JRE，也就是 Java 运行环境，包含了 JVM 和 Java 类库

JDK 可以看作是 JRE 的一个超集，提供了更多工具，比如编译器、各种诊断工具等

Java 是解释执行”这句话，这个说法不太准确。

JIT（Just-In-Time）编译器，也就是通常所说的动态编译器；热点代码就属于编译执行，而不是解释执行了。

## 1.Exception和Error有什么区别？

运行时异常与一般异常有什么区别？

都是继承了 Throwable（思瑞饿包）

Exception 和 Error 体现了 Java 平台设计者对不同异常情况的分类

I.Exception 是程序正常运行中，可以预料的意外情况，可能并且应该被捕获，进行相应处理。

RuntimeException:NullPoint、ArrayIndexOutOfBoundsException

IOException:

I.Error 是指在正常情况下，不大可能出现的情况；处于非正常的、不可恢复状态 OutOfMemoryError

## 2.NoClassDefFoundError 和 ClassNotFoundException 有什么区别，

听结尾，一个是error 一个是Exception

ClassNotFoundExceptio：加载时ClassLoader ：抛出ClassNotFoundException。

NoClassDefFoundError：编译成功，运行时不成功 new 类是找不到;JVM抛出

## 3.谈谈final、finally、 finalize有什么不同？

final 的方法也是不可以重写的（override

finally 则是 try-catch-finally 来进行类似关闭 JDBC 连接 保证 unlock 锁等动

finalize 是基础类 java.lang.Object 的一个方法，它的设计目的是保证对象在被垃圾收集前完成特定资源的回收。finalize 机制现在已经不推荐使用，并且在 JDK 9 开始被标记为 deprecated。

## 4.强引用、软引用、弱引用、幻象引用有什么区别？

主要体现的是对象不同的可达性（reachable）状态和对垃圾收集的影响。

强引用new

软引用soft reference内存不够时回收 ；OutOfMemoryError 之前

弱引用weak reference第一次垃圾收集清除

虚引用phantom reference幽灵引用;finalize 方法调用之后

## 5.String、StringBuffer、StringBuilder有什么区别？

String 被声明成为 final class，所有属性也都是 final;由于它的不可变性，类似拼接、裁剪字符串等动作，都会产生新的 String 对象。

StringBuffer 线程安全的 stringBuilder

StringBuilder 绝大部分情况下进行字符串拼接的首选。

StringBuffer 和 StringBuilder 底层 （char，JDK 9 以后是 byte）数组 ，二者都继承了 AbstractStringBuilder

构建时初始字符串长度加 16

String操作会自动转换成stringBuilder

Java 6 以后提供了 intern(因特嗯) 方法，目的是提示 JVM 把相应字符串缓存起来，以备重复使用。

## 6.动态代理是基于什么原理？谈谈 Java 反射机制，动态代理是基于什么原理？

反射机制是 Java 语言提供的一种基础功能，赋予程序在运行时自省的能力。

通过反射我们可以直接操作类或者对象，比如获取某个对象的类定义，获取类声明的属性和方法，调用方法或者构造对象，甚至可以运行时修改类定义。

ORmaping框架、限流、入参出参打印

反射：

打开privateAccessibleObject.setAccessible(boolean flag)

动态代理 代理机制 解耦

## 7. int和Integer有什么区别？

Java 的 8 个原始数据类型（ boolean、byte 、short、char、int、float、double、long）之一 一切都是对象，但原始数据类型是例外。

Integer这个值默认缓存是 -128 到 127 之间。 比如最大值、最小值、位数等

自动装箱、自动拆箱机制(语法糖 编译阶段 ):建议避免无意中的装箱、拆箱行为

private final”，所以，它们同样是不可变类型！

对象头

## 8.对比Vector、ArrayList、LinkedList有何区别？

Vector 是 Java 早期提供的线程安全的动态数组，

ArrayList 是应用更加广泛的动态数组实现

LinkedList 顾名思义是 Java 提供的双向链表

Vector 和 ArrayList 作为动态数组 所以非常适合随机访问的场合。除了尾部插入和删除元素，往往性能会相对较差，比如我们在中间位置插入一个元素，需要移动后续所有元素。

而 LinkedList 进行节点插入、删除却要高效得多，但是随机访问性能则要比动态数组慢。

内部排序，至少掌握基础算法如归并排序、交换排序（冒泡、快排）、选择排序、插入排序等。

外部排序，归并(二路归并)按照分块大小拆分成N块。 每两块进行比较，完成后合并

Collection 接口是所有集合的根

List，也就是我们前面介绍最多的有序集合，它提供了方便的访问、插入、删除等操作

Set，Set 是不允许重复元素的 ,Map的马甲

Queue/Deque，则是 Java 提供的标准队列结构的实现，除了集合的基本功能，它还支持类似先入先出（FIFO， First-in-First-Out）或者后入先出（LIFO，Last-In-First-Out）

TreeSet 支持自然顺序访问，但是添加、删除、包含等操作要相对低效（log(n) 时间）。

HashSet 则是利用哈希算法，理想情况下，如果哈希散列正常，可以提供常数时间的添加、删除、包含等操作，但是它不保证有序。

## 9.对比Hashtable、HashMap、TreeMap有什么不同？

Hashtable 是早期 Java 类库提供的一个哈希表实现，本身是同步的，不支持 null 键和值，由于同步导致的性能开销，所以已经很少被推荐使用。

HashMap 不支持同步

TreeMap 则是基于红黑树的一种提供顺序访问的 Map

HashMap(线程不安全)默认大小16；负载因子0.75每次增加2N

(当前数据量>12(默认大小16\*0.75负载因子) )就扩容.0.75空间利用率最高

JDK8之前Hash+链表

JDK8Hash+红黑树 链表大小超过阈值（TREEIFY\_THRESHOLD, 8）

HashMap 内部实现基本点分析。容量（capacity）和负载系数（load factor）。树化 。

哈希碰撞拒绝服务攻击-构建相同的哈希值，变成链表CPU100

## 10.何保证集合是线程安全的? ConcurrentHashMap如何实现高效地线程安全？

一直在变化 默认容量是16 段16个 负载因子0.75

get不加锁

I.JDK7:分成segment (赛格ment) 数组+链表

每个segment段有若干个桶桶里存放K-V形式的链表

Segment 的数量由所谓的 concurrentcyLevel 决定，默认是 16，

每次锁定segment

I.JDK8 Node + CAS + Synchronized设计

node数组，默认为16，可以自动扩展，扩展速度为0.75

每一个节点，挂载一个链表，当链表挂载数据大于8时，链表自动转换成红黑树

put的时候 cas+synchronized

1.空node节点的时候cas

2.有数据node的节点就synchronized

## 11.Java提供了哪些IO方式？ NIO如何实现多路复用？

网络模式

1.阻塞 I/O（blocking IO）BIO

2.非阻塞 I/O（nonblocking IO）NIO

3.I/O 多路复用（ IO multiplexing）非阻塞

3.1.select只支持1024个(每次都遍历)

3.2.poll和select没有区别于链表来存储的.没有个数限制(每次都遍历)：

3.3.epoll（注册—监听事件—处理—再注册） 对比：select和poll每次都全部轮训；epoll只有触发监听才访问

4.信号驱动 I/O（ signal driven IO）

5.异步 I/O（asynchronous IO）

I.java NIO

操作系统底层epoll不断轮训：利用了单线程轮询事件的机制，通过高效地定位就绪的 Channel

Buffer 高效的数据容器，除了布尔类型，所有原始数据类型都有相应的 Buffer 实现。

Channel 文件描述符

Selector(依赖epoll) 多路复用基础，管理channel 单线程对多 Channel 的高效管理。

II.java NIO2 2Asynchronous IO (AIO)异步 IO 操作基于事件和回调机制

利用事件和回调 Future

AsynchronousServerSocketChannel serverSock = AsynchronousServerSocketChannel.open().bind(sockAddr);

serverSock.accept(serverSock, new CompletionHandler<>() {

@Override public void completed(AsynchronousSocketChannel sockChannel, AsynchronousServerSocketChannel serverSock) {

serverSock.accept(serverSock, this); // 另外一个 write（sock，CompletionHandler{}）

sayHelloWorld(sockChannel, Charset.defaultCharset().encode ("Hello World!"));

}

});

## 第12讲 Java有几种文件拷贝方式？哪一种最高效？

NIO transferTo/From(穿丝佛) 利用现代操作系统底层机制，避免不必要拷贝和上下文切换。

I.普通拷贝:用户态空间（User Space）和内核态空间（Kernel Space）先在内核态将数据从磁盘读取到内核缓存，再切换到用户态将数据从内核缓存读取到用户缓存。

会带来一定的额外开销，可能会降低 IO 效率。

磁盘》内核空间》用户空间》内核空间》网络socket

0拷贝：DMA 直接存储器访问:磁盘》内核空间》网络socket

NIO transferTo 的实现方式，在 Linux 和 Unix 上，则会使用到零拷贝技术

优化网络:

使用缓存等机制，合理减少 IO 次数

使用 transferTo 等机制，减少上下文切换和额外 IO 操作。

尽量减少不必要的转换过程，比如编解码；对象序列化和反序列化，比如操作文本文件或者网络通信，如果不是过程中需要使用文本信息，可以考虑不要将二进制信息转换成字符串，直接传输二进制信息。

## 第13讲 谈谈接口和抽象类有什么区别？

I.接口是抽象方法的集合，利用接口可以达到 API 定义和实现分离的目的。

要么是抽象方法，要么是静态方法

任何属性都是 public static final

接口，不能实例化；

implements

I.抽象类是不能实例化的类，用 abstract 关键字修饰 class，其目的主要是代码重用。

除了不能实例化，形式上和一般的 Java 类并没有太大区别，可以有一个或者多个抽象方法，也可以没有抽象方法

。抽象类大多用于抽取相关 Java 类的共用方法实现或者是共同成员变量，然后通过继承的方式达到代码复用的目的。

extends

## 第14讲 | 谈谈你知道的设计模式？

创建型模式，是对对象创建过程的各种问题和解决方案的总结，包括各种工厂模式（Factory、Abstract Factory）、单例模式（Singleton）、构建器模式（Builder）、原型模式（ProtoType）。

结构型模式，包括桥接模式（Bridge）、适配器模式（Adapter）、装饰者模式（Decorator）、代理模式（Proxy）、组合模式（Composite）、外观模式（Facade）、享元模式（Flyweight）等。

行为型模式，是从类或对象之间交互、职责划分等角度总结的模式。比较常见的行为型模式有策略模式（Strategy）、解释器模式（Interpreter）、命令模式（Command）、观察者模式（Observer）、迭代器模式（Iterator）、模板方法模式（Template Method）、访问者模式（Visitor）

BeanFactory 工厂s

AOP 代理模式

listener监听 观察者模式s

## 第15讲 | synchronized和ReentrantLock有什么区别呢？

synchronized，我们根本无法进行公平性的选择，其永远是不公平的

ReentrantLock可重入锁 带超时的获取锁可中断 、可实现公平锁

第16讲 | synchronized底层如何实现？什么是锁的升级、降级？

monitorenter嗯特儿/monitorexit艾克赛特

偏向锁>轻量级锁>重量级锁

对象头 markword

## 第17讲 | 一个线程两次调用start()方法会出现什么情况？

Java 的线程是不允许启动两次的，第二次调用必然会抛出 IllegalThreadStateException不合法线程状态 （A李狗ThreadState）

java.lang.Thread.State

新建（NEW），表示线程被创建出来还没真正启动的状态，可以认为它是个 Java 内部状态。

就绪（RUNNABLE），表示该线程已经在 JVM 中执行，当然由于执行需要计算资源，它可能是正在运行，也可能还在等待系统分配给它 CPU 片段，在就绪队列里面排队。

阻塞（BLOCKED），这个状态和我们前面两讲介绍的同步非常相关，阻塞表示线程在等待 Monitor lock。比如，线程试图通过 synchronized 去获取某个锁，但是其他线程已经独占了，那么当前线程就会处于阻塞状态。

等待（WAITING），被其他线程唤醒，表示正在等待其他线程采取某些操作。 。Thread.join() 也会令线程进入等待状态。

计时等待（TIMED\_WAIT），其进入条件和等待状态类似，但是调用的是存在超时条件的方法，比如 wait 或 join 等方法的指定超时版本，如下面示例：

终止（TERMINATED），不管是意外退出还是正常执行结束，线程已经完成使命，终止运行，也有人把这个状态叫作死亡。

Thread.onSpinWait()，这是 Java 9 中引入的特性

start

join 方法，等待线程结束 ；在线程a中调用线程b的join()方法,此时线程a进入阻塞状态

yield 让出CPU

Object wait/notify/notifyAll

## 第18讲 | 什么情况下Java程序会产生死锁？如何定位、修复？

jstack

避免死锁的方法

1.破坏占用且等待条件-一次性申请所有资源

2.破坏不可抢占条件--主动释放自己占有的资源；synchronized 是做不到;Lock可以解决；占用资源的线程进一步申请资源时，如果申请不到，就主动释放自己占有的资源

3.破坏循环等待条件--对资源进行排序；资源有线性顺序，先申请资源号小的。

## 第19讲 | Java并发包提供了哪些并发工具类？

I.Lock和Condition模拟管程操作

I.Semaphore(赛卖for)信号量(操作系统PV)限流器并发池等

模拟信号量

I.ReadWriteLock读写锁

I.StampedLock邮戳8新增的读写锁升级

三种锁模式

1.写锁writeLock

2.读锁readLock悲观

3.乐观读tryOptimisticRead:validate(验证是否修改)--无锁

I.CountDownLatch和CyclicBarrir（赛科雷 百瑞尔）:如何让多线程步调一致

CountDownLatch主要解决一个线程等待多个线程(不能循环利用)

CyclicBarrier主要解决一组线程之间互相等待(自动重置循环利用)

I.并发容器：性能高

List：CopyOnWriteArrayList内存中两个队列写一个读另外一个：

I.Future(飞偶车)接口和FutureTask实现类获得线程返回值

II.CompletableFuture（康姆铺莱特爆-飞偶车）:异步编程8才提供

串行关系、AND汇聚关系、OR汇聚关系

II.CompetionService(康姆铺莱特身)批量执行异步任务(Future+阻塞队列)

批量的future提交

## 第20讲 | 并发包中的ConcurrentLinkedQueue和LinkedBlockingQueue有什么区别？

“Concurrent\*”容器，才是真正代表并发

java.util.concurrent 包提供的容器 包含:Concurrent\*、CopyOnWrite和 Blocking

Concurrent 类型基于 lock-free，在常见的多线程访问场景，一般可以提供较高吞吐量。

而 LinkedBlockingQueue 内部则是基于锁，并提供了 BlockingQueue 的等待性方法。

## 第21讲 | Java并发类库提供的线程池有哪几种？ 分别有什么特点？

Executors 一克在克特死

newCachedThreadPool()

newFixedThreadPool 无界的工作队列 固定的

newSingleThreadExecutor() 单独的

newScheduledThreadPoo

用什么队列、初始多少最大多少

corePoolSize ：核心线程数量

maximumPoolSize ：线程最大线程数

workQueue ：阻塞队列，存储等待执行的任务 很重要 会对线程池运行产生重大影响

keepAliveTime ：线程没有任务时最多保持多久时间终止

unit ：keepAliveTime的时间单位

threadFactory ：线程工厂，用来创建线程

rejectHandler ：当拒绝处理任务时的策略

## 第22讲 | AtomicInteger底层实现原理是什么？如何在自己的产品代码中应用CAS操作？

CAS CPU指令

Unsafe 不安全的 compareAndSet

## 第23讲 | 请介绍类加载过程，什么是双亲委派模型？

加载loading：通过全名获得二进制字节流；Class特殊是对象，但存放在方法区

连接Linking(

1.验证Verification

2.准备Preparation:给初始值int给0;final的时候会赋值、

3.解析Resolution可以初始化以后再解析（常量池的符号引用替换为直接引用的过程）)

初始化Initialization（是clinit方法执行的过程）<clinit>方法只有在有静态代码块才有,且只执行一次

使用using

卸载Unloading

双亲委派（防止重复加载；防止核心类库被窜改） 全新类先由启动类加载器加载，然后是拓展类加载器，然后是应用程序类加载器，最后是自定义加载器

启动类加载器BootstrapClassLoad(C++实现虚拟机一部分)加载jre/lib/rt.jar

拓展类加载器ExtensionClassloader:加载jre/lib/ext类

应用程序类加载器:加载环境里加载的java类

## 第24讲 | 有哪些方法可以在运行时动态生成一个Java类？

Compiler康拍乐 编译者

自己写个文件，调用命令生成 class 再类加载

## 第25讲 | 谈谈JVM内存区域的划分，哪些区域可能发生OutOfMemoryError?

程序计数器 只有这里不会内存溢出

Java 虚拟机栈 StackOverFlowError栈溢出

堆（Heap），

方法区 java.lang.OutOfMemoryError: Metaspace

本地方法栈 StackOverFlowError栈溢出

虚拟机栈和本地方法栈：StackOverFlowError 当应用程序递归太深而发生堆栈溢出时，抛出该错误。

堆内存不足

## 第26讲 | 如何监控和诊断JVM堆内和堆外内存使用？

JConsole、VisualVM

jstat用于收集HotSpot虚拟机各方面的运行数据 包括eden两个survivor、老年代、永久代、已用时间、GC时间合计

jmap 生成java堆存储快照

jstack 显示虚拟机线程快照

8Eden 1Survivor form 1Survivor to

## 第27讲 | Java常见的垃圾收集器有哪些？

1.标记清除算法（mark sweep） 缺点1：标记和清除效率都不高；缺点2:收集后没有大块的空闲空间给大对象

2.复制算法(新生代用jvm8Eden：1From survivor：1To Survivor):把内存分为1V1，只用一半，收集时，把还存活的复制到另一半，把之前的整块清理

3.标记清除整理算法（老年代用）：比算法1多了一个整理；把存活的内存复制到一起，清理其余的内存

新生代：存活较小，用复制算法

老年代：存活较多，用标记清除算法；或者标记清除整理算法

server模式

jdk6之前 Parallel Scavenge + Serial Old

jdk6、7、8 Parallel Scavenge + Parallel Old

jdk9 G1(Garbage First) 整理上看基于标记整理，局部Region之间基于复制算法实现；没有空间碎片

II.新生代young收集器

1.Serial(C瑞欧)连续的（单个CPU效率最高，client模式下）：复制算法；STW(StopTheWorld)后单线程收集

2.ParNew新式的（多线程版本的serial）:复制算法；STW后多线程收集；在单CPU情况下效率不如Serial

3.Parallel scavenge（拍瑞浪 死噶壁纸）平行的(比parnew多自适应调节策略制定最大停顿时间) 吞吐量Throughput=用户代码时间/(用户代码时间+垃圾收集时间)

II.老年代old收集器

1.Serial Old(MSC)全搭配(单线程标记清除整理算法)

2.Parallel Old（多线程标记清除整理算法）JDK6才开始提供

3.CMS（多线程标记清除无整理）并发收集低停顿

II.G1（面向服务端的收集器）新老都用（低停顿）jdk7时开始提供：

并行与并发:G1能充分利用多CPU

分代收集-管理整个堆

空间整合:整理上看基于标记整理，局部Region（睿真）之间基于复制算法实现；没有空间碎片

可预测的停顿:对于CMS的优势可以指定一个长度X毫秒内的时间片段

每个region也分新生代老年代

## 第28讲 | 谈谈你的GC调优思路?

内存占用（footprint）、

延时（latency）

吞吐量（throughput）

## 第29讲 | Java内存模型中的happen-before是什么？

Happen-befor 提供跨线程的内存可见性保证

先行发生原则

具体的规则：

(1)程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作。

(2)监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁。

(3)volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。

(4)传递性：如果A happens-before B，且B happens-before C，那么A happens-before C。

(5)start()规则：如果线程A执行操作ThreadB.start()（启动线程B），那么A线程的ThreadB.start()操作happens-before于线程B中的任意操作。

(6)Join()规则：如果线程A执行操作ThreadB.join()并成功返回，那么线程B中的任意操作happens-before于线程A从ThreadB.join()操作成功返回。

(7)程序中断规则：对线程interrupted()方法的调用先行于被中断线程的代码检测到中断时间的发生。

(8)对象finalize规则：一个对象的初始化完成（构造函数执行结束）先行于发生它的finalize()方法的开始

## 第30讲 | Java程序运行在Docker等容器环境有哪些新问题？

虚拟化

1.JVM 堆和元数据区、直接内存等参数 Java 就有可能试图使用超过容器限制的内存，最终被容器 OOM kill，或者自身发生 OOM

2.错误判断了可获取的 CPU 资源 。Docker 限制了 CPU 的核数，JVM 就可能设置不合适的 GC 并行线程数等。

VM os

docker 模拟环境

相同的问题

## 第31讲 | 你了解Java应用开发中的注入攻击吗

SQL 注入攻击

Runtime.exec

XML 注入攻击。Java 核心类库提供了全面的 XML 处理、转换等各种 API

## 第32讲 | 如何写出安全的Java代码？

1.使用框架

2.代码质量

3.权限管理

## 第33讲 | 后台服务出现明显“变慢”，谈谈你的诊断思路？

1.看接口 cat 等流检查 分析具体那一步慢

2.full gc STW

3.CPU 网络等占用

## 第34讲 | 有人说“Lambda能让Java程序慢30倍”，你怎么看？

7新增了指令 8增加了语法

## 第35讲 | JVM优化Java代码时都做了什么？

1.编译器优化 字符串相加 2+1 =3

不会执行的代码优化掉

2.运行时优化

即时编译器（JIT）

公共子表达式消除\数组边界检查消除、方法内联inline、锁消除、锁膨胀

## 第36讲 | 谈谈MySQL支持的事务隔离级别，以及悲观锁和乐观锁的原理和应用场景？

悲观锁一般就是利用类似 SELECT … FOR UPDATE

MVCC，其本质就可以看作是种乐观锁机制

## 第37讲 | 谈谈Spring Bean的生命周期和作用域？

I.创建流程

实例化 Bean 对象。

设置 Bean 属性。

Aware 饿外耳

调用 BeanPostProcessor 的前置初始化方法 postProcessBeforeInitialization。

如果实现了 InitializingBean 接口，则会调用 afterPropertiesSet 方法。

调用 Bean 自身定义的 init 方法。

调用 BeanPostProcessor 的后置初始化方法 postProcessAfterInitialization。

调用 BeanPostProcessor 的后置初始化方法 postProcessAfterInitialization。

I.销毁流程

Spring Bean 的销毁过程会依次调用 DisposableBean 的 destroy 方法和 Bean 自身定制的 destroy 方法。

I.Spring Bean 有五个作用域

Singleton，这是 Spring 的默认作用域，也就是为每个 IOC 容器创建唯一的一个 Bean 实例。

Prototype，针对每个 getBean 请求，容器都会单独创建一个 Bean 实例。

如果是 Web 容器，则支持另外三种作用域：

Request，为每个 HTTP 请求创建单独的 Bean 实例。

Session，很显然 Bean 实例的作用域是 Session 范围。

GlobalSession，用于 Portlet 容器，因为每个 Portlet 有单独的 Session，GlobalSession 提供一个全局性的 HTTP Session。

## 第38讲 | 对比Java标准NIO类库，你知道Netty是如何实现更高性能的吗？

更加优雅的 Reactor 模式实现、灵活的线程模型、利用 EventLoop 等创新性的机制，可以非常高效地管理成百上千的 Channel。

充分利用了 Java 的 Zero-Copy 机制

使用更多本地代码。例如，直接利用 JNI 调用 Open SSL 等方式，获得比 Java 内建 SSL 引擎更好的性能。

在通信协议、序列化等其他角度的优化。

Netty 并没有 Java 核心类库那些强烈的通用性、跨平台等各种负担，针对性能等特定目标以及 Linux 等特定环境，采取了一些极致的优化手段。

## 第39讲 | 谈谈常用的分布式ID的设计方案？Snowflake是否受冬令时切换影响？

基本的要求包括：

全局唯一，区别于单点系统的唯一，全局是要求分布式系统内唯一。

有序性，通常都需要保证生成的 ID 是有序递增的。例如，在数据库存储等场景中，有序 ID 便于确定数据位置，往往更加高效。

MongoDB 的ObjectId提 Time+Machine+PID+INC自增计数器

基于UUID

基于数据库主键自增

Reids INCR

## 40讲 | CAP原则Consistency（一致性）。 Availability（可用性）。Partition tolerance（分区容错性）

分布式系统的CAP理论：理论首先把分布式系统中的三个特性进行了如下归纳：  
　　● 一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。（等同于所有节点访问同一份最新的数据副本）

● 可用性（A）：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（对数据更新具备高可用性）

● 分区容错性（P）：以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。

与可用的决择

CAP理论就是说在分布式存储系统中，最多只能实现上面的两点。**而由于当前的网络硬件肯定会出现延迟丢包等问题，所以分区容忍性是我们必须需要实现的。**所以我们只能在一致性和可用性之间进行权衡，没有NoSQL系统能同时保证这三点。

对于web2.0网站来说，关系数据库的很多主要特性却往往无用武之地

1. 数据库事务一致性需求   
   　　很多web实时系统并不要求严格的数据库事务，对读一致性的要求很低，有些场合对写一致性要求并不高。允许实现最终一致性。
2. 数据库的写实时性和读实时性需求  
   　　对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出来这条数据的，但是对于很多web应用来说，并不要求这么高的实时性，比方说发一条消息之 后，过几秒乃至十几秒之后，我的订阅者才看到这条动态是完全可以接受的。
3. 对复杂的SQL查询，特别是多表关联查询的需求   
   　　任何大数据量的web系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的报表查询，特别是SNS类型的网站，从需求以及产品设计角 度，就避免了这种情况的产生。往往更多的只是单表的主键查询，以及单表的简单条件分页查询，SQL的功能被极大的弱化了。

与NoSQL的关系

传统的关系型数据库在功能支持上通常很宽泛，从简单的键值查询，到复杂的多表联合查询再到事务机制的支持。而与之不同的是，**NoSQL系统通常注重性能和扩展性，而非事务机制**（事务就是强一致性的体现）[2]  。  
　　传统的SQL数据库的事务通常都是支持ACID的强事务机制。A代表原子性，即在事务中执行多个操作是原子性的，要么事务中的操作全部执行，要么一个都不执行;C代表一致性，即保证进行事务的过程中整个数据加的状态是一致的，不会出现数据花掉的情况;I代表隔离性，即两个事务不会相互影响，覆盖彼此数据等;D表示持久化，即事务一旦完成，那么数据应该是被写到安全的，持久化存储的设备上（比如磁盘）。  
　　NoSQL系统仅提供对行级别的原子性保证，也就是说同时对同一个Key下的数据进行的两个操作，在实际执行的时候是会串行的执行，保证了每一个Key-Value对不会被破坏。

与BASE的关系

BASE就是为了解决关系数据库强一致性引起的问题而引起的可用性降低而提出的解决方案。

BASE是下面三个术语的缩写：

* 基本可用（Basically Available）
* 软状态（Soft state）
* 最终一致（Eventually consistent）

1. 目前最快的KV数据库,10W次/S, 满足了高可用性。
2. Redis的k-v上的v可以是普通的值（基本操作：get/set/del） v可以是数值（除了基本操作之外还可以支持数值的计算） v可以是数据结构比如基于链表存储的双向循环list（除了基本操作之外还可以支持数值的计算，可以实现list的二头pop,push）。如果v是list，可以使用redis实现一个消息队列。如果v是set,可以基于redis实现一个tag系统。与mongodb不同的地方是后者的v可以支持文档，比如按照json的结构存储。redis也可以对存入的Key-Value设置expire时间。
3. Redis的v的最大远远超过memcache。这也是实现消息队列的一个前提。

## 41讲 |一致性哈希

一致性hash算法（DHT）通过减少影响范围的方式解决了增减服务器导致的数据散列问题，从而解决了分布式环境下负载均衡问题，如果存在热点数据，那么通过增添节点的方式，对热点区间进行划分，将压力分配至其他服务器。重新达到负载均衡的状态。