1、redis的特点：

基于内存、单线程、IO多路复用

2、IO多路复用：

经典的Reactor设计模式，也称为异步阻塞IO，



用户首先将需要进行IO操作的socket添加到select函数中，然后阻塞等待select系统调用返回。当数据到达时，socket被激活，select函数返回。用户线程正式发起read请求，读取数据并继续执行。用户可以注册多个socket，然后不断地调用select读取被激活的socket，即可达到在同一个线程内同时处理多个IO请求的目的。

通过Reactor的方式，用户线程注册事件处理器之后可以继续执行做其他的工作（异步），而Reactor线程负责调用内核的select函数检查socket状态。当有socket被激活时，则通知相应的用户线程，执行handle\_event进行数据读取、处理的工作。由于select函数是阻塞的，因此多路IO复用模型也被称为异步阻塞IO模型。注意，这里的所说的阻塞是指select函数执行时线程被阻塞，而不是指socket。

**3 、redis单线程问题**

    单线程指的是网络请求模块使用了一个线程（所以不需考虑并发安全性），即一个线程处理所有网络请求，其他模块仍用了多个线程。采用单线程,避免了不必要的[上下文切换](https://www.baidu.com/s?wd=%E4%B8%8A%E4%B8%8B%E6%96%87%E5%88%87%E6%8D%A2&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)

4、redis数据结构

String、List、Set、hash、zSet

5、spring的注入方式

常用的注入方式主要有三种：构造方法注入，setter注入，基于注解的注入。

<https://blog.csdn.net/yinni11/article/details/80212614>

6、java虚拟机参数指令

-Xms20m 堆最小值20MB

-Xmx20m堆最大值20MB

-Xss128k 栈容量128KB

-XX:MaxPermSize=10M 永久代(方法区)最大值为10MB

7、类加载过程（类的生命周期）

类加载过程包括 加载、验证、准备、解析、初始化五个阶段。

加载：（三步）

a、通过类全限定名获取类的二进制字节流。

b、将该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。

c、在内存中生成一个代表该类的java.lang.Class对象，作为方法区这个类的访问入口。

验证：验证阶段的第一步，为了确保Class文件的字节流符合虚拟机的安全。

准备：为类变量（static修饰）分配内存并设置初值（0值）。

解析：将常量池内的符号引用替换为直接引用的过程。

符号引用: 用一组符号来描述所引用的目标，符号引用与虚拟机实现的内存布局无关，引用的目标不一定已经加载到内存中。

直接引用：可以是直接指向目标的指针、相对偏移量或句柄。直接引用和虚拟机实现的内存布局相关，如果有直接引用，那应用的目标已存在在内存中。

初始化：类加载过程的最后一步，执行类构造器<clinit>()方法的过程。

8、类加载器

启动类加载器（Bootstrap ClassLoader）:

加载<JAVA\_HOME>\lib目录中的

扩展类加载器（ExtensionClassLoader）：

加载<JAVA\_HOME>\lib\ext目录中的

应用程序类加载器（ApplicationClassLoader）

加载ClassPath所指定的类库，也称系统类加载器

双亲委派模型：

除启动类加载器外，其余类加载器都有父加载器，关系以“复用”实现而不是继承。

当一个类加载器收到类加载的请求时，它会先把这个请求委派给父类加载器来完成，所有的加载请求都会送到顶层的启动类加载器，只有当父类加载器反馈无法加载时，子加载器才会尝试加载。双亲委派模型使得java类随着其类加载器具有了优先级的层次关系。例如，Object类被任何一个类加载器加载时都是同一个类。

9、垃圾回收判定

可达性分析算法：当一个到GC Roots没有任何应用链相连时，对象会被判定为可回收的对象。

引用类型：

强引用（引用还在就不会被回收）

软引用（有用但并非必需的对象，系统内存溢出前才会回收这类对象）

弱引用（非必需对象，只能生存到下一次垃圾收集发生之前）

虚引用（虚引用不影响对象生存时间，也无法取得对象的实例，唯一作用是在对象被回收时收到系统通知）

垃圾回收时要进行两次标记，可达性标记第一次，第二次标记finalize()方法，若对象覆写该方法并在方法中重建引用链，则对象不会被回收。

10、垃圾收集算法

虚拟机都采用“分代收集”算法，将内存分为新生代与老年代。

(1)标记-清除算法(mark-sweep)

首先标记出所有需要回收的对象，标记完成后统一回收，缺点：会产生不连续的内存碎片。

(2)复制算法(copying)

hotspot将新生代内存分为Eden空间和两块Survivor空间，大小比为8：1，每次使用Eden和其中一块Survivor，当回收时，将Eden和Survivor中还存活的对象复制到另一块Survivor上，最后清理掉Eden与刚才的Survivor。当Survivor空间不够时，将使用老年代进行分配担保。

(3)标记-整理算法(mark-compact)

对老年代使用，标记过程与标记-清除算法一样，但后续不是直接对对象回收而是让存活的对象向一端移动，清理掉边界以外的内存。

11、垃圾收集器



（1）Serial收集器

Serial收集器是一个单线程的垃圾收集器，特点如下：

·仅仅使用一个线程完成垃圾收集工作；

·在垃圾收集时必须暂停其他所有的工作线程，直到垃圾收集结束；

·Stop the World是在用户不可见的情况下执行的，会造成某些应用响应变慢；

·使用复制算法；

Serial收集器的工作流程如下图：



虽然如此，Serial收集器依然是虚拟机运行在Client模式下的默认新生代收集器。它的优点同样明显：简单而高效（单个线程相比），并且由于没有线程交互的开销，专心做垃圾收集自然课获得最高的单线程效率。

（2）ParNew收集器

ParNew收集器其实是Serial收集器的多线程版本，与Serial不同的地方就是在垃圾收集过程中使用多个线程，剩下的所有行为都一样。ParNew收集器也使用复制算法。ParNew收集器的工作流程如下图：



ParNew收集器看似没有多大的创新之处，但却是许多运行在Server模式下的虚拟机中首选的新生代收集器，因为，除了Serial收集器外，目前只有ParNew收集器能够与CMS收集器配合工作，而CMS收集器是HotSpot在JDK 1.5时期推出的具有划时代意义的垃圾收集器。可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数。

（3）Parallel Scavenge收集器

Parallel Scavenge收集器和ParNew类似，是一个使用复制算法的并行多线程收集器。不过和ParNew不同的是，Parallel Scavenge收集器的关注点不同。

CMS等收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目的则是达到一个可控制的吞吐量。吞吐量就是CPU用于运行用户代码的时间与CPU总消耗时间的比值，即吞吐量=运行用户代码时间/（运行用户代码时间+运行垃圾收集时间）。如果虚拟机一共运行100分钟，垃圾收集运行了1分钟，那么吞吐量就是99%。

停顿时间越短就越适合与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可以高效的利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

Parallel Scavenge收集器提供了两个参数来精确控制吞吐量，分别是控制最大垃圾收集停顿时间的-XX:MaxGCPauseMillis参数以及直接设置吞吐量大小的-XX:GCTimeRatio参数。

Parallel Scavenge收集器也叫吞吐量优先收集器，它还有一个参数-XX:UseAdaptiveSizePolicy，这是一个开关参数，当这个参数打开后，就不需要手工指定新生代的大小（-Xmn）、Eden和Survivor的比例（-XX:SurvivorRatio）、晋升老年代对象年龄（-XX:PretenureSizeThreshold）等细节了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最适合的停顿时间或最大的吞吐量，这叫GC自适应的调节策略。这也是Parallel Scavenge收集器和ParNew收集器的一个重要区别。

（4）Serial Old收集器

Serial Old是Serial的老年版本，在Serial的工作流程图中可以看到，Serial Old收集器也是一个单线程收集器，使用“标记-整理”算法。这个收集器主要给Client模式下的虚拟机使用。如果在Serve模式下，它有两个用途：一个是在JDK 1.5之前的版本中与Parallel Scavenge收集器搭配使用；另一个就是作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用。这个收集器的工作流程在Serial的后半部分有所体现。

（5）Parallel Old收集器

Parallel Old收集器是Parallel Scavenge收集器的老年版本，它也使用多线程和“标记-整理”算法。这个收集器是在JDK 1.6开始提供。

在注重吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge加Parallel Old收集器的组合。

（6）CMS收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。在重视响应速度和用户体验的应用中，CMS应用很多。

CMS收集器使用“标记-清除”算法，运作过程比较复杂，分为4个步骤：

初始标记（CMS initial mark）

并发标记（CMS Concurrent mark）

重新标记（CMS remark）

并发清除（CMS Concurrent Sweep）

其中，初始标记和并发标记仍然需要Stop the World、初始标记仅仅标记一下GC Roots能直接关联到的对象，速度很快，并发标记就是进行GC RootsTracing的过程，而重新标记阶段则是为了修正并发标记期间因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段长，但远比并发标记的时间短。

由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以整体上说，CMS收集器的内存回收过程是与用户线程一共并发执行的。下图是流程图：



CMS的优点就是并发收集、低停顿，是一款优秀的收集器。不过，CMS也有缺点，如下：

CMS收集器对CPU资源非常敏感。CMS默认启动的回收线程数是（CPU数量+3）/4，当CPU个数大于4时，垃圾收集线程使用不少于25%的CPU资源，当CPU个数不足时，CMS对用户程序的影响很大；

CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败而导致另一次Full GC；

CMS使用标记-清除算法，会产生内存碎片；

（7）G1收集器

G1（Garbage first）收集器是最先进的收集器之一，是面向服务端的垃圾收集器。与其他收集器相比，G1收集器有如下优点：

并行与并发：有些收集器需要停顿的过程G1仍然可以通过并发的方式让用户程序继续执行；

分代收集：可以不使用其他收集器配合管理整个Java堆；

空间整合：使用标记-整理算法，不产生内存碎片；

可预测的停顿：G1除了降低停顿外，还能建立可预测的停顿时间模型；

G1中也有分代的概念，不过使用G1收集器时，Java堆的内存布局与其他收集器有很大的差别，它将整个Java堆划分为多个大小相等的独立区域（Region），G1收集器之所以能建立可预测的停顿时间模型，是因为它可以有计划的避免在整个Java堆中进行全区域的垃圾收集。G1跟踪各个Region里垃圾堆积的价值大小（回收所获得的空间大小以及回收所需要的时间的经验值），在后台维护一个优先列表，每次优先收集价值最大的那个Region。这样就保证了在有限的时间内尽可能提高效率。