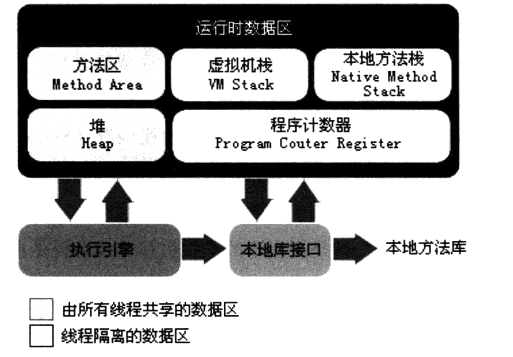
# JVM篇

## 1.1 Java内存和常见异常

运行时数据区域，如下图所示：



**程序计数器：**

* 用来记录指令的执行
* 协助完成线程的切换（时间分片）
* 线程私有的，伴随着线程生死
* 可能的异常未内存溢出异常

**Java虚拟机栈：**

* 线程私有的
* Java方法的执行会在虚拟机栈中形成一个栈帧
* 可能的异常为内存溢出和栈溢出

**堆：**

* 存放对象的实例
* GC区
* 线程共享的区域

本地方法栈：

方法区：

## 1.2 垃圾回收

### 1.2.1 对象死亡的判断

#### 1.2.1.1 引用计数算法

循环引用问题

#### 1.2.1.2 根搜索算法

#### 1.2.1.3 四种引用

#### 1.2.1.4 死亡标记与自救

1、 首先是无根的

2、 进行第一次标记

3、 进入队列

4、 Finalize() ，可以进行自救 oblect = this

### 垃圾收集

#### 标记清除算法

#### 复制算法

常用于新生代

#### 标记整理算法

常用于老年代

#### 分代收集算法

新生代：复制算法

老年代：标记整理

### 1.2.1 常见的垃圾收集器

## 1.3 内存分配

内存特点：

老年代

新生代

### 1.3.1 优先在Eden区分配

* 担保机制下向老年代的转移
* FullGC和MinorGC的区别

### 1.3.2 老年代的特点

* 大对象分配内存时会直接进入老年代
* 长期存活的对象会进入老年代（对象年龄）

### 1.3.3 空间分配担保

长期存活在新生代的大对象，需要老年代提供分配担保进入老年代

# 多线程篇

## 2.1 并发编程的挑战

### 上下文切换的原理

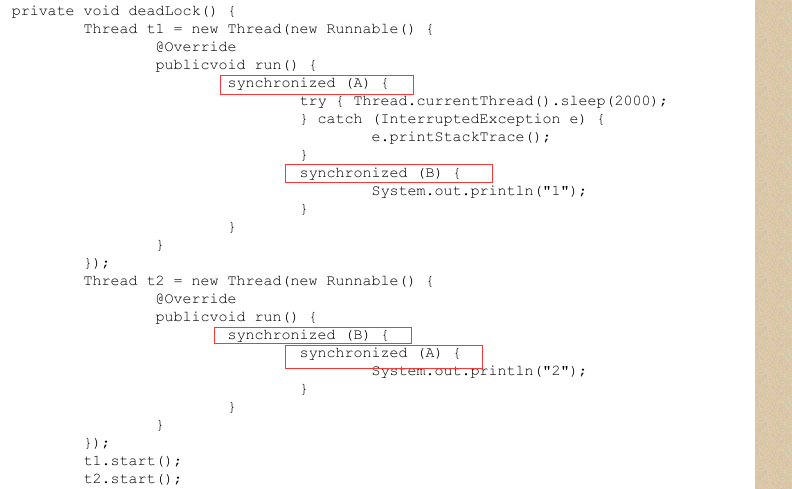
Cpu划分时间片

### 减少上下文切换的方法

* 无锁并发编程
* **使用CAS**
* 使用协程

### 死锁

代码案例：



原因：

**你中有我，我中有你**

**措施：**

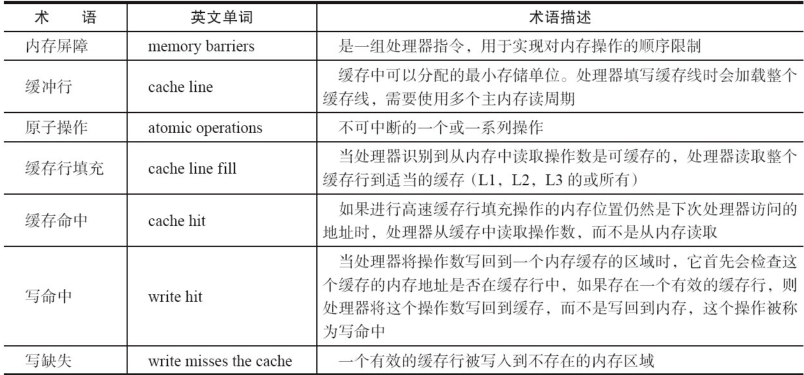
* 使用超时锁，锁的自动释放
* 数据库使用锁，保证获取锁与释放锁在同一个连接里面
* 避免消费一个资源的线程获取多个锁（一个锁只占用一个资源）

## 底层实现原理

### Volatile：

* 1、将缓存的数据写回内存中；2、是其他CPU缓存失效

CPU术语定义：



线程03

线程02

线程01

**缓存失效策**略

缓存

缓存

缓存

总线锁会所著所有资源，消耗较大

内存

### Synchronized：

不同情况下使用syn关键字时具体的锁对象：

1. 同步方法：调用当前方法的对象
2. 同步代码快：括号里面的参数
3. 静态同步方法：Class对象

几种锁类型：

偏向锁（锁的重入）：实现原理，对象头部记录偏向锁的ID.偏向锁的释放。适用于一个资源被同一个线程多次访问的情况

轻量级锁：线程快速切换，代码块执行花费时间短的

重量级锁：追求量，代码快执行时间较长

Java对象的头存储结构：

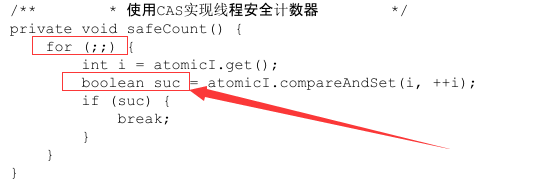
对象的hashcode值、分代年龄值、锁标记

**原子操作的实现原理：**

1. **内存总线锁（当资源数据不在同一个缓存行的时候）**
2. **缓存锁**

**JAVA中实现锁：**

1. **加锁**
2. **通过CAS操作（ABA问题，解决方案是加上一个版本号码；自旋时间过长）**



## 内存模型

通信

线程

线程

同步

缓存

缓存

内存

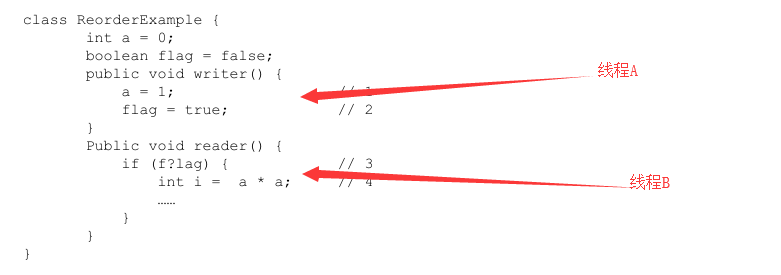
**核心问题：线程间的通信和数据同步**

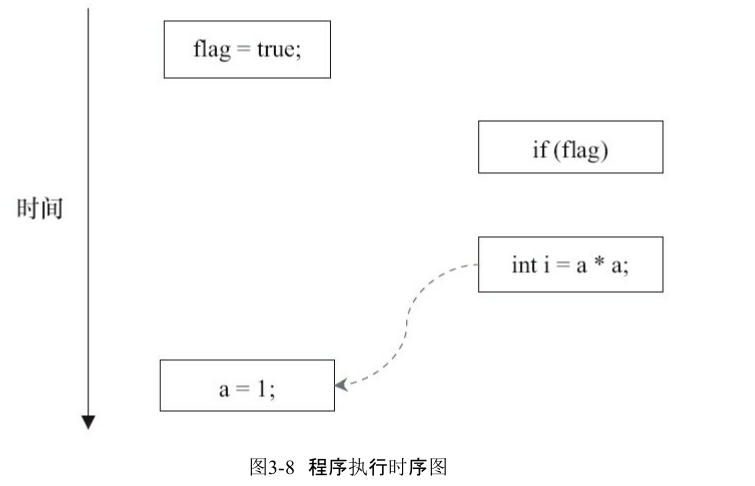
### 2.3.1 指令重排序

重要的是指令重排序不会导致结果的改变

Happen-before原则

**重排序对多线程的影响：**





### 2.3.1 顺序一致性

线程A

线程B

内存中顺序执行

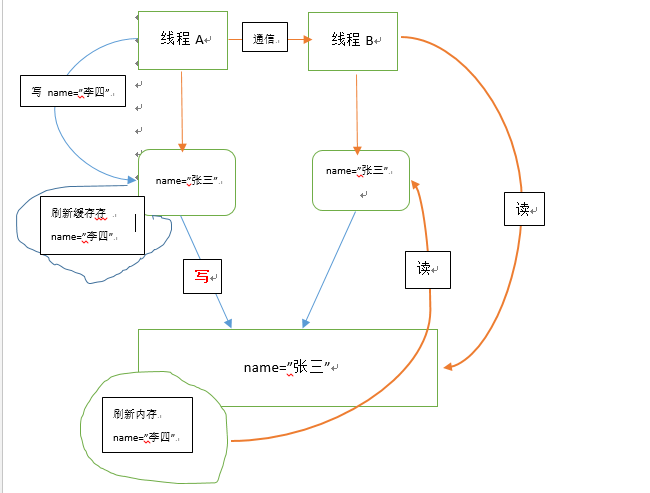
### 2.3.3 Volatile的内存语义

首先遵循happens-before原则。即保证锁的释放和锁的获取之间的内存可见性。在volatite变量中，意味着对volatitle变量的写可以让任意线程（读这个volatitle变量的线程）看到最新的写入结果。

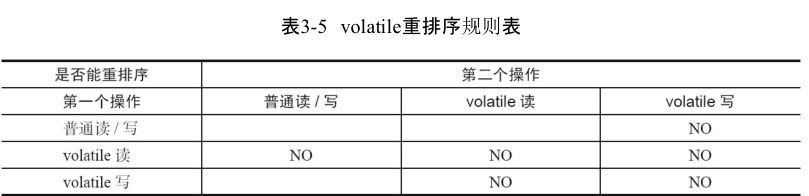
内存图解：

volatitle String name =”张三”

**volatitle 的读写内存模型：**



关于内存屏障的知识点：



### 2.3.4 锁的内存语义

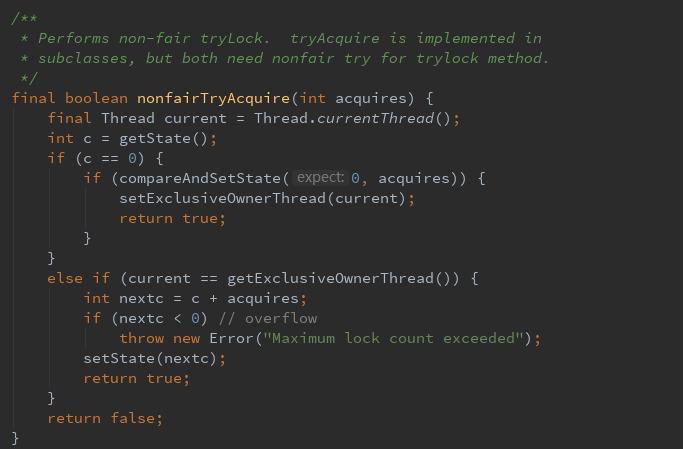
锁的获取与释放，对应着数据的写入与读取

### 2.3.5 ReentrantLock源码分析

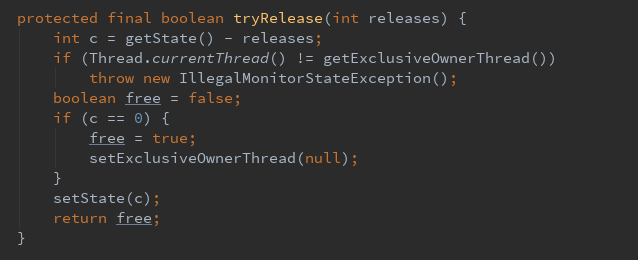
使用模型：

public class ReentrantLockExample {  
 ReentrantLock reentrantLock =new ReentrantLock();  
  
 private String appName = "安卓";  
 public static void main(String[] args) {  
  
 }  
  
 public void write(String name) {  
 reentrantLock.lock();  
 try {  
 this.appName = name;  
 } finally {  
 reentrantLock.unlock();  
 }  
 }  
  
 public String read() {  
 reentrantLock.lock();  
 try {  
 return this.appName;  
 } finally {  
 reentrantLock.unlock();  
 }  
 }  
  
}

**加锁：**



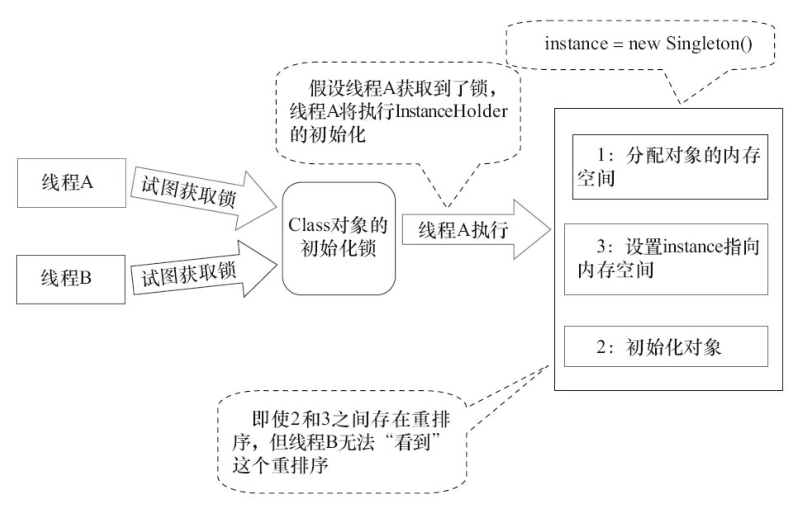
**解锁：**



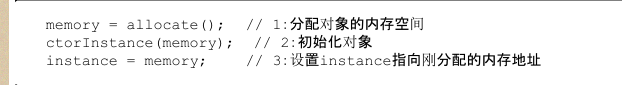
### 2.3.6 双重检查

涉及到的知识点：

1. 类初始化的时候会加锁

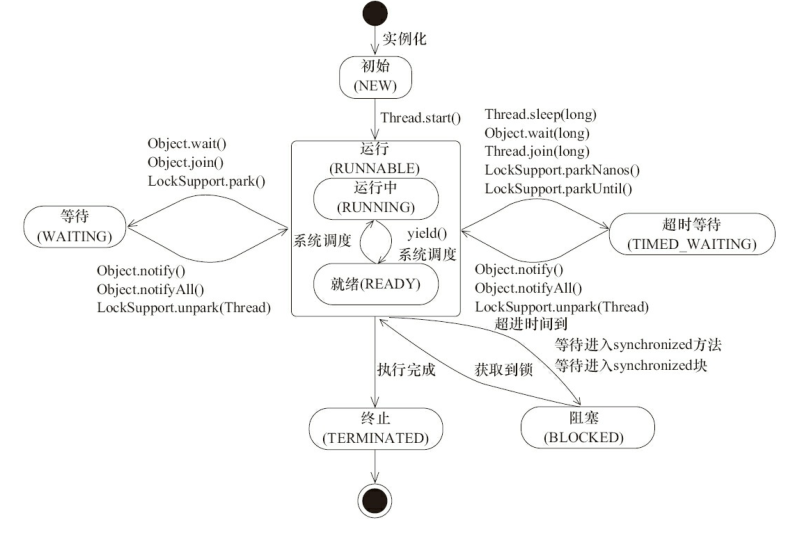


1. Object object = new Object();是经历了三个步骤



## 2.4 并发编程的基础

### 2.4.1 线程的状态转换



1、New 一个线程

2、Start一个线程（是否获得锁/获得锁后是否可以拿到时间分片）

3、2满足的情况下，调用wait()会释放锁，调用sleep()会睡眠但不释放锁

### 线程的相关知识点

* 守护线程
* 线程的终止策略
* 线程的中断

**中断的实战代码：**

public class Interrupted {  
  
 public static void main(String[] args) throws Exception{  
 Thread sleep = new Thread(new SleepThread(),"SleepThread");  
 Thread work = new Thread(new WorkThread(),"WorkThread");  
 sleep.setDaemon(true);  
 work.setDaemon(true);  
 sleep.start();  
 work.start();  
 Thread.*sleep*(2000);  
 sleep.interrupt();  
 work.interrupt();  
 */\*\*  
 \* SleepThread:false  
 \* WorkThread:true  
 \*/* System.*out*.println(sleep.getName()+":"+sleep.isInterrupted());  
 System.*out*.println(work.getName()+":"+work.isInterrupted());  
 }  
 private static class SleepThread implements Runnable {  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 private static class WorkThread implements Runnable {  
 @Override  
 public void run() {  
 while(true){  
 }  
 }  
 }  
}

如何优雅的停止线程：

*/\*\*  
 \* description:优雅的停止线程  
 \*  
 \** ***@author*** *zb 2019/06/25 17:49  
 \*/*public class Interrupted01 {  
  
 public static void main(String[] args) {  
 WorkThread workThread = new WorkThread();  
 Thread thread = new Thread(workThread,"WorkThread");  
 thread.start();  
 try {  
 Thread.*sleep*(10);  
  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 System.*out*.println("异常");  
 }  
 workThread.cancel();  
 System.*out*.println("thread status:"+ thread.getState());  
 // 休眠1s，再看目标线程的状态  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println("线程状态: " + thread.getState());  
 }  
  
 public static class WorkThread implements Runnable {  
 private AtomicInteger i = new AtomicInteger();  
 public volatile boolean flag = true;  
 @Override  
 public void run() {  
 //双重支持&& !Thread.currentThread().isInterrupted()  
 while(flag && !Thread.*currentThread*().isInterrupted()) {  
 i.getAndAdd(1);  
 System.*out*.println("flag:"+flag+";"+"i:"+i.get());  
 }  
 }  
 public void cancel() {  
 System.*out*.println("中止线程！");  
 flag = false;  
 }  
 }  
}

### 线程间通信

线程3

线程2

线程1

进入队*列*

进入队*列*

等待队列

锁对象

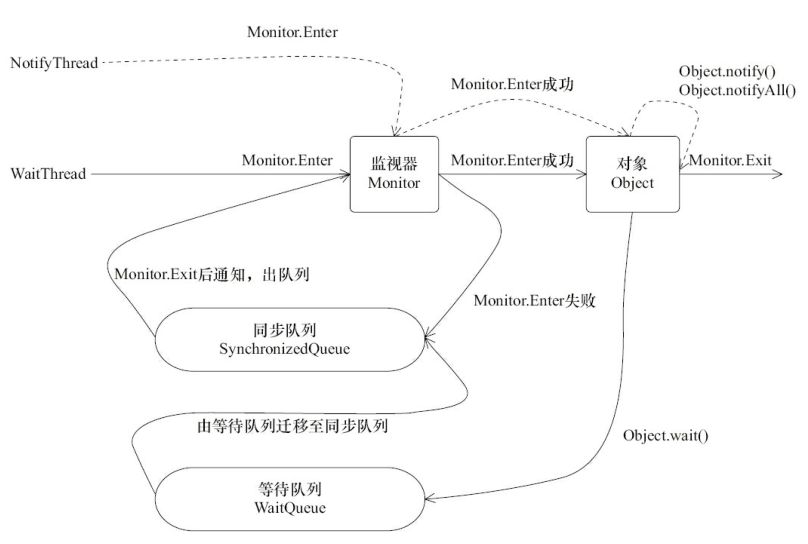
Monitor.Enter成功

Notify

1. package com.zhang.Thread\_Topic.notify;
3. import java.text.SimpleDateFormat;
4. import java.util.Date;
5. import java.util.concurrent.TimeUnit;
7. /\*\*
8. \* description
9. \*
10. \* @author zb 2019/06/30 15:55
11. \*/
12. **public** **class** WaitNotify {
14. **private** **static** Object lock = **new** Object();
15. **private** **static** Boolean flag = **true**;
17. **public** **static** **void** main(String[] args) throws Exception {
18. Thread waitThread = **new** Thread(**new** Wait(), "WaitThread");
19. waitThread.start();
20. TimeUnit.SECONDS.sleep(1);
21. Thread notifyThread = **new** Thread(**new** Notify(), "NotifyThread");
22. notifyThread.start();

25. /\*\*
26. \* falg:true;16:05:15  //获取锁并执行同步代码快，wait()释放锁，进入等待队列
27. \* Thread[NotifyThread,5,main] hold lock. notify @ 16:05:16 //获取锁并执行同步代码快，notifyall唤醒等待队列的代码到同步代码块中，待自己同步代码快执行完成之后释放锁
28. \* Thread[NotifyThread,5,main] hold lock again. sleep @  16: 05: 21
29. \* 验证释放锁后代码是否会执行
30. \* falg:false;16:05:26
31. \*/
32. }
34. **static** **class** Wait implements Runnable {
36. @Override
37. **public** **void** run() {
38. synchronized (lock) {
39. **while** (flag) {
40. //完成消费
41. System.out.println("falg:" + flag + ";" +
42. **new** SimpleDateFormat("HH:mm:ss").format(**new** Date()));
43. **try** {
44. lock.wait();//释放锁
45. /\*\*
46. \*在再次获取锁的时候会接着执行这段代码
47. \*/
48. System.out.println("验证释放锁后代码是否会执行");
49. } **catch** (InterruptedException e) {
50. e.printStackTrace();
51. }
52. }
54. System.out.println("falg:" + flag + ";" +
55. **new** SimpleDateFormat("HH:mm:ss").format(**new** Date()));
56. }
57. }
58. }
60. **static** **class** Notify implements Runnable {
61. @Override
62. **public** **void** run() {
63. // 加锁，拥有lock的Monitor
64. synchronized (lock) {
65. // 获取lock的锁，然后进行通知，通知时不会释放lock的锁，
66. // 直到当前线程释放了lock后，WaitThread才能从wait方法中返回
67. System.out.println(Thread.currentThread() + " hold lock. notify @ " +
68. **new** SimpleDateFormat("HH:mm:ss").format(**new** Date()));
69. lock.notifyAll();
70. flag = **false**;
71. **try** {
72. Thread.sleep(5000);
73. } **catch** (InterruptedException e) {
74. e.printStackTrace();
75. }
76. }
77. // 再次加锁
78. synchronized (lock) {
79. System.out.println(Thread.currentThread() + " hold lock again. sleep @ " + **new** SimpleDateFormat(" HH: mm: ss ").format(**new** Date()));
80. **try** {
81. Thread.sleep(5000);
82. } **catch** (InterruptedException e) {
83. e.printStackTrace();
84. }
85. }
86. }
87. }
89. }

以上的模型：



### 线程使用案例

1. **public** Connection get(**long** mills) {
2. synchronized (pool) {
3. /\*\*
4. \* 立即获取
5. \*/
6. **if** (mills < 0) {
7. **while** (pool.isEmpty()) {
8. **try** {
9. pool.wait();
10. } **catch** (InterruptedException e) {
11. e.printStackTrace();
12. }
13. }
14. **return** pool.getFirst();
15. } **else** {
16. **long** future = System.currentTimeMillis() + mills;
17. **long** remaining = mills;
18. **while** (pool.isEmpty() && remaining > 0) {
19. **try** {
20. /\*\*
21. \* 当前线程等待有控限连接时被唤醒
22. \*/
23. pool.wait(remaining);
24. /\*\*
25. \* 更新时间
26. \*/
27. remaining = future - System.currentTimeMillis();
28. } **catch** (InterruptedException e) {
29. e.printStackTrace();
30. }
31. }
32. Connection result = null;
33. **if** (!pool.isEmpty()) {
34. result = pool.getFirst();
35. }
36. **return** result;
37. }
38. }
39. }
41. **public** **void** release(Connection connection) {
42. **if** (connection != null) {
43. synchronized (pool) {
44. pool.add(connection);
45. /\*\*
46. \* 这里很重要，唤醒等待获取连接的线程
47. \*/
48. pool.notifyAll();
49. }
50. }
51. }

数据库连接池使用案例

知识点：

1. 超时问题
2. Wait/notify案例

## 线程中的锁

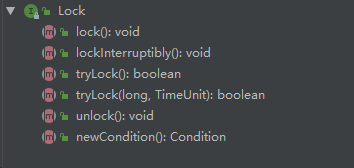
### 2.4.1 Lock

使用案例：

1. **public** **class** ReentrantLockExample {
2. ReentrantLock reentrantLock =**new** ReentrantLock();
4. **private** String appName = "安卓";
5. **public** **static** **void** main(String[] args) {
7. }
9. **public** **void** write(String name) {
10. reentrantLock.lock();
11. **try** {
12. **this**.appName = name;
13. } finally {
14. reentrantLock.unlock();
15. }
16. }
18. **public** String read() {
19. reentrantLock.lock();
20. **try** {
21. **return** **this**.appName;
22. } finally {
23. reentrantLock.unlock();
24. }
25. }
27. }

源码分析：

Lock

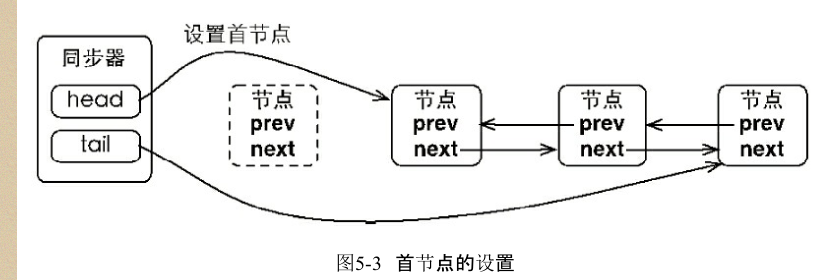


### 2.4.2 读写锁的实现分析

自旋的核心代码：

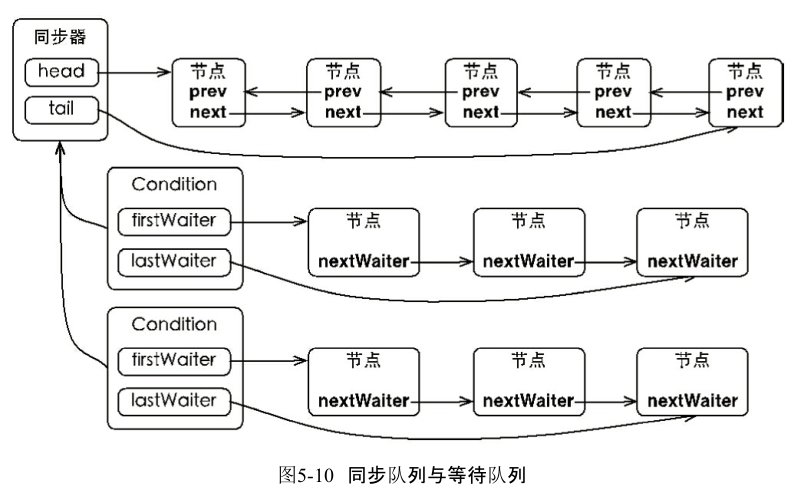
1. final boolean acquireQueued(final Node node, **int** arg) {
2. boolean failed = **true**;
3. **try** {
4. boolean interrupted = **false**;
5. **for** (;;) {
6. final Node p = node.predecessor();
7. **if** (p == head && tryAcquire(arg)) {
8. setHead(node);
9. p.next = null; // help GC
10. failed = **false**;
11. **return** interrupted;
12. }
13. **if** (shouldParkAfterFailedAcquire(p, node) &&
14. parkAndCheckInterrupt())
15. interrupted = **true**;
16. }
17. } finally {
18. **if** (failed)
19. cancelAcquire(node);
20. }
21. }

自旋判断获取锁

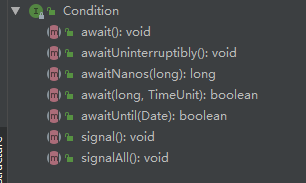


### Condition接口事例

核心原理：



核心接口：



Await()方法：

1. **public** final **void** await() throws InterruptedException {
2. **if** (Thread.interrupted())
3. **throw** **new** InterruptedException();
4. Node node = addConditionWaiter();
5. **long** savedState = fullyRelease(node);
6. **int** interruptMode = 0;
7. **while** (!isOnSyncQueue(node)) {
8. LockSupport.park(**this**);
9. **if** ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
10. **break**;
11. }
12. **if** (acquireQueued(node, savedState) && interruptMode != THROW\_IE)
13. interruptMode = REINTERRUPT;
14. **if** (node.nextWaiter != null) // clean up if cancelled
15. unlinkCancelledWaiters();
16. **if** (interruptMode != 0)
17. reportInterruptAfterWait(interruptMode);
18. }

# 网络通信

## 3.1 TCP/UDP

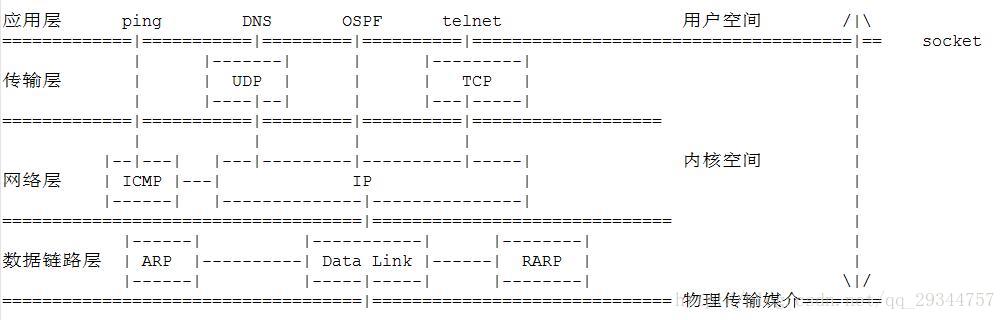
基本区别：

基于连接与无连接  
TCP要求系统资源较多，UDP较少；   
UDP程序结构较简单   
流模式（TCP）与数据报模式(UDP);   
TCP保证数据正确性，UDP可能丢包   
TCP保证数据顺序，UDP不保证

UDP应用场景：

面向数据报方式  
网络数据大多为短消息   
拥有大量Client  
对数据安全性无特殊要求  
网络负担非常重，但对响应速度要求高

### 3.1.2 TCP/IP协议四层模型





**IP:源IP地址，目标IP地址**

**TCP: 目标端口**

**TCP的握手与挥手（面试重点）**

经典好文：<https://juejin.im/post/5c6fbf54f265da2db718216a>

### 3.1.2 流量控制



滑动窗口实现了TCP流控制。首先明确滑动窗口的范畴：TCP是双工的协议，会话的双方都可以同时接收和发送数据。TCP会话的双方都各自维护一个发送窗口和一个接收窗口。各自的接收窗口大小取决于应用、系统、硬件的限制（TCP传输速率不能大于应用的数据处理速率）。各自的发送窗口则要求取决于对端通告的接收窗口，要求相同。

滑动窗口解决的是流量控制的的问题，就是如果接收端和发送端对数据包的处理速度不同，如何让双方达成一致。接收端的缓存传输数据给应用层，但这个过程不一定是即时的，如果发送速度太快，会出现接收端数据overflow，流量控制解决的是这个问题。

**滑动机制**

* 发送窗口只有收到发送窗口内字节的ACK确认，才会移动发送窗口的左边界。
* 接收窗口只有在前面所有的段都确认的情况下才会移动左边界。当在前面还有字节未接收但收到后面字节的情况下，窗口不会移动，并不对后续字节确认。以此确保对端会对这些数据重传。
* 遵循快速重传、累计确认、选择确认等规则。
* 发送方发的window size = 8192;就是接收端最多发送8192字节，这个8192一般就是发送方接收缓存的大小。

<https://www.cnblogs.com/woaiyy/p/3554182.html>

### 3.1.3 Socket编程模型

### 3.1.4 Http协议

1、Http请求

请求头

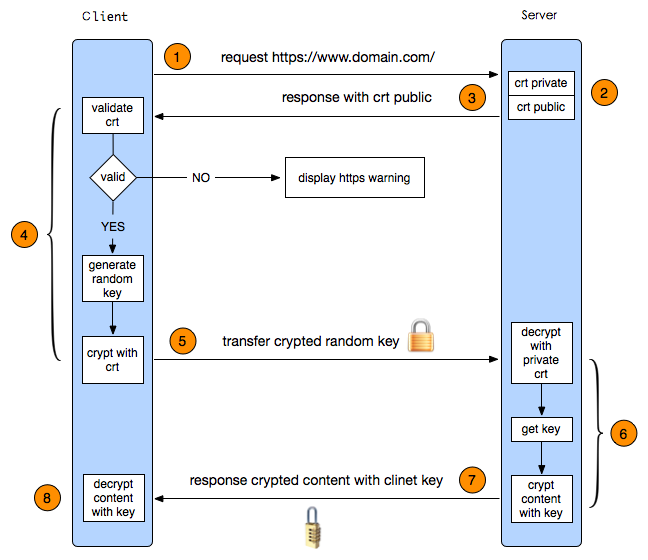
请求内容

请求体

2、Https请求

3、Http的无状态特性

4、Https通信的加密算法



|  |  |
| --- | --- |
|  | -> 客户端向服务端发送请求  -> 服务端返回数字证书  -> 客户端用自己的CA[主流的CA机构证书一般都内置在各个主流浏览器中]公钥去解密证书,  如果证书有问题会提示风险  -> 如果证书没问题客户端会生成一个对称加密的随机秘钥然后再和刚刚解密的服务器端的公  钥对数据进行加密,然后发送给服务器端  -> 服务器端收到以后会用自己的私钥对客户端发来的对称秘钥进行解密  -> 之后双方就拿着这个对称加密秘钥来进行正常的通信 |

5、Cookie与Session

### 3.1.5 Web攻击手段

## 3.2 IO

一些基本概念：

**同步与异步（客户端）**

* 同步： 同步就是发起一个调用后，被调用者未处理完请求之前，调用不返回。
* 异步： 异步就是发起一个调用后，立刻得到被调用者的回应表示已接收到请求，但是被调用者并没有返回结果，此时我们可以处理其他的请求，被调用者通常依靠事件，回调等机制来通知调用者其返回结果。

**阻塞和非阻塞（服务端）**

* 阻塞： 阻塞就是发起一个请求，调用者一直等待请求结果返回，也就是当前线程会被挂起，无法从事其他任务，只有当条件就绪才能继续。
* 非阻塞： 非阻塞就是发起一个请求，调用者不用一直等着结果返回，可以先去干其他事情。

### 3.2.1 BIO

同步阻塞模型

1：1

线程1

Read decode process encode write

线程2

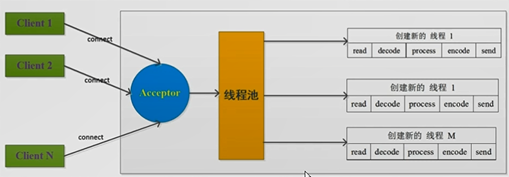
Read decode process encode write

Read decode process encode write

线程n

**缺点：**线程过多/

M:N，引入线程池模型



缺点：线程池会满/同步阻塞模型，客户端会阻塞

1. **package** com.zhang.netty;
3. **import** lombok.extern.slf4j.Slf4j;
5. **import** java.io.\*;
6. **import** java.net.ServerSocket;
7. **import** java.net.Socket;
9. /\*\*
10. \* description
11. \*
12. \* @author zb 2019/06/29 23:00
13. \*/
14. @Slf4j
15. **public** **class** BIOServer {
17. **private** **static** **final** Integer port = 7777;
19. /\*\*
20. \* 单列模式初始化
21. \*/
22. **private** **static** **class** SingletonInstance {
23. **private** **static** ServerSocket INSTANCE = **null**;
25. **public** **static** ServerSocket INIT() {
26. **if** (INSTANCE == **null**) {
27. **try** {
28. INSTANCE = **new** ServerSocket(port);
29. } **catch** (IOException e) {
30. e.printStackTrace();
31. }
32. }
33. **return** INSTANCE;
34. }
35. }

38. **public** **static** **void** start() {
39. ServerSocket serverSocket = SingletonInstance.INIT();
40. log.info("服务端已经启动，端口号为：", port);
41. **try** {
42. /\*\*
43. \* Acceptor :接收新的Socket连接
44. \*/
45. **while** (**true**) {
46. Socket socket = serverSocket.accept();
47. **new** Thread(() -> {
48. log.info("客户端Socket信息：", socket.toString());
49. BufferedReader bReader = **null**;
50. BufferedWriter bWriter = **null**;
51. **try** {
52. bReader = **new** BufferedReader(**new** InputStreamReader(socket.getInputStream()));
53. bWriter = **new** BufferedWriter(**new** PrintWriter(socket.getOutputStream()));
54. /\*\*
55. \* 处理消息
56. \*/
57. **while** (**true**) {
58. //自旋
59. String msg;
60. **if** ((msg = bReader.readLine()) != **null**) {
61. bWriter.write("服务端接受到消息并处理,消息内容：" + msg);
62. } **else** {
63. **break**;
64. }
65. }
66. } **catch** (IOException e) {
67. e.printStackTrace();
68. } **finally** {
69. **if** (bReader != **null**) {
70. **try** {
71. bReader.close();
72. } **catch** (IOException e) {
73. e.printStackTrace();
74. }
75. }
76. **if** (bWriter != **null**) {
77. **try** {
78. bWriter.close();
79. } **catch** (IOException e) {
80. e.printStackTrace();
81. }
82. }
83. }
84. }).start();
85. }
86. } **catch** (IOException e) {
87. e.printStackTrace();
88. } **finally** {
89. **if** (serverSocket != **null**) {
90. **try** {
91. serverSocket.close();
92. } **catch** (IOException e) {
93. e.printStackTrace();
94. }
95. }
96. }
97. }
99. }

### 3.2.2 NIO

回顾基于BIO的Socket的通信模型：

1. {
2. ExecutorService executor = Excutors.newFixedThreadPollExecutor(100);//线程池
4. ServerSocket serverSocket = **new** ServerSocket();
5. serverSocket.bind(8088);
6. **while**(!Thread.currentThread.isInturrupted()){//主线程死循环等待新连接到来
7. Socket socket = serverSocket.accept();
8. executor.submit(**new** ConnectIOnHandler(socket));//为新的连接创建新的线程
9. }
11. **class** ConnectIOnHandler extends Thread{
12. **private** Socket socket;
13. **public** ConnectIOnHandler(Socket socket){
14. **this**.socket = socket;
15. }
16. **public** **void** run(){
17. **while**(!Thread.currentThread.isInturrupted()&&!socket.isClosed()){死循环处理读写事件
18. String someThing = socket.read()....//读取数据
19. **if**(someThing!=null){
20. ......//处理数据
21. socket.write()....//写数据
22. }
24. }
25. }
26. }

上面的变成模型适用于少量的连接需要处理的情况。

缺点在于：

 1. 线程的创建和销毁成本很高，在Linux这样的操作系统中，线程本质上就是一个进程。创建和销毁都是重量级的系统函数。

2. 线程本身占用较大内存，像Java的线程栈，一般至少分配512K～1M的空间，如果系统中的线程数过千，恐怕整个JVM的内存都会被吃掉一半。

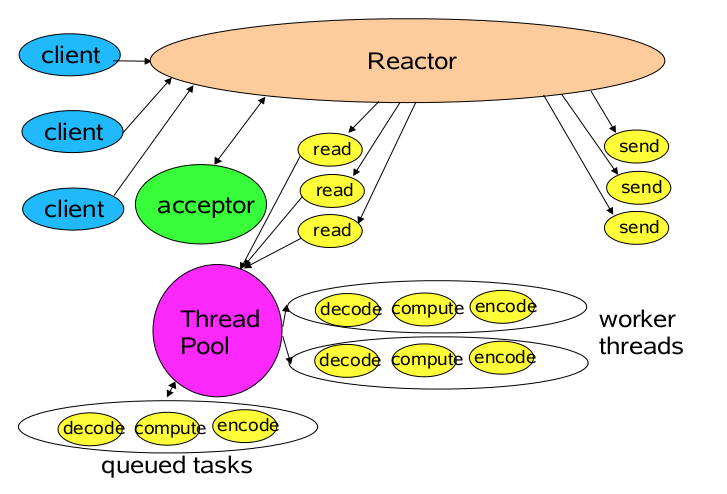
3. 线程的切换成本是很高的。操作系统发生线程切换的时候，需要保留线程的上下文，然后执行系统调用。如果线程数过高，可能执行线程切换的时间甚至会大于线程执行的时间，这时候带来的表现往往是系统load偏高、CPU sy使用率特别高（超过20%以上)，导致系统几乎陷入不可用的状态。

4. 容易造成锯齿状的系统负载。因为系统负载是用活动线程数或CPU核心数，一旦线程数量高但外部网络环境不是很稳定，就很容易造成大量请求的结果同时返回，激活大量阻塞线程从而使系统负载压力过大。

现如今的NIO编程模型：

1. **interface** ChannelHandler{
2. **void** channelReadable(Channel channel);
3. **void** channelWritable(Channel channel);
4. }
5. **class** Channel{
6. Socket socket;
7. Event event;//读，写或者连接
8. }
10. //IO线程主循环:
11. **class** IoThread **extends** Thread{
12. **public** **void** run(){
13. Channel channel;
14. **while**(channel=Selector.select()){//选择就绪的事件和对应的连接
15. **if**(channel.event==accept){
16. registerNewChannelHandler(channel);//如果是新连接，则注册一个新的读写处理器
17. }
18. **if**(channel.event==write){
19. getChannelHandler(channel).channelWritable(channel);//如果可以写，则执行写事件
20. }
21. **if**(channel.event==read){
22. getChannelHandler(channel).channelReadable(channel);//如果可以读，则执行读事件
23. }
24. }
25. }
26. Map<Channel，ChannelHandler> handlerMap;//所有channel的对应事件处理器
27. }

**这个程序很简短，也是最简单的Reactor模式：注册所有感兴趣的事件处理器，单线程轮询选择就绪事件，执行事件处理器。**



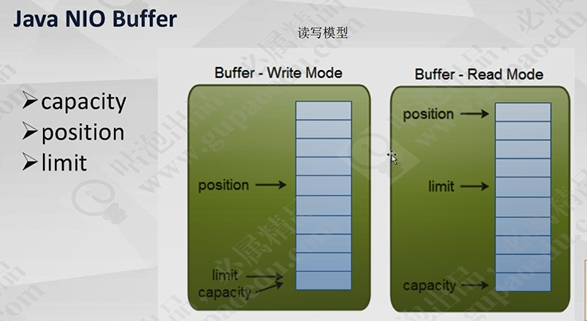
会将BIO中的read/decode/process/encode分别使用不同的线程处理，属于**同步非组赛**

#### 核心组件

1. Channel
2. Buffer
3. Selector

**Buffer:**





Thread

注册事件

Selector

Channel

Channel

Channel

1. // 打开服务器套接字通道
2. ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
3. // 服务器配置为非阻塞
4. serverSocketChannel.configureBlocking(**false**);
5. // 检索与此通道关联的服务器套接字
6. ServerSocket serverSocket = serverSocketChannel.socket();
7. // 进行服务的绑定
8. serverSocket.bind(**new** InetSocketAddress(port));
9. // 通过open()方法找到Selector
10. selector = Selector.open();
12. // 注册到selector，等待连接
13. serverSocketChannel.register(selector, SelectionKey.OP\_ACCEPT);
14. System.out.println("Server Start----8888:");
16. // 向发送缓冲区加入数据
17. send.put("data come from server".getBytes());
18. // 测试此键的通道是否已准备好接受新的套接字连接。
19. **if** (selectionKey.isAcceptable())
20. {
21. // 返回为之创建此键的通道。
22. server = (ServerSocketChannel) selectionKey.channel();
24. // 此方法返回的套接字通道（如果有）将处于阻塞模式。
25. client = server.accept();
26. // 配置为非阻塞
27. client.configureBlocking(**false**);
28. // 注册到selector，等待连接
29. client.register(selector, SelectionKey.OP\_READ
30. | SelectionKey.OP\_WRITE);
31. }
32. **else**
33. **if** (selectionKey.isReadable())
34. {
35. // 返回为之创建此键的通道。
36. client = (SocketChannel) selectionKey.channel();
37. // 将缓冲区清空以备下次读取
38. receive.clear();
39. // 读取服务器发送来的数据到缓冲区中
40. client.read(receive);
42. System.out.println(**new** String(receive.array()));
44. selectionKey.interestOps(SelectionKey.OP\_WRITE);
45. }
46. **else**
47. **if** (selectionKey.isWritable())
48. {
49. // 将缓冲区清空以备下次写入
50. send.flip();
51. // 返回为之创建此键的通道。
52. client = (SocketChannel) selectionKey.channel();
54. // 输出到通道
55. client.write(send);
57. selectionKey.interestOps(SelectionKey.OP\_READ);
58. }

### 3.2.3 AIO

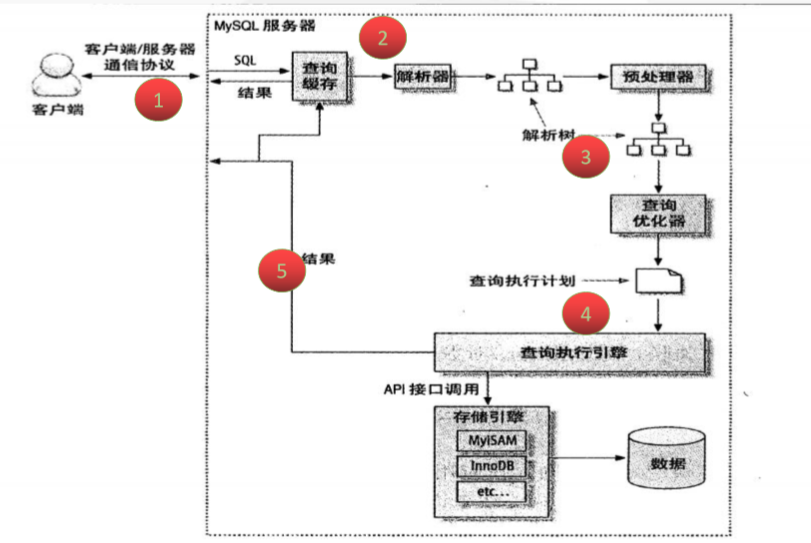
### 3.2.4 Netty

# 4. 数据库

## 4.1 Mysql

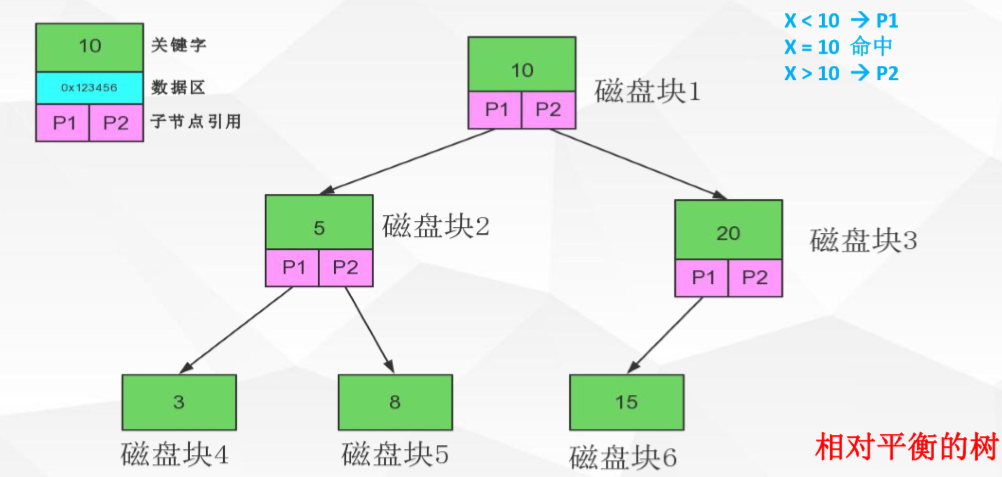






### 4.1.1 Mysql的索引

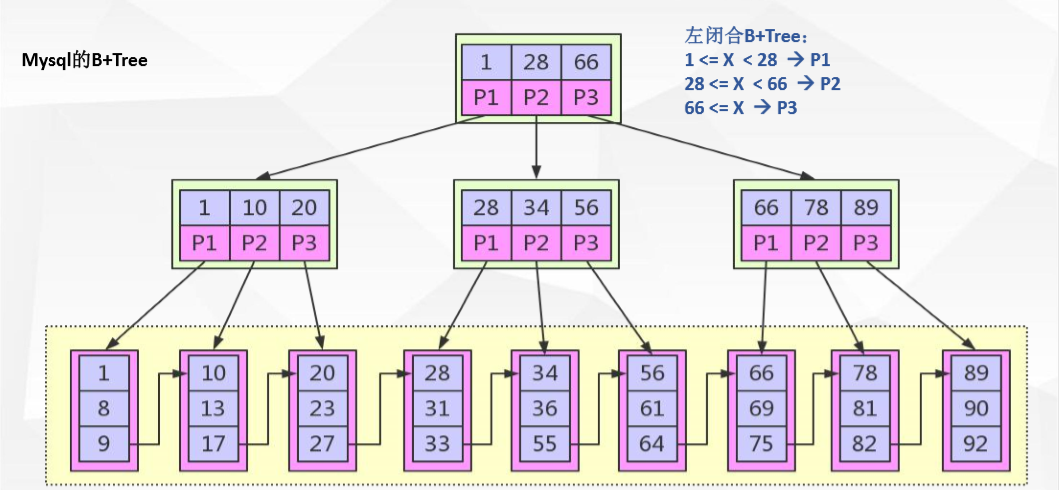
二叉查找树，结构如下：



每个节点的组成：

* 1. 关键字
  2. 数据区域
  3. 子节点引用

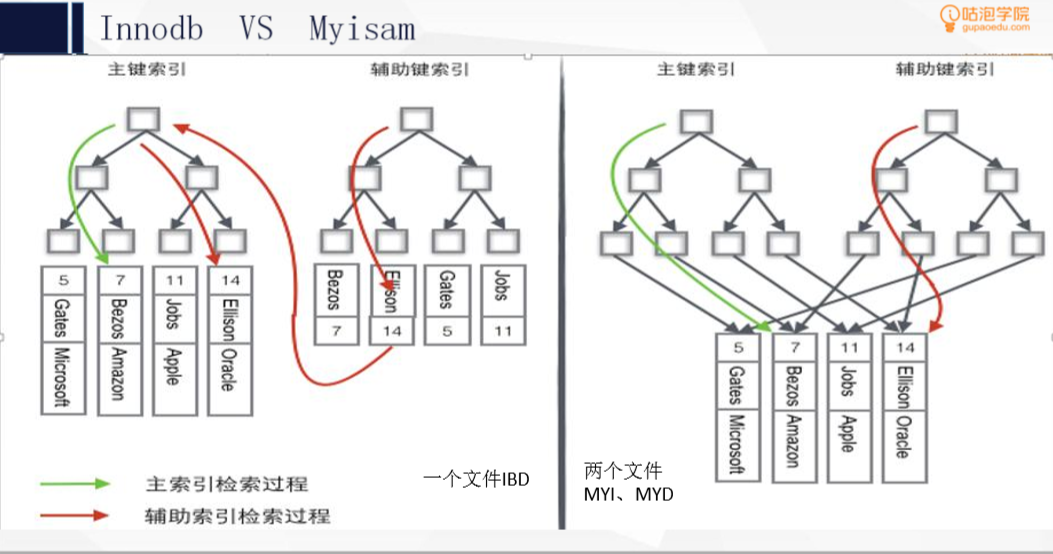
加强的多路平衡树：



特点：

1. 采用左闭合策略
2. 非叶子节点不保存数据，只保存子节点指针和数据区间关键字
3. 真正的数据保存在叶子节点
4. 叶子节点是顺序的，并且有引用关系

**InoDB的二级索引**



**索引的一些注意点：**

1. **作为索引的字段尽量离散**
2. **最左匹配原则**
3. **联合索引**

### 4.1.2 Mysql的锁

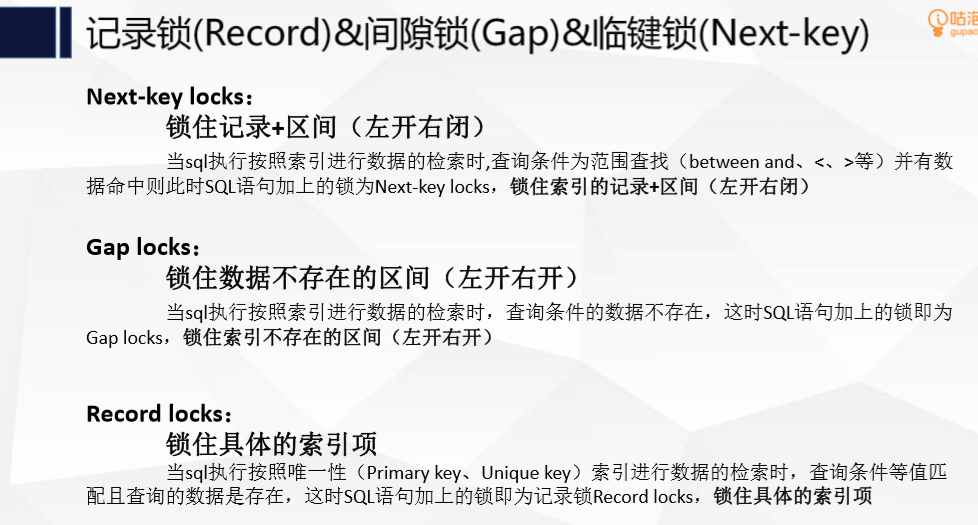
**InnoDB引擎的锁类型**

1. **共享锁**
2. **排他锁**
3. **意向共享锁**
4. **意向排他锁**
5. **自增锁**

**InnoDB行锁的实现：**

通过对索引加锁来实现。

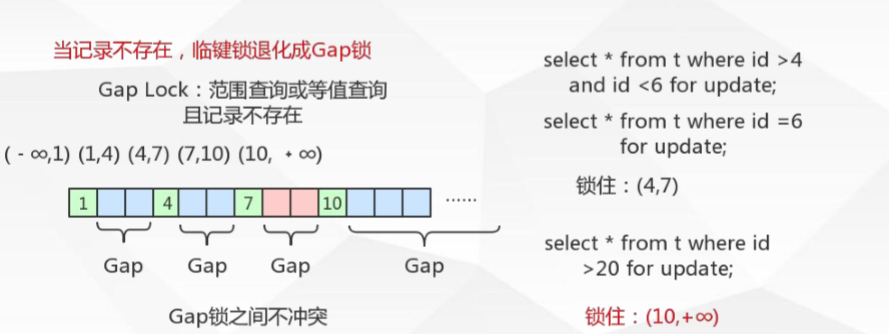
如果不走索引的时候会走表锁



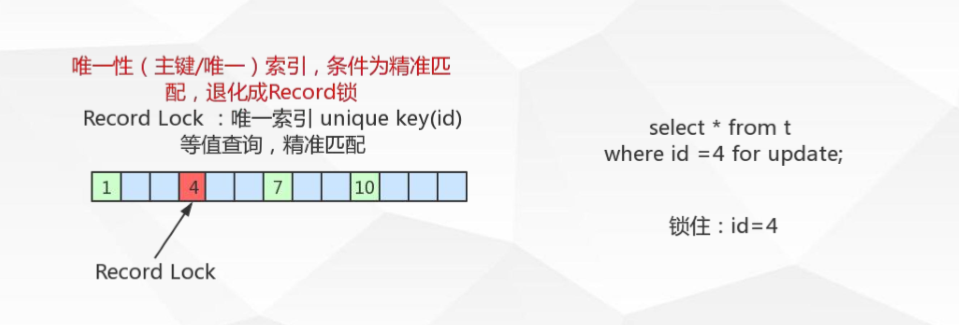
临键锁：



间隙锁：



记录锁：



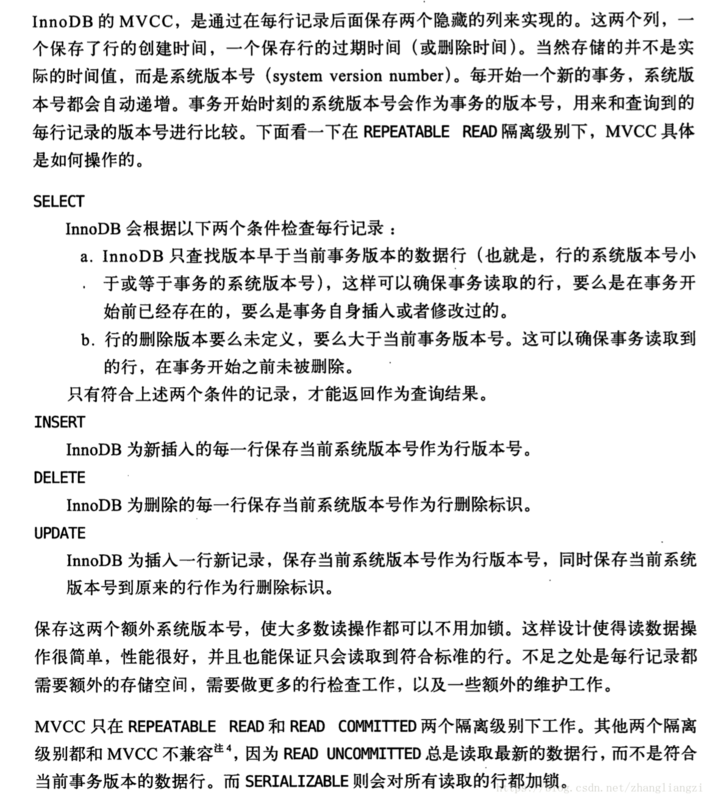
### 4.1.3 MMVC(多版本并发控制)

**普通话解释： 并发访问(读或写)数据库时，对正在事务内处理的数据做 多版本的管理。以达到用来避免写操作的堵塞，从而引发读操 作的并发问题。**

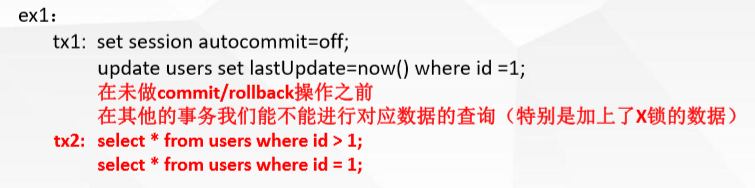


数据行的版本号：全局事物唯一ID

删除版本号：全局事物唯一ID



**假设当前事物版本号码为88**



**事物1**

**Update操作：**

1. 复制原始记录，版本控制号码设置成为 88重新插入
2. 将原始记录的删除版本号设置为88

**事物2**

**快照读：**通过MVCC实现

SQL读取的数据是快照版本，也就是历史版本，普通的SELECT就是快照读

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

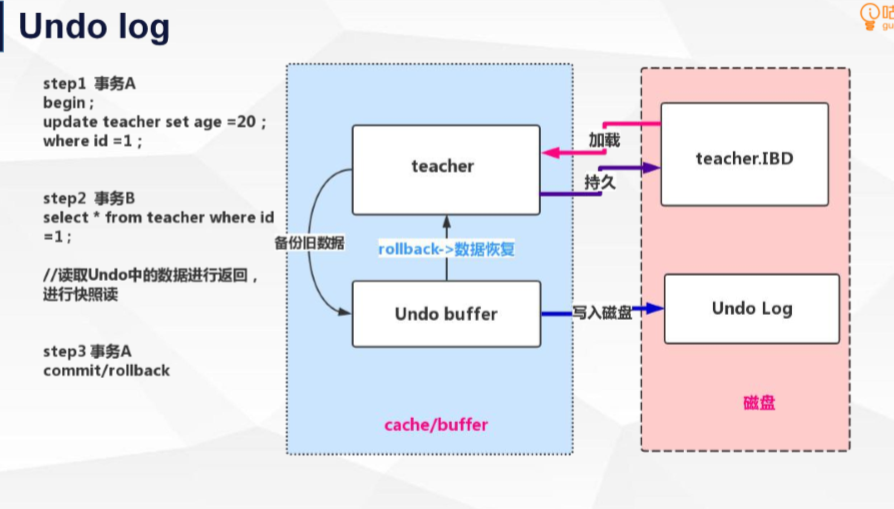
**当前读：**

SQL读取的数据是最新版本。通过锁机制来保证读取的数据无法通过其他事务进行修改

UPDATE、DELETE、INSERT、SELECT … LOCK IN SHARE MODE、SELECT … FOR UPDATE都是当前读

### 4.1.4 日志文件

undo日志用于存放数据修改被修改前的值



用来做数据恢复，主要记录逻辑操作（到每一行的操作）

**Binlog:**

 binlog是一个二进制格式的文件，用于记录用户对数据库**更新的SQL语句信息**，例如更改数据库表和更改内容的SQL语句都会记录到binlog里，但是对库表等内容的**查询不会记录**。

 默认情况下，binlog日志是二进制格式的，不能使用查看文本工具的命令（比如，cat，vi等）查看，而使用mysqlbinlog解析查看。

**Mysql基于BinLog的主从复制机制：**



<https://www.cnblogs.com/dukuan/p/10120820.html>

* 异步复制
* 半异步复制
* GTID复制

### 4.1.5 Mysql优化

性能瓶颈原因：

1. 连接数过多
2. 数据量过大

* 查开开销变大
* 创建索引

1. 硬件资源不足

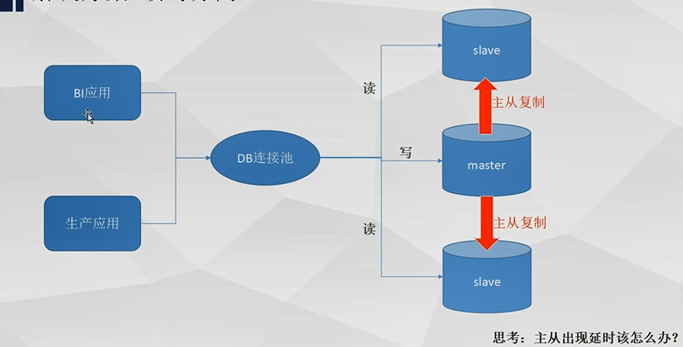
**解释：**

**TPS :每秒事物量**

**QPS:每秒查询量**

**常见Sql优化：**

* 创建索引
* Sql优化
* 使用缓存
* 读写分离
* 分库分表



分库分表：

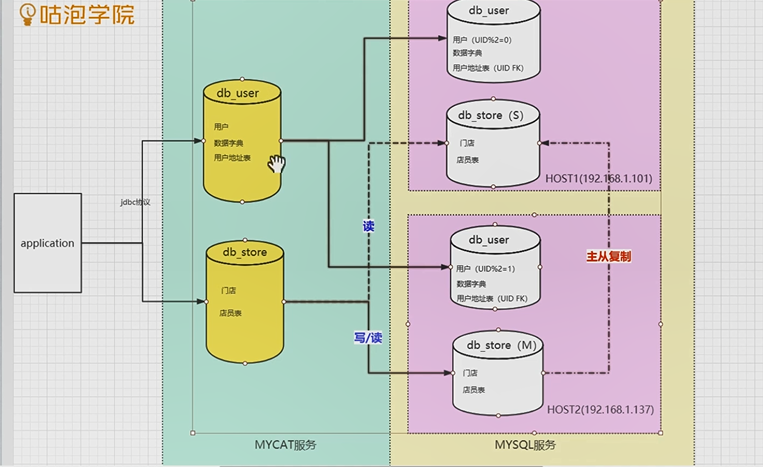
1. 垂直拆分

1. 水平拆分

解决单表数据量过大的问题

#### 4.1.5.1 MyCat

沽泡学院：



**注意：**

用户表做了水平拆分，也就是分库分表。一般而言会有一个分库规则（哈希/取模等方式）。

数据字典表作为全局表需要和用户表进行匹配运算，会在用户表的每个一个库下面进行一个全备份

用户地址表和用户表会存在一堆多的关系

门店表和电源表是做了读写分离

Mycat配置：

<https://www.cnblogs.com/chanshuyi/p/head_first_of_mycat.html>

**主从延时的时候的策略：**

**首先检测延时是否超出范围：**

**是：会直接从主库去读取数据**

**否：从读库读取数据**

**Mycat做数据分片：**

1. **连续分片**

列如：每天的数据做一次分片/每个月的数据做一次分片

优点：数据迁移容易，有助于与时间序列有关的操作

缺点：数据分布不具有随机性，可能存在热点数据集中问题（微博当日热点话题数据）

1. **取模分片**

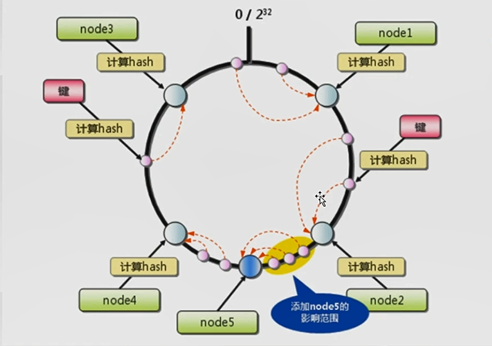
列如：根据用户ID取模或者Hash等等

优点：数据分散较均匀，热点数据不会集中，有效增加并发访问的能力

缺点：数据迁移困难，数据搜索不易命中

1. **综合类分片**

**一致性HasH**



## 4.2 Redis

## 4.3 MongoDB

# 5.分布式

## 序列化

* Java本身提供的系列化
* 基于Xml的序列化

### Java本身自带的序列化

1. ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();  
   try {  
    ObjectOutputStream objectOutputStream = new ObjectOutputStream(byteArrayOutputStream);  
    objectOutputStream.writeObject(person);  
    byteArrayOutputStream.toByteArray();  
     
   } catch (IOException e) {  
    e.printStackTrace();  
   }

<https://www.jianshu.com/p/61f5038df016>

需要关注的点：

* 序列化实现深度克隆
* Transient声明避免序列化
* 实体需要实现writeObject(ObjectOutputStream obj)、readObject( ObjectInputStream objectInputStream )）的方法
* 静态属性的序列化：序列化保存的是对象的状态，静态变量属于类的状态，因此 序列化并不保存静态变量。

### 5.1.2 Xml序列化

核心辅助累 XStream

### 5.1.3 Json实现序列化

### 5.1.4 Hession序列化

### 5.1.5 ProtoBuf序列化

1、BitMap/BitSet的算法原理

Int 4\*8=32

## 5.2 RPC

远程过程调用

### 5.2.1 RMI

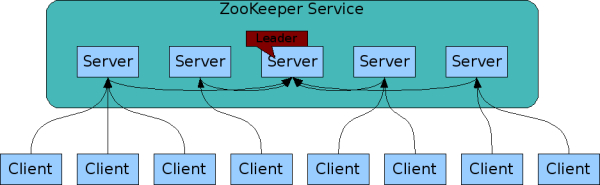
原理：服务端和客户端，服务端注册到一个网络

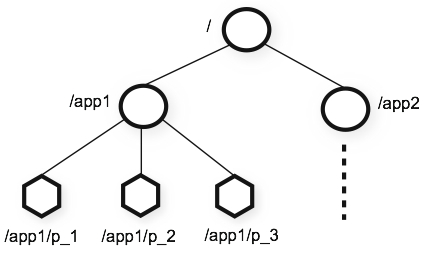
### 5.3.1 Zookeeper

分布式锁

联想在进程中的多线程共享资源

#### 架构及其数据存储：





zookeeeper的集群

处理事务请求（增 删 改）

Marter

事物请求

数据同步

**slave**

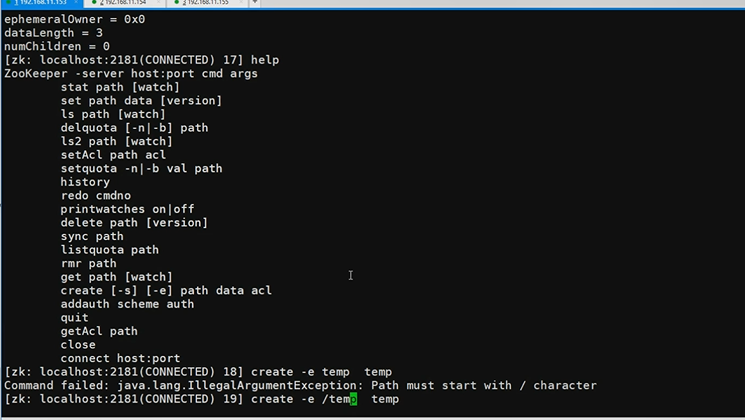
Observer

slave

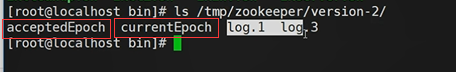
数据同步： 2pc 最终一致性

Zookeeper的客户端可以查看相关的数据存储信息

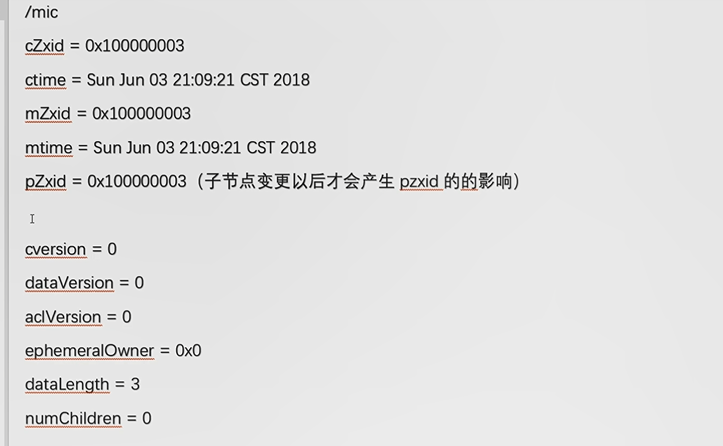
* 有序节点
* 临时节点和持久节点
* 同级节点的唯一性
* 树形的存储结构



DataDir：



Zookeeper中文件得具体内容



#### Observera集群监控

ZK

#### 应**用模型**

文件存储模型

OrderService1

OrderService2

OrderService3

#### ZK设计实现分析

* 1. 单点故障问题
  2. 数据同步问题（改进版的2pc）
  3. 数据一致性问题
  4. 事物机制（过半节点ACK即可）
  5. 选举机制

2PC模型

1

1

参与者2

协调者

参与者1

yes

yes

Commit

Commit

**3PC模型：**

第一阶段

CanCommit:

事物询问

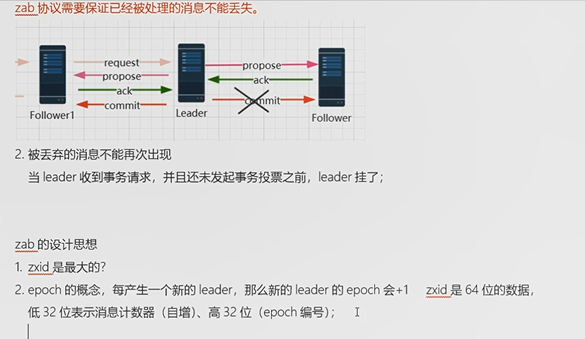
参与者反馈

PreCommit:

#### 崩溃恢复

1. Leader挂掉后的机制

Zxid最大的处理方法



**选举Leader的比较依据：**

Myid/zxid/epoch

**Zk节点的状态：**

**LOKING/LEADING/FLOWING/OBSERVERING**

#### 实现分布式锁

惊群效应

Zookeeper

ClientA

获得锁的节点

/Lock

**监听**

ClientB

**监听**

ClientC

**监听**

ClientD

惊群效应

Zookeeper

ClientA

获得锁的节点

/Lock

**监听**

ClientB

/Lock

**监听**

ClientC

/Lock

**监听**

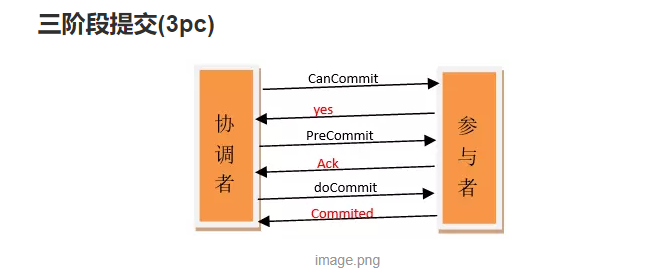
ClientD

/Lock

**可以具体参考Java多线程编程中的实现**

**具体代码：**<https://www.cnblogs.com/wuzhenzhao/p/9996522.html>

#### Zk补充

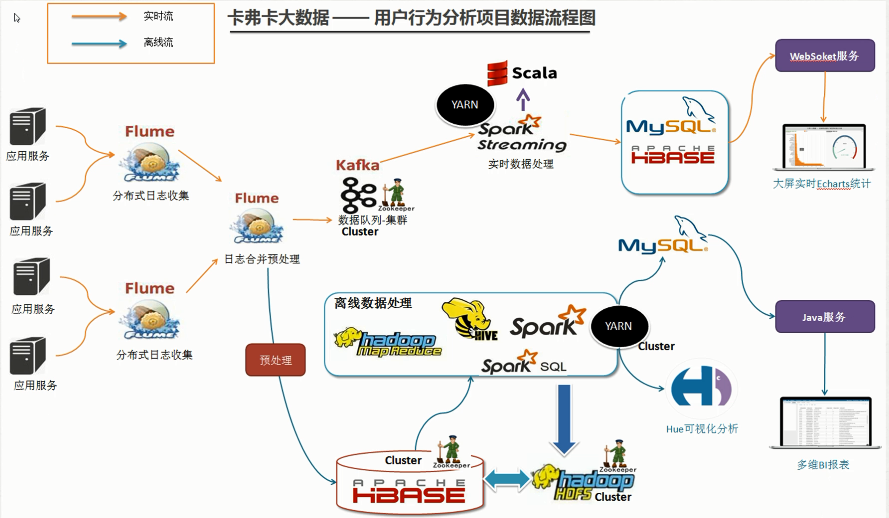


# 6.大数据专题

常用命令：

hadoop dfs -du -h 查看各级文件大小

**常见的架构技术栈：**

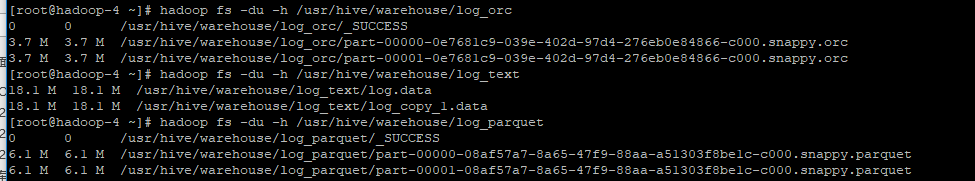


## 6.1 数据存储格式

<https://www.cnblogs.com/eRrsr/p/6098454.html>

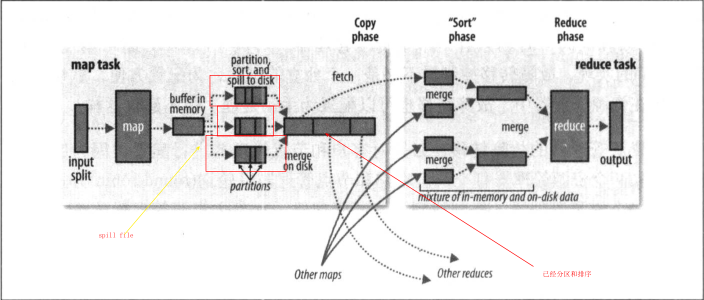
<https://www.cnblogs.com/xiohao/p/6421478.html>

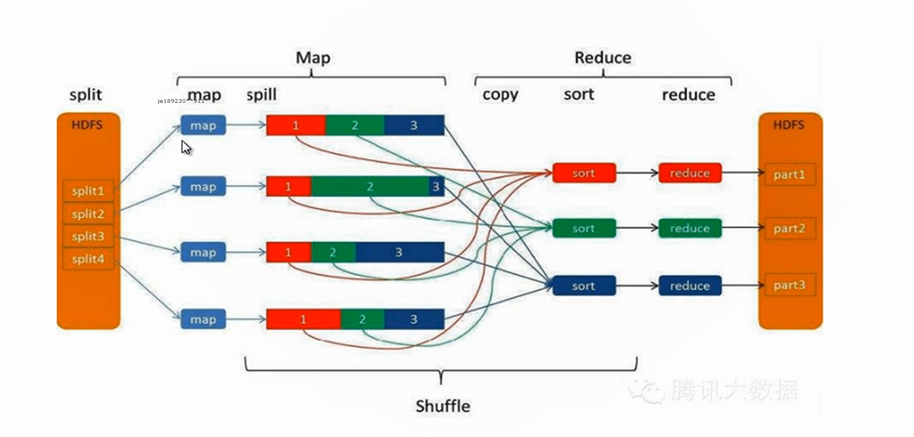
<https://www.jianshu.com/p/694f044d1c34>



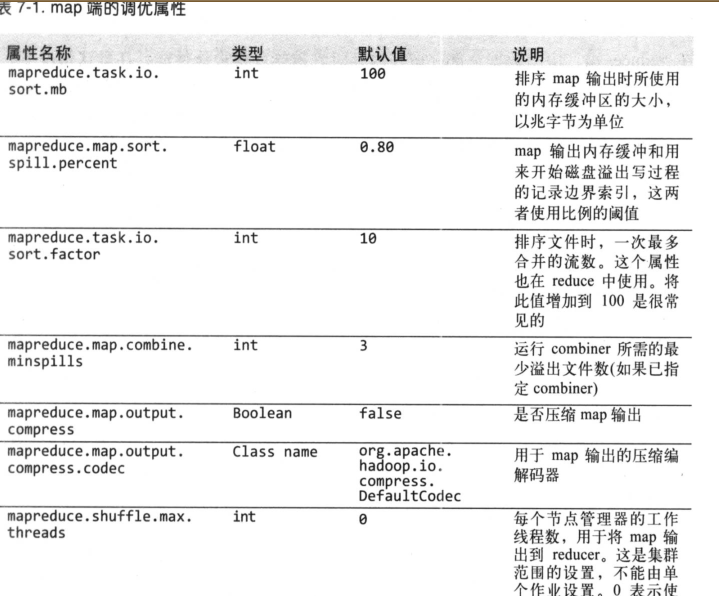


## 6.2 MR原理

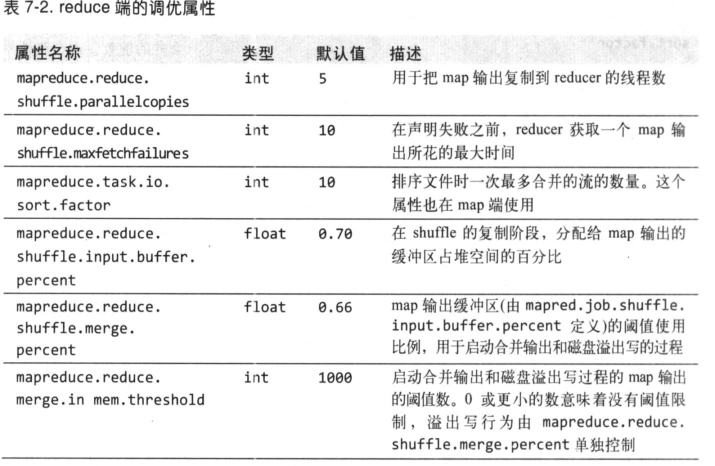




MAP端调优参数



REDUCE端调优



### 考察点

* Map端和Reduce端的调优原理
* Reduce什么时候开始从Map端拉去数据

## 6.3 考点

## 6.4 Spark内核解析

### 6.4.1 核心组件

资源管理器

注册app,请求资源

心跳汇报

分配执行器

运行程序

分配任务

转换

Action

生成job

缓存RDD

执行Task

换份stage

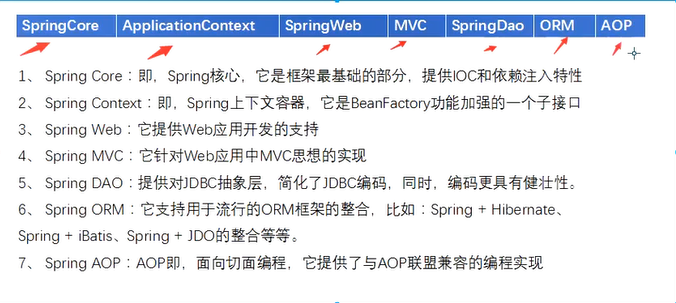
生成taskset

# 7.JavaEE专题

## 7.1 Spring

### 7.1.1 核心框架

* IOC
* AOP

**整体架构：**

### 7.1.2 重点注解

@Scope: 单列(主动加载)、多实例(懒加载)、Session、Request

@ComponentScan：Bean扫描

@Component

@Lazy：懒加载，在使用实例的时候才会初始化bean。否则在容器初始化的时候便会初始化bean实例。