无论是普通的gc还是full gc都会发生stop the world，发生的原因是查询那些是需要回收的对象，但是如果不停止的话，可能会发生错误回收，销毁掉又加入内存的对象等

1. 枚举跟节点

从可达性分析中从GC roots节点找引用为例，可作为GC roots的节点基本上是全局性的引用与执行上下文、如果要逐个检查引用，必然肯定浪费时间

可达性分析对执行时间的敏感还体现在gc停顿上，因为这项分析工作需要在一个能确保一致性的快照中进行---这里的一致性的意思就是整个分析期间整个系统执行看起来冻结在某个时间点，不可以出现分析过程中对象引用还在不断的变化情况，该点不满足的话分析结果的准确性就没法得到保障。**这点是导致gc进行时必须暂停所有java执行线程的其中一个重要原因。**由于目前主流的Java虚拟机使用的是准确式gc，当执行系统停顿下来之后，并不需要一个不漏的监察执行上下文和全局引用位置，虚拟机应当有办法知道哪些地方存放的是对象的引用，在hotspot的实现中，是使用一组OOpMap的数据结构来达到这个目的

2.安全点

在OOpMap的协助下，hotspot可以快速且准确的完成gc roots的枚举，但可能导致引用关系变化的指令比较多，如果为每一个指令生成oopMap的话，那么会需要大量的额外空间，这样gc的空间成本就会很高

实际上，hotspot的确没有为每条指令生成oopMap,只是在特定的位置记录这些信息，这些信息被称为安全点（SafePoint），safePoint的选定不能太少，以至于让gc等待时间太久，也不能设置的太频繁，以至于增加运行负载。这些安全点的设置是一让程序”是否具有让程序长时间执行的特征“为标准选定的。”长时间执行“最显著的特征就是指令序列的复用，例如方法调用、循环跳转、异常跳转等、所以具备这些功能的指令才会产生safe point

对于safe point，另一个问题就是如何在gc发生时让所有的线程都跑到安全点停顿下来。这里有两个方案：

（1）抢先式中端：不需要线程代码主动配合，当gc发生，首先把所有的线程终端，如果发现中断的位置不在安全点上，就恢复线程，让他跑到安全点上（几乎没有虚拟机使用这个方式响应jvm）

（2）主动式中断：不直接对线程进行操作，仅仅简单的设置一个标志，各个线程执行时主动去轮训这个标志，发信中断标志为真时就自己中断挂起，轮训标志

的地方和安全点是重合的另外再加上创建对象需要分配的内存的地方

3.安全区域

使用安全点似乎已经完美的结局了如何进入Gc的问题，但是实际情况并不一定，安全点机制保证程序执行时，不太长的时间就进入到可达的gc的安全点，但是程序如果不执行呢，所谓的程序不执行就是没有分配cpu时间，典型的就是线程出于sleep状态或者blocked状态，这个时候线程无法响应jvm中断请求，jvm显然不太肯能等待线程重新分配Cpu时间，对于这种情况，我们使用安全区来处理
安全区域是指在一段代码中，引用关系不会发生变化，这个区域的任何地方开始GC都是安全的，可以把安全区看做安全点的扩展区来解决
当线程执行到安全区域中的代码时，首先标识自己已经进入了安全区，那样当在这段时间里，JVM要发起GC时，就不用管标识自己为安全区域状态的线程了。
当线程要离开安全区域时，他要检查系统是否完成了根节点枚举，如果完成了，那线程就继续执行，否则他就必须等待，直到收到可以安全离开安全区域的信号为止