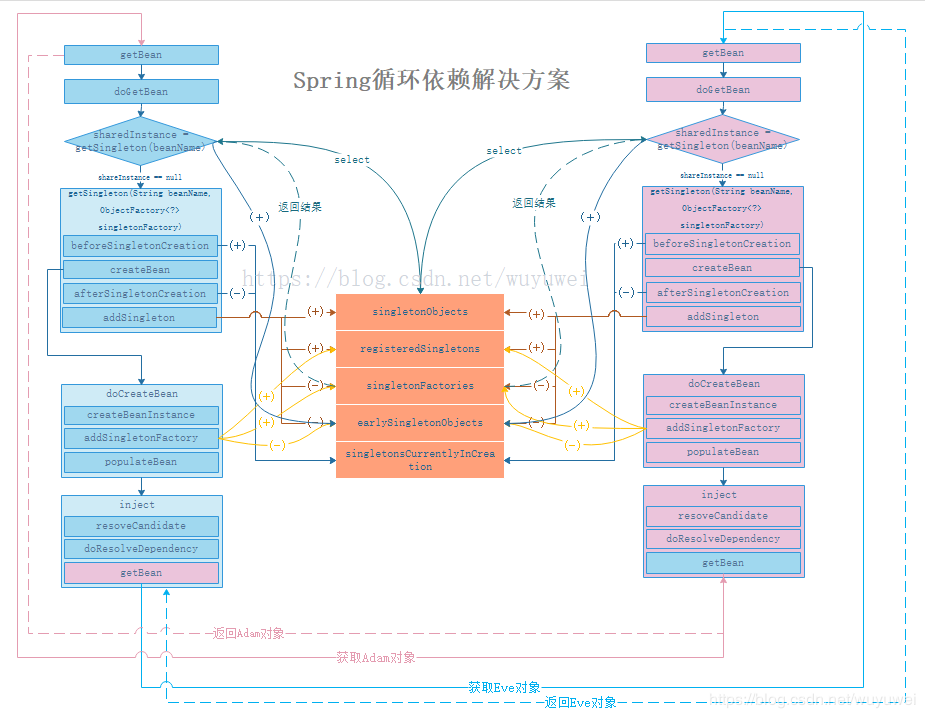
<https://www.cnblogs.com/wangzhuxing/p/10125437.html>

<https://zhuanlan.zhihu.com/p/163683403>

<https://www.zhihu.com/question/311885878/answer/596739819>

# Spring

## Spring解决循环依赖



假设目前“亚当”和“夏娃”这两个bean都还未曾创建，现在流程走到亚当的创建流程（蓝色流程）。因为亚当创建之前不曾有亚当创建出来被容器托管，故流程会很顺利的走到createBean。在createBean之前，会通过beforeSingletonCreation在bean工厂的singletonsCurrentlyInCreation Set集合中添加亚当这个bean的beanName——“yadang”。

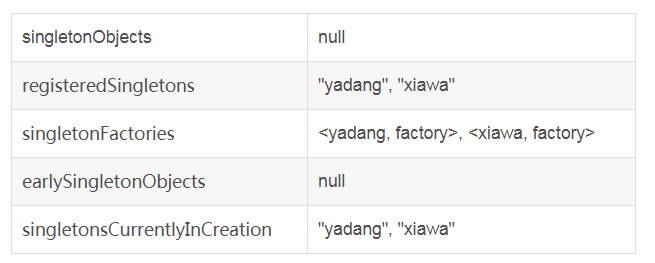
随后流程陷入doCreateBean方法中，首先通过createBeanInstance创建出亚当这个对象adamObj，请注意目前**亚当的xiawa是null**。

接着流程通过addSingletonFactory，如果在singletonObjects中没有找到“yadang”这个对象，那么就尝试在bean工厂的singletonFactories中进行一些步骤，在bean工厂的各个缓存容器加入yadang，如下图。



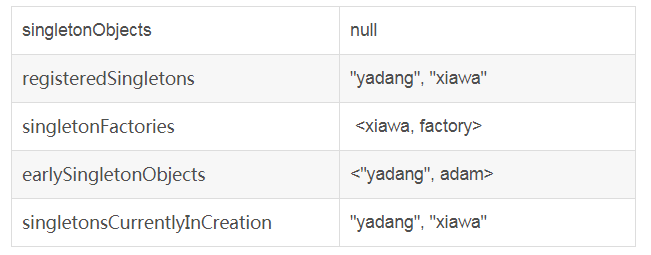
亚当创建完成后，就需要通过populateBean对属性进行依赖注入，流程会陷入inject方法，最终通过doResolveDependency一路调用到getBean(“xiawa”)。这时流程就走到了夏娃的创建流程（粉色流程）。

经过同样的步骤，xiawa也变成这样。



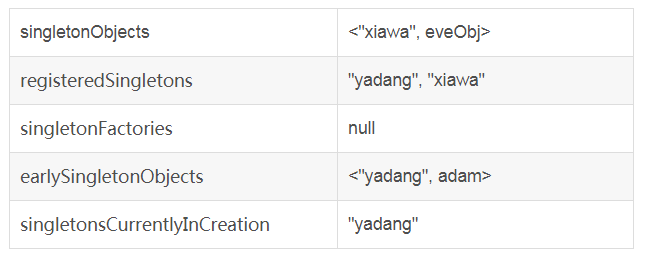
同样的，夏娃创建完成后，就需要通过populateBean对属性进行依赖注入，流程会陷入inject方法，最终通过doResolveDependency一路调用到getBean(“yadang”)。这时流程又就走到了亚当的创建流程（蓝色流程）。

再次进入蓝色的getBean-->doGetBean-->getSingleton("yadang")-->getSingleton("yadang"，true)。根据bean工厂中各个缓存容器的值层层过滤后，最终满足所有条件。此时bean工厂中各个缓存容器的值如下：

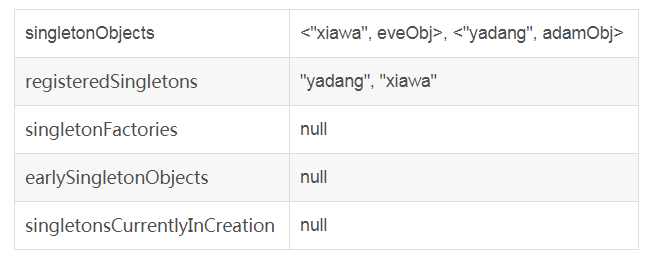


最终通过几步简单的调用步骤后，返回最终获取到的adam对象，红色doResolveDependency方法的getBean方法执行完成执行后一路返回，最终在inject中完成夏娃对象adam属性的装配，这是eveObj对象：**她的yadang得到了yadang对象**。

完成eve对象的装配后一路返回doGetBean方法。随后通过afterSingletonCreation方法移除singletonsCurrentlyInCreation中的中的“xiawa”。这样bean工厂中各个缓存容器的值如下：



这样，夏娃的创建过程就完毕了。流程一路返回eveObj对象到蓝色的doResolveDependency方法。一路返回eveObj对象到inject完成adamObj的装配。随后一路返回到getSingleton(String beanName, ObjectFactory<?> singletonFactory)方法的afterSingletonCreation和addSingleton方法操作bean工厂中各个缓存容器后的值如下：



此时亚当和夏娃这两个bean都缓存在了singletonObjects中，完成了这两个循环依赖bean的创建过程。

## IOC

IOC是什么，怎么实现的。

SpringIOC的方式有哪几种？

Required是什么意思怎么用

# C语言

## Sizeof

在C语言中，sizeof是一个操作符（operator），而不是函数！其用于判断数据类型或者表达式长度（所占的内存字节数）。

不管在32位系统还是64位系统，sizeof(int)的值都是4。因为sizeof的结果是由编译器(或者说是编译选项)决定的，不是操作系统。所以sizeof(int)是个特例。

但是在32位系统上sizeof(指针)的值是4，在64位系统上sizeof(指针)的值是8。

32位系统：



#include <stdio.h>

int main(void)

{

char \*s = "hello";

printf("sizeof(char) = %u\n", sizeof(char));

printf("sizeof(char\*)= %u\n", sizeof(char\*));

printf("sizeof('a') = %u\n", sizeof('a'));

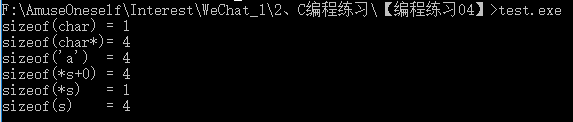
printf("sizeof(\*s+0) = %u\n", sizeof(\*s+0));

printf("sizeof(\*s) = %u\n", sizeof(\*s));

printf("sizeof(s) = %u\n", sizeof(s));

return 0;

}



其中sizeof('a')=4是因为这里的’a’取的是ascii码，是一个整数。如果定义char a = ‘a’则sizeof(a)=1。

其中sizeof(\*s+0)=4是因为结果转换成了int。

其中sizeof(s)=4是因为此处s是一个指针。

<https://blog.csdn.net/zhengnianli/article/details/84076264>

对于64位系统，区别如下：



<http://www.myexceptions.net/c/1969634.html>

# GO

## Slice

Slice不是线程安全的。

程序示例：

package main

import (

"fmt"

"sync"

)

var list []int = []int{}

var wgList sync.WaitGroup = sync.WaitGroup{}

func addNotSafe() {

list = append(list, 1)

wgList.Done()

}

func main() {

max := 10000

wgList.Add(max)

for i := 0; i < max; i++ {

go addNotSafe()

}

wgList.Wait()

fmt.Printf("list len = %d", len(list))

}

输出结果：

9217

为什么输出结果不是10000呢？因为slice不是线程安全的。对于多核CPU，假设cpu1执行线程A，将slice的长度从1变成了2，同时cpu2执行线程B，也将slice的长度从1变成了2，因为两个cpu写内存时产生了覆盖，导致slice的长度少写了1。

<https://www.imooc.com/article/67836/>

# Git

## 配置

配置个人的用户名称和电子邮件地址：

git config --global user.name "runoob"

git config --global user.email test@runoob.com

如果用了 --global 选项，那么更改的配置文件就是位于主目录下的那个，以后所有的项目都会默认使用这里配置的用户信息。

如果要在某个特定的项目中使用其他名字或者电邮，只要去掉 --global 选项重新配置即可，新的设定保存在当前项目的 .git/config 文件里。

要检查已有的配置信息，可以使用 git config --list 命令：

git config --list

## 查看Git版本

git --version

## 概念和操作

HEAD和master是一个概念，命令中的HEAD可以用 master 来替换

Git 使用 git init 命令来初始化一个 Git 仓库：

git init newrepo

添加

git add \*.c

或git add README

git commit -m '初始化项目版本'

其中-m表示要添加message

克隆仓库的命令格式为：

git clone <repo>

如果我们需要克隆到指定的目录，可以使用以下命令格式：

git clone <repo> <directory>

几种效果等价的git clone写法：

git clone http://github.com/CosmosHua/locate new

git clone http://github.com/CosmosHua/locate.git new

git clone git://github.com/CosmosHua/locate new

git clone git://github.com/CosmosHua/locate.git new

新项目中，添加所有文件很普遍，我们可以使用 git add . 命令来添加当前项目的所有文件

git add .

git status 以查看在你上次提交之后是否有修改。

演示该命令的时候加了 -s 参数，以获得简短的结果输出。如果没加该参数会详细输出内容。

git status -s

git diff 命令显示已写入缓存与已修改但尚未写入缓存的改动的区别。git diff 有两个主要的应用场景。

尚未缓存的改动：git diff

查看已缓存的改动： git diff --cached

# 哈夫曼树、哈夫曼编码

哈夫曼编码用于找出存放一串字符所需的最少的二进制编码。

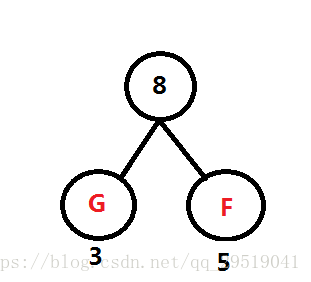
例如一串字符只包含ABCDEFG这7个字符，各字符出现的频率为

频率表 D:69 A：60 B:45 E:14 C:13 F:5 G:3

**下面我们构造哈夫曼树：**

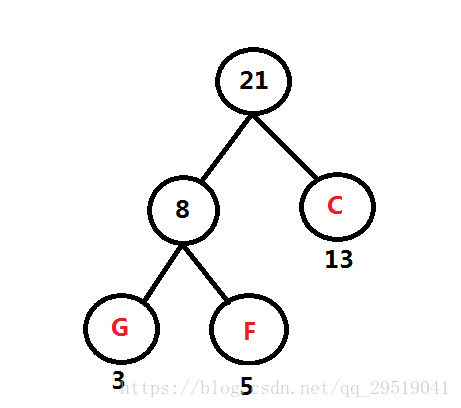
**第一步：**找出字符中最小的两个，小的在左边，大的在右边，组成二叉树。在频率表中删除此次找到的两个数，并加入此次最小两个数的频率和。

F和G最小，因此如图，从字符串频率计数中删除F与G，并返回G与F的和 8给频率表



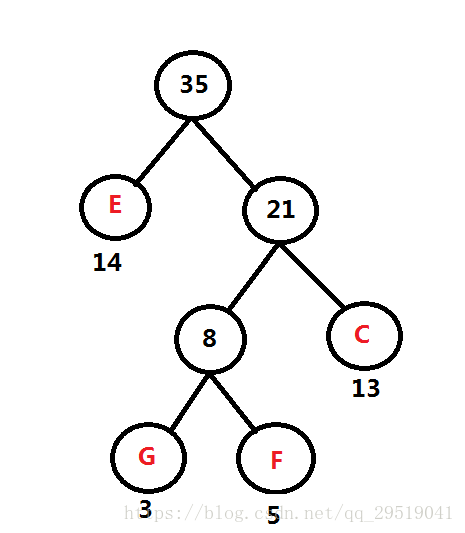
**此时频率表变为D:69 A：60 B:45 E:14 C:13 FG:8**

**然后重复第一步**



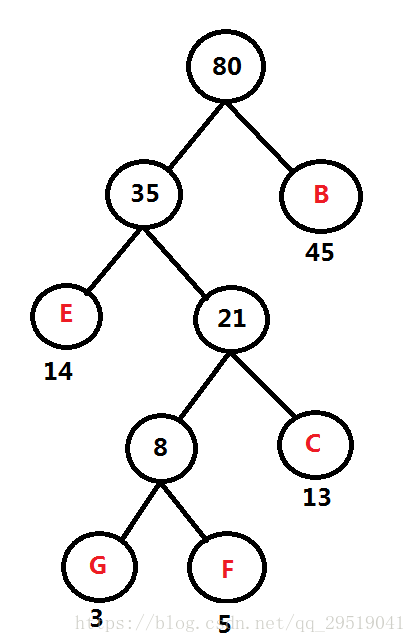
此时频率表变为 D: 69 A：60 B: 45 CFG: 21 E: 14

然后重复第一步



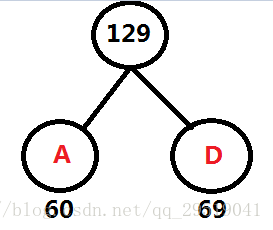
此时频率表变为D: 69 A：60 B: 45 ECFG: 35

然后重复第一步



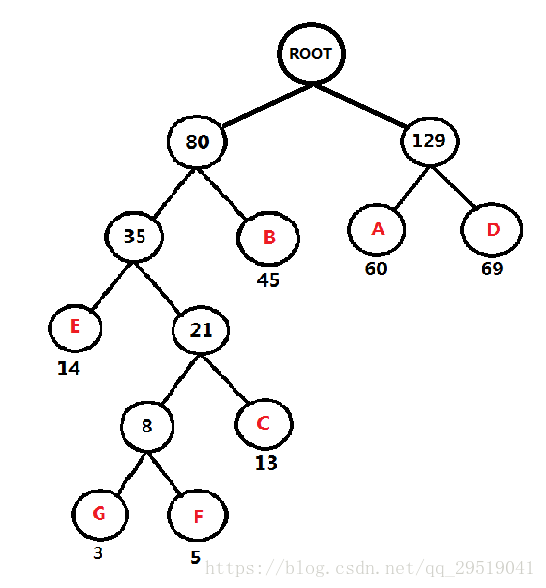
此时频率表变为BECFG: 80 D: 69 A：60

然后重复第一步



此时频率表变为DA: 129 BECFG: 80

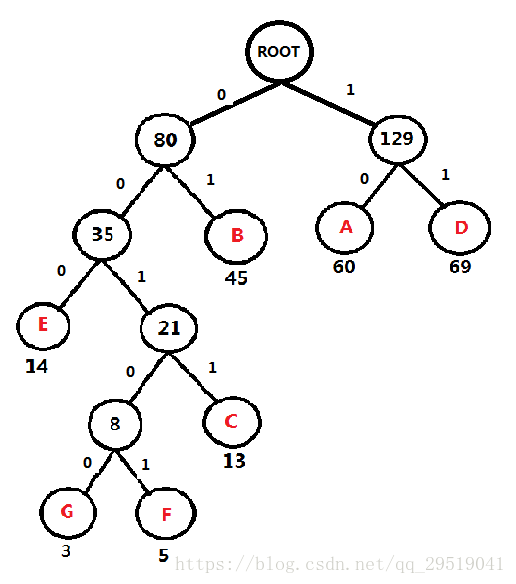
然后重复第一步



此时频率表变DABECFG: 209

**哈夫曼树构造完成**

**然后左0右1编码**



得到：

A 10

B 01

C 0011

D 11

E 000

F 00101

G 00100

**因为哈夫曼树的它的字母都在叶子节点上，因此不会出现一个字母的编码为另一个字母编码左起子串的情况。也就不会出现解码冲突、解码矛盾的情况了，解码就不会误判了。**

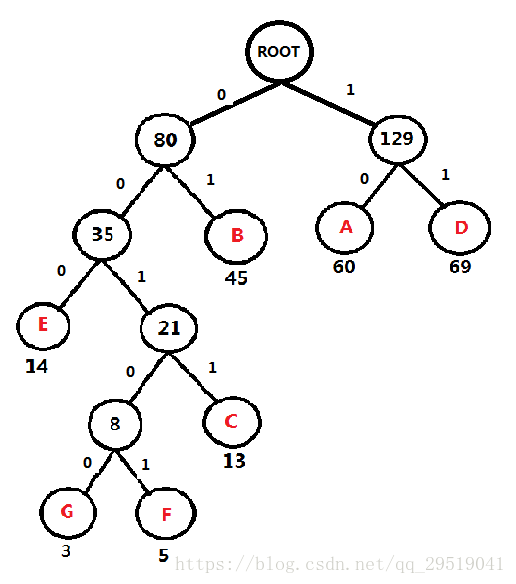
**哈夫曼树的构造并不是唯一的。**因为如果有两个字符的出现频率相等，或者构造时合并的频率出现相等，则取最小的两个数，可能只能取到其中一个。例如A：8，B：8，C：7，可以取BC合并编码，或者AC合并编码，不能唯一确定。或者A：8，B：7，C：3，D：5，编码后变成A：8，DC：8，B：7，继续编码时，可以取DCB编码，也可以取AB编码，得到的结果是不一样的。

**哈夫曼树(Huffman)树又称最优二叉树,是带权路径长度最短的二叉树。**

带权路径计算为：

WPL = W1L1 + W2L2 + ...... + WnLn

其中n为二叉树中叶子结点的个数；Wk为第k个叶子的权值;Lk为第k个叶子结点的路径长度



例如上图，A的权值为60，路径长度为到根节点需要的编码数量，即路径长度为2。F的权值为5，路径长度为5

**为什么哈夫曼树(Huffman)树带权路径长度最短？因为权重大的节点距离根节点最近。**

**节点都是叶子节点的情况下，哈夫曼树的带权路径长度最短。**

# 二叉树

## 遍历

二叉树遍历分为三种：前序、中序、后序

**前序：根左右**

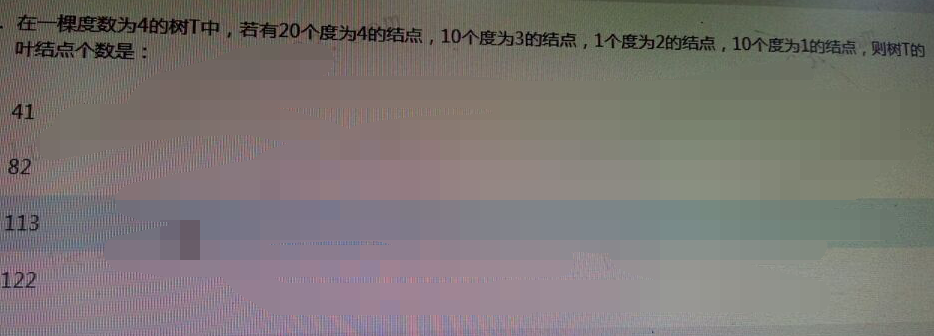
**中序：左根右**

**后序：左右根**

**前序后序不能还原二叉树。前序中序可以还原二叉树。中序后序可以还原二叉树。**

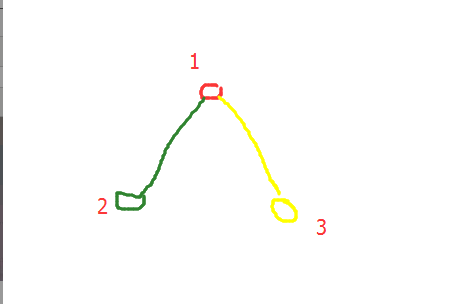
**还原技巧：前序的第一个值就是根节点，后序的最后一个值就是根节点，用根节点来把中序切割为根节点的左子树、根节点的右子树。然后再根据中序的分割把前序、后序分割成左子树、右子树。然后依次方法类推。**

## 节点、度



对于这一题，可以这样看：

**（1）总结点数=树中的边数+1** 。见下图可以清晰明白。



**节点的度就是结点拥有子结点的数量，也就是边的数量。**

所以题中树的总结点数=20\*4+10\*3+1\*2+10\*1+1=123

（2）叶子节点数（叶子节点度为0）=总结点数-度数非零的节点数

叶节点是度为0的节点

所以题中树的叶子节点数=123-20-10-1-10=82

对于二叉树，如果知道了总结点数，叶子结点数，则可以求出度为2的节点数、度为1的节点数。因为：

总结点数 - 1 = 2 \* 度为2的节点 + 1 \* 度为1的节点

总结点数 - 叶子结点数 = 度为2的节点 + 度为1的节点

以上二元一次方程简化，**即度为2的节点数等于叶子结点数减一，度为1的节点数等于总结点数减去度2节点数、叶子节点数。**

## 满二叉树

一棵二叉树的结点要么是叶子结点，要么它有两个子结点。没有度为1的节点。

## 完全二叉树

若设二叉树的深度为k，除第 k 层外，其它各层 (1～k-1) 的结点数都达到最大个数，第k 层所有的结点都连续集中在最左边，这就是完全二叉树。

**完全二叉树实质上就是一个连续数组。数组的存放顺序就是完全二叉树的广度遍历（层次遍历）。**

例如完全二叉树，根节点为0，根节点左孩子为1，右孩子为2；左孩子1的左孩子为3，右孩子为4；右孩子2的左孩子为5，右孩子为6；左孩子3的左孩子为7，右孩子为8。这个完全二叉树的广度遍历得到数组[0,1,2,3,4,5,6,7,8]。对于数组中位置在i的节点，如果该节点有左孩子和右孩子，则左孩子位置必然在2\*i+1，右孩子位置必然在2\*i+2。

**完全二叉树用数组存储即可，不需要复杂的数据结构。**

## 二叉查找树、二叉搜索树、二叉排序树

二叉排序树（Binary Sort Tree），又称二叉查找树（Binary Search Tree），简称BST

**特点：**

若左子树不空，则左子树上所有结点的值均小于或等于它的根结点的值；

若右子树不空，则右子树上所有结点的值均大于或等于它的根结点的值；

左、右子树也分别为二叉排序树

**查找：**

如果树是空的，则查找结束，无匹配。

如果被查找的值和根结点的值相等，查找成功。

如果被查找的值小于根结点的值就选择左子树。

如果被查找的值大于根结点的值就选择右子树。

**插入：**

先查找树中是否存在元素，**如果存在则不插入**。

如果插入的元素值小于根结点值，（1）根节点左子树为空，插入元素作为根节点左子树。（2）根节点左子树不为空，递归。

如果插入的元素值大于根结点值，（1）根节点右子树为空，插入元素作为根节点右子树。（2）根节点右子树不为空，递归。

**删除：**

根节点为空，直接返回。

要删除的元素值等于根节点的值，执行节点删除。

要删除的元素值小于根节点的值，将删除递归到根节点的左子树。

要删除的元素值大于根节点的值，将删除递归到根节点的右子树。

**节点删除：**

如果节点的右子树为空，则用节点的左子树（左子树可能为null）替代当前节点，然后释放当前节点。

如果节点的左子树为空，则用节点的右子树替代当前节点，然后释放当前节点。

如果节点的左右子树都不为空，则找到节点左子树的最大值，以及最大值的父节点（如果节点左子树第一个节点没有右子树，则最大值就是节点左子树的第一个节点，也可能在节点左子树第一个节点的右子树上）；然后用节点左子树的最大值替换当前节点的值；（1）如果节点左子树第一个节点没有右子树，则将节点左子树第一个节点的左子树接到当前节点上，然后释放节点左子树的第一个节点；

（2）如果节点左子树第一个节点有右子树，则把最大值那个节点的左子树接到最大值的父节点右子树上（最大值那个节点右子树一定为空，因为如果不为空，最大值就在右子树上了，但是左子树不一定为空），然后释放节点左子树最大值的那个节点。

## 平衡二叉树

平衡二叉树（Balanced Binary Tree）或平衡二叉查找树又被称为AVL树（有别于AVL算法），且具有以下性质：它是一 棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。这个方案很好的解决了二叉查找树退化成链表的问题，把插入，查找，删除的时间复杂度最好情况和最坏情况都维持在O(logN)。但是频繁旋转会使插入和删除牺牲掉O(logN)左右的时间，不过相对二叉查找树来说，时间上稳定了很多。

Leetcode上有一道题，判断一棵树是否是平衡二叉树。（1）一般的思路就是将这个问题分解递推，转为判断左子树是否平衡，右子树是否平衡，左右子树高度差是否大于1。但是仅仅用这样的思路去写程序，跑起来会很慢，为什么呢？因为每次求子树高度的时候，都要做一次深度优先遍历，这是很没有必要的。怎么去解决这个问题呢？（2）为了解决这个问题，首先要确定，求子树高度是必须的，知道了左右子树的高度才能确定树是否平衡。如果左右子树高度超过1，那就没必要继续求左右子树高度了，直接向上返回，但是怎么返回呢？（3）如果用抛异常的方法，实际测试发现更慢了，因为抛异常很费资源。如果加入一个全局变量is\_balance作为判断条件，作为信息，告诉其他递归函数树已经不平衡了，没必要继续求子树高度了，就是一个非常好的处理办法。（4）综上，可以写出代码来了。

public Boolean is\_balance = true;

public int height(TreeNode tree) {

if (is\_balance == false) {

return 0;

}

if (tree == null) {

return 0;

}

int leftHeight = height(tree.left);

int rightHeight = height(tree.right);

if (leftHeight - rightHeight > 1 || leftHeight - rightHeight < -1) {

is\_balance = false;

}

return leftHeight > rightHeight ? leftHeight + 1 : rightHeight + 1;

}

public boolean isBalance(TreeNode treeNode) {

height(treeNode);

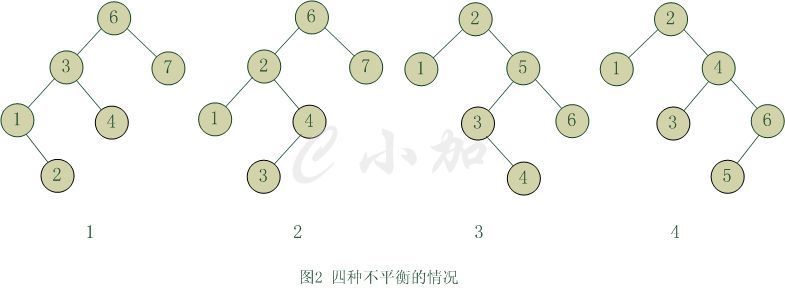
return is\_balance;

}

参考：<https://blog.csdn.net/pengchengliu/article/details/93844275>

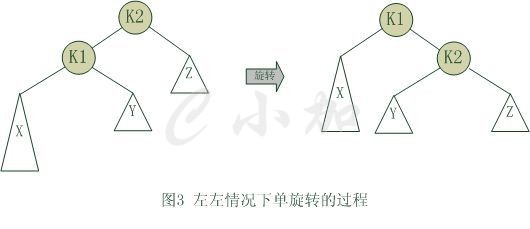
**二叉搜索树的节点是带权的，各节点的权重满足： 左 < 中 <右**

平衡二叉树插入、删除，可能导致树的平衡性缺失，即左右子树高度差超过1，有以下四种情况：

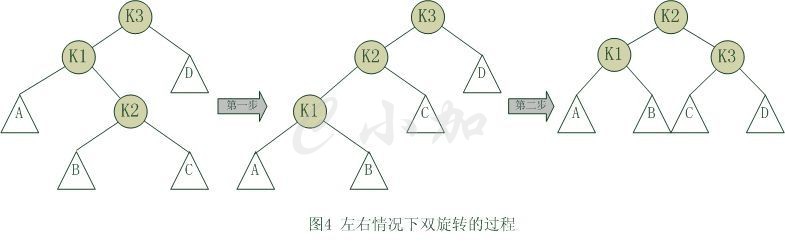


这时需要调整。

对于1和4的情况，调整原则如下图。情况4是下图反过来的。这样可以把高度差减小。



对于2和3的情况，调整原则如下图。即先旋转一次，再旋转一次，才能把顺序调整过来。第一次旋转不改变该子树的平衡性，只是把左子树的高度增加1，右子树的高度减少1，但是方便了下一次的旋转。



# 大根堆（最大堆）

**堆满足完全二叉树的性质，所以堆用一个简单的数组存储即可。**

**大根堆（最大堆）中每个节点的值都大于或等于其左右孩子的值，所以叫大根堆。相反，小根堆（最小堆）中每个节点的值都小于或等于其左右孩子的值。**

这里只讲大根堆，方便熟悉该类数据结构的性质。

大根堆有增、删、查三个需求，先说查这个需求。

用java编写api。

public class MaxRootHeap {

//数组

private int[] arr;

//最大堆已有节点的个数

int num = 0;

//构造函数，设置最大堆的最大容量

public MaxRootHeap(int cap) {

arr = new int[cap];

}

//获取父节点在数组中的位置。如果node是根节点，则node=0，算出来的父节点还是根节点自己，因为java中-1/2=0。

private int parent(int node) {

return (node - 1)/2;

}

//获取左孩子在数组中的位置。

private int left(int node) {

return 2\*node + 1

}

//获取右孩子在数组中的位置。

private int right(int right) {

return 2\*node + 2

}

//获取大根堆的最大值，或者说是根节点

public int max() {

return arr[0];

}

}

再说增这个需求。

**增加新元素，就把新元素放在数组的最后，然后让新元素上浮，到达它应该在的位置。代码如下：**

public void insert(int val) {

arr[num] = val;

rise(num);

num++;

}

//新元素上浮

private void rise(int idx) {

//到达大根堆根节点，结束；或者已经小于等于父节点的值，结束。否则继续。

while (idx > 0 && arr[parent(idx)] < arr[idx]) {

int temp = arr[idx];

arr[idx] = arr[parent(idx)];

arr[parent(idx)] = temp;

idx = parent(idx);

}

}

再说删这个需求。

**大根堆只允许删除堆的最大值，即根节点。**所以删除的步骤是**（1）拿掉根节点；（2）把数组的最后一个元素放到根节点，让数组长度减一；（3）根节点下沉，到达它应该在的位置。**代码如下：

public int deleteMax() {

int max = arr[0];

num--;

arr[0] = arr[num];

sink(0);

return max;

}

//根节点下沉

private void sink(int idx) {

//如果没有左孩子了，下沉就结束

while(left(idx) < num) {

//假设左孩子的值比右孩子大

int bigger\_idx = left(idx);

//如果有右孩子，则要与右孩子比较，找出哪个大

if (right(idx) < num && arr[bigger\_idx] < arr[right(idx)]) {

bigger\_idx = right(idx);

}

//根节点与左右孩子中较大的进行比较，比较结果如果是大于等于，则不用继续下沉了，调整结束。如果比较结果是小于，那就跟左右孩子中较大的进行置换

if (arr[bigger\_idx] < arr[idx]) {

break;

}

int temp = arr[idx];

arr[idx] = arr[bigger\_idx];

arr[bigger\_idx] = temp;

idx = bigger\_idx;

}

}

## 堆排序

大根堆可以用来做堆排序，思路是（1）对给定的数组，将数组中0,1,2,3。。。。。。的数依次插入大根堆中；（2）将大根堆的最大值删除，让堆排序重新调整得到新的大根堆，然后把刚才删除的最大值放到数组的末尾（这个末尾已经不在最大堆中了，存放排序数组刚刚好）；（3）这样递归下去，就可以得到从小到大的有序数组。



堆排序复杂度分析：

（1）不需要额外的空间，所以空间复杂度为O（1）；

（2）如果数组中有值相同的数，由于调整，不能保证其初始顺序，所以是不稳定的。

（3）最好情况，数组是从大到小有序的，这样建立大根堆时免去了调整过程，每次仅需比较大小，时间复杂度O（n）；

但是输出有序数组时，仍然需要调整大根堆，每次调整约需要log2n次比较和交换，一共需要n次，所以时间复杂度是O（nlog2n）。

（4）最坏情况，数组乱序，建立大根堆每次调整约需要log2n次比较和交换，一共需要n次，时间复杂度是O（nlog2n）；

输出有序数组时，调整大根堆，时间复杂度仍然是O（nlog2n）；综合下来，时间复杂度就是O（nlog2n）。

**综上，堆排序由于排序不稳定，比归并排序差一点点。**

但是堆排序用来寻找前k小的数，就是最好的。因为堆排序把前K小的数保存在大根堆中，如果有更小的数出现，就删除大根堆中最大的数并插入更小的数。

**如果用一个数组保存前k小的数，更新时，约需要k次交换，可能需要n次更新，所以时间复杂度是O（n\*k）**

**如果用大根堆来保存前k小的数，更新时，约需要log2k次比较和交换，减少了交换的次数，所以时间复杂度是O（n\*log2k）。这个时间复杂度比排序算法的平均时间复杂度O（nlog2n）更低。**

# 红黑树

Java中的TreeMap和TreeSet就是红黑树。

红黑树首先是一个二叉查找树，在此基础上，红黑树需要满足四个要求：

**（1）根节点必须是黑色；**

**（2）叶子节点必须是黑色，必须是null；**

**（3）红色节点的子节点必须是黑色；**

**（4）从任意节点出发到其下的叶子节点结束，虽然有很多不同的路径，但是这些路径上包含的黑色节点数目相同。**

由于红黑树的（3）（4）这两个要求，假设红黑树有n个节点（不包括叶子节点，叶子节点没用），则红黑树的高度小于等于2log(n+1)。

同时红黑树的高度大于等于log(n+1)，这是满二叉树的性质，因此红黑树最长路径与最短路径的比值小于1:2，接近平衡。

**红黑树的构造不是唯一的**，可以有很多同分异构，即元素相同结构不同，例如插入顺序不同可能造成同分异构。

**红黑树可能所有节点都是黑的**，比如先插入节点，然后把所有的红色节点删掉，就会变成一个全黑的红黑树。但是新插入的节点一定是红的，不需要调整颜色。

**红黑树添加节点**

**第一步**，如果红黑树没有根节点，则将添加的节点作为红黑树根节点，标为黑色。

**第二步**，如果红黑树有根节点，则按照二叉查找树的规则，插入节点，节点的左右孩子必然为空，因为二叉查找树新插入的节点就是这样；然后将节点颜色标为红色，因为标红色只会违背红黑树的要求（3），违背的要求越少越容易调整。

**第三步**，调整节点的颜色。如果刚才插入的节点，其父节点是红色，则需要调整节点颜色，否则不需要调整节点颜色。

（1）如果父节点是祖父节点的左孩子（祖父节点一定存在，因为父节点是红色，祖父节点必然是黑色）

（a）如果祖父节点的右孩子也是红色，即叔叔节点也是红色。那就将父节点和叔叔节点标为黑色，将祖父节点标为红色。这样祖父节点只会违背红黑树的要求（3），其余部分都符合红黑树的四个要求。现在问题转化为递归调整祖父节点的颜色。

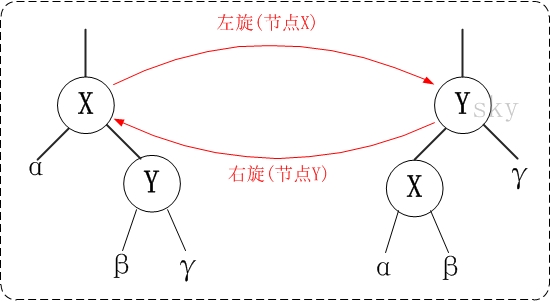
（b）否则叔叔节点是黑色，则要看刚插入的节点，是其父节点的左孩子还是右孩子。

（b1）如果是父节点的左孩子。就将父节点标为黑色，将祖父节点标为红色，然后以祖父节点为支点进行右旋，具体操作就是将父节点提升从而替代祖父节点成为曾祖父节点的孩子，父节点的右孩子作为祖父节点的左孩子，祖父节点作为父节点的右孩子。这样操作完成后，红黑树就调整好了，完全符合红黑树的四个要求。右旋如下图所示。

（b2）否则是父节点的右孩子。没办法像（b1）那样调整，但是可以以父节点为支点进行左旋，具体操作就是将刚插入的节点提升从而替代父节点成为祖父节点的孩子，刚插入的节点的左孩子作为父节点的右孩子，父节点作为刚插入的节点的左孩子。这样操作完成后，就变成了（b1）的情形了，进入（b1）分支进行调整即可。

（2）否则父节点是祖父节点的右孩子。那么可以像（1）分支那样进行成镜像操作，具体说就是判断右孩子变成判断左孩子，右旋变成左旋，左旋变成右旋。

**第四步**，将根节点标为黑色。因为（a）分支的操作可能使根节点颜色变为红色，所以这里将根节点颜色标为黑色。



**红黑树删除节点**

**第一步**，按照二叉查找树的规则，删除节点。二叉查找树删除时，可能有三种情况①删除的节点没有左右孩子；②删除的节点有一个孩子；③删除的节点有两个孩子。对于情况①②，直接删除节点，然后把孩子接到节点的父节点上即可。对于情况③，可以找一个合适的替代节点，补掉删除节点的缺口，同时继承删除节点的颜色，然后把替代节点删掉，由于二叉查找树的性质，替代节点的孩子必然小于等于1，这样就可以把问题简化，纳入情况①②中去处理。

**第二步**，如果删除的节点是红色节点，则不需要调整颜色，否则需要调整颜色。

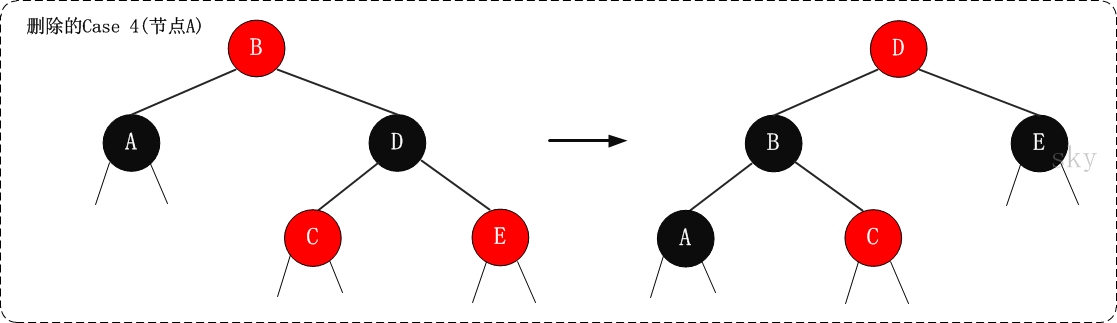
（1）对于情况①②，如果删除节点的孩子是红色，那么把孩子标为黑色，即可重新符合红黑树的四个要求。

（2）否则删除节点的孩子是黑色，那么令删除节点的孩子为当前节点，即讨论的原点，然后分情况来分析。

（a）如果当前节点是根节点，那么不需要调整了，已经符合红黑树的四个要求，结束。

（b）否则当前节点不是根节点，当前节点可能是其父节点的左孩子，也可能是右孩子，分开讨论。

（b1）当前节点是其父节点的左孩子，假设其兄弟是父节点的右孩子。如果兄弟节点的颜色是黑色，兄弟的右孩子是红色，那么就可以通过将兄弟节点提升从而替代父节点成为祖父节点的孩子，



（b2）否则当前节点是其父节点的右孩子，todo

以下链接的插入调整是错的，请注意不要被坑。

<https://www.cnblogs.com/skywang12345/p/3245399.html>