GC的历史（1960年）其实比Java更加漫长，并不是Java的伴随产品。

尽管内存的动态分配技术和内存回收技术已经相当成熟， 但是当我们需要排查各种内存溢出，内存泄漏问题时，当垃圾收集已经成为高并发量的瓶颈时，我们就需要对这些自动化的技术实行必要的监控和调节

GC的主要问题是：

1）哪些内存需要回收?

2）何时回收?

3）如何回收?

回顾第二章， 我们知道pc counter，虚拟机栈，本地方法栈（栈帧多在编译时即确定）随着线程而生，随线程而死，有条不紊，不需要考虑垃圾回收。但是在Java堆和方法区不同，一个接口的多个实现类可能要求内存不同，一个方法的多个分支也可能不同，我们只能在程序运行时才能知道创建哪些对象。这两个部分才是我们关注的垃圾回收区域。

3.1 对象死了吗?

传统的引用计数器方法，并不实用，因为，对象之间可以相互引用，导致永远无法释放。

在主流的程序语言中，我们都是采用的可达性分析，来判定对象的死活。

首先确定一个 GC ROOT ：

a.虚拟机栈（栈帧中的本地变量表）引用对象

b.方法区中的静态属性， 或是常量， 引用的对象

c. Native方法中引用的对象。

原理：通过一系列的GC ROOT的对象的点出发，往下开始搜索，所走过的路径，叫做引用链，若一个对象不在这个引用链上，则称对象是不可到达的。

闲聊：再谈谈引用。

四大引用：

a. 强引用。在代码中普遍存在的，类似new出来的引用。只要强引用还在，对象就永远不会被GC器回收掉

b.软引用。有用但并非必须的对象。只有在要发生内存溢出异常之前，才会把这些对象进行第二次回收，如果回　收了还不够，只好报异常了

c.弱引用。 只能生存到下一次 GC之前。当 GC器工作时，都会被回收掉

d.虚引用。对象的存在与否完全不受引用的影响，也无法通过一个虚引用来取得一个对象实例。其存在的唯一目的：这个对象被回收时能打个报告通知。

生存还是死亡?

一个对象在判定不可达之后，只是判处它“死缓”。要宣告一个对象死亡，至少经历两次标志。

1）对象在第一次可达性分析中，发现不可达，则会被第一次标志，并且进行筛选此对象有无必要执行finalize()方法。没必要执行，则直接回收了

2）如果对象没有覆盖finalize() || finalize()已经被JVM执行过一次， 则无必要执行finalize()。然后把它放入到一个F-Queue之中，并且稍后由JVM自动创建一个低优先级的Finalizer()线程去执行它（所谓的“执行”，并不保证执行完，因为finalize()可能缓慢，死循环，则会卡死F-Queue中其他对象）。

3）finalize()方法是对象最后一次死里逃生的机会，只要在finalize()重新连接到引用链上的任何一个对象，则可以移出“即将被回收”的集合；否则就加入“即将被回收”的集合，然后等死了。当然这种自救的机会只能有一次，一个对象的finalize()方法只能被JVM至多使用一次。

实际上，我们会尽量避免使用finalize()中复活本来该死的对象。使用这种方法，完全可以用try-finally()，或者其他方式复活。

3.2 回收方法区

方法区被叫做永久代，但是并不是真的没有垃圾回收，一直存在，只是在这部部分回收垃圾“性价比”很低，相比于在新生代一次就能释放
70%~95%的内存。

永久代，主要回收两部分：

1）废弃常量： 与回收堆类似，只要一个值，不再有指向它的常量，就可以回收

2）回收类： 要判断一个类是否是无用的，要求较多

a.该类的所有实例已经回收

b.加载该类的ClassLoader已经回收

c. 该类对应的java.lang.class对象已经不在任何地方被引用， 无法在任何地方使用反射机制访问该类的方法。

在大量使用反射，动态代理，或者像动态生成JSP这类频繁自定义classLoader（）的场景都需要JVM具备类卸载的功能，
以保证永久代不会溢出

3.3 垃圾回收算法

GC算法的实现设计大量细节，我们只介绍几种算法

1)标志 - 清除 算法 Mark- Sweep 算法 ==> 效率低 ，碎片化严重

2）复制算法。 现代的商业JVM采用的这种。新生代中的对象，98%都是朝生夕死的“短命鬼”，所以并不要求等比例的分配内存

我们把新生代内存分成 Eden，From survivor， To survivor（8:1:1的内存比例）。每次使用Eden和From survivor中还存活的对象一次性copy 到 To survivor中，然后清理掉这两块原来的区域。之后再使用，Eden 和 To survive,即原来的From survive变成了现在的To survivor，原来的To survivor变成了现在的From survivor。

这样子做，只有10%的内存空间被浪费掉， 其实只符合绝大的情况（98%），但是每次回收时，如果survivor空间不够，我们就必须使用老年代进行保证。

3）标志 - 整理算法。在1）的基础上，采用内存紧凑技术，避免碎片化

4）分代收集算法 。 把Java堆分成新生代和老年代，在新生代每次Minor GC都会有大量对象死亡，因此词用2）算法。在老年代，对象存活率很高，就采用1）或者3）

3.4 HotSpot的算法实现 && 垃圾回收器

采用一种OopMap的数据结构记录安全点（SafePoint），在这个点必须要Stop The World，只有在这些安全点才能支持停顿下来实现GC。当GC需要中断线程时候，设置一个标志，然后让所有的线程取轮询这个标志，然后检查到了这个中断标志就自己中断挂起。但是在这个点，有可能有些线程本来就没运行， 所以可以采用一段区域的安全领域

垃圾回收器： 不同版本，不同厂家实现不同，还可以自己组合新生代和老年代的GC器

新生代:

1） Serial GC： 单线程， 并且执行时造成所有的用户Thread暂停，直至其收集结束。即Stop The World。

2）ParNew GC： 1)的多线程版本，运行在Sever模式下的JVM首选的新生代收集器

3）Parallel Scavenge GC： 追求吞吐量最大化的GC

老年代:

4）Serial Old GC : 1)的老版本， 和1）差不多。

5）Parallel Old GC :使用多线程和标志-整理算法 IDK1.6

6）CMS GC （concurrent Mark Sweep）:获取最短停顿时间的GC ， 绝大部分的Java应用都集中在互联网上，并发收集，低停顿

7）G1 GC（Garage First）:面向服务器， JDK1.7中发行，最新成果

3.5内存分配与回收策略

对象的分配， 往大方向说，在堆上；往小了说，主要在新生代的Eden区上；如果启动了TLAB，将按线程优先级在TLAB上分配。极少情况会直接分配到老年代。

1） 对象优先在 Eden分配 。

大数下，对象在新生代Eden上分配。当Eden没有足够空间时， JVM将发起一次Minor GC。

2） 大对象直接进入老年代

大对象（需要大量连续空间，比如很长的字符串和数组），并且“短命的”我们要尽量避免（因为一旦出现，就导致内存还有不少空间时候就提前触发了GC）。-XX：PretenureSizeThreshold来设置大于这个值的对象可以直接分配在老年代中。

3）长期存活的对象将进入老年代

JVM给每个对象定义一个对象年龄计数器Age。 在Eden中，对象获得新生， 如果经历一次Minor GC之后还活着， 就把它复制到Survivor中， 并且设置Age = 1 。之后在这个区域， 对象每“熬过”一次Minor GC， 就会执行一次 Age++,直到默认的15，就会晋身到老年代中， 这个默认的15岁，可以通过 -XX： MaxTenuringThreshold设置

4）动态对象年龄判定

并非一定要到达MaxTenuringThreshold（默认15）才能晋身到老年代 。 如果Survivor空间中相同年龄所有对象大小的总和 >= 10%的内存的一块Survivor空间， 那么Age >=此对象的年龄的对象就可以晋身到老年代

5）内存分配担保

首先介绍下“冒险”：

新生代使用Minor GC 后仍然存活的对象 （绝大数 <= 2% ）， 要复制到只占10%空间的Survivor空间。要是存活的对象总大小 > 10%的一块Survivor空间怎么办？那就只好copy到老年代的连续剩余空间中了 。

在发生一次 Minor GC之前， JVM首先会检查一次老年代中连续的最大空间大小（设为X） 是否 > 新生代所有对象的总空间。

a. yes , 则Minor GC 肯定是安全的，直接运行就是

b . no. JVM查看HandlePromotionFailure参数是否允许担保失败 。

b1.允许 && 老年代中连续的最大空间大小 > 历次晋身到老年代的对象的平均大小， 则有必要尝试一次 Minor GC

b2 . 否则 ， 执行一次 Full GC（出现一次该Full GC 往往至少伴随一次Minor GC），其速度是Minor GC的 10倍以上