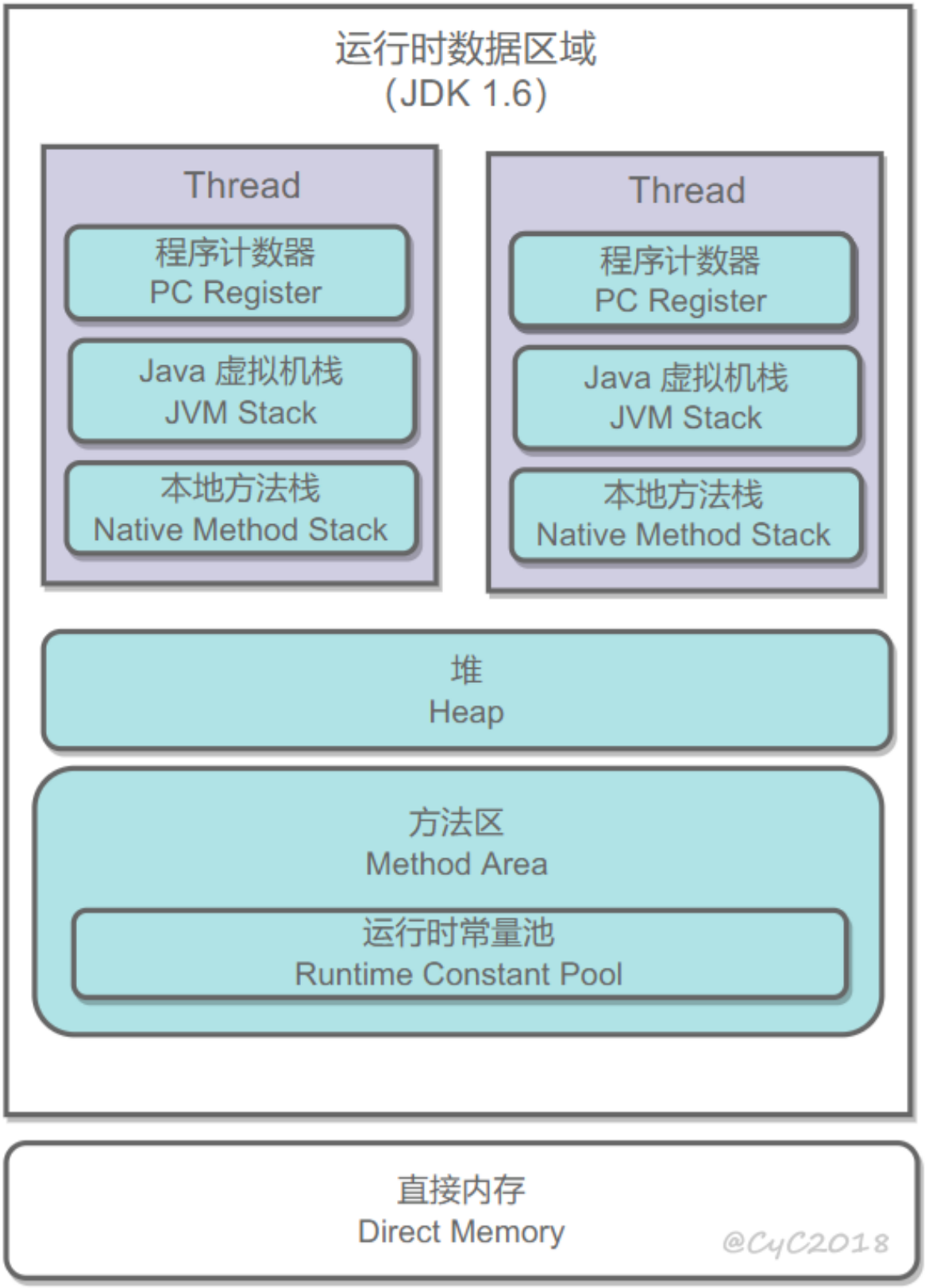
一、运行时数据区域

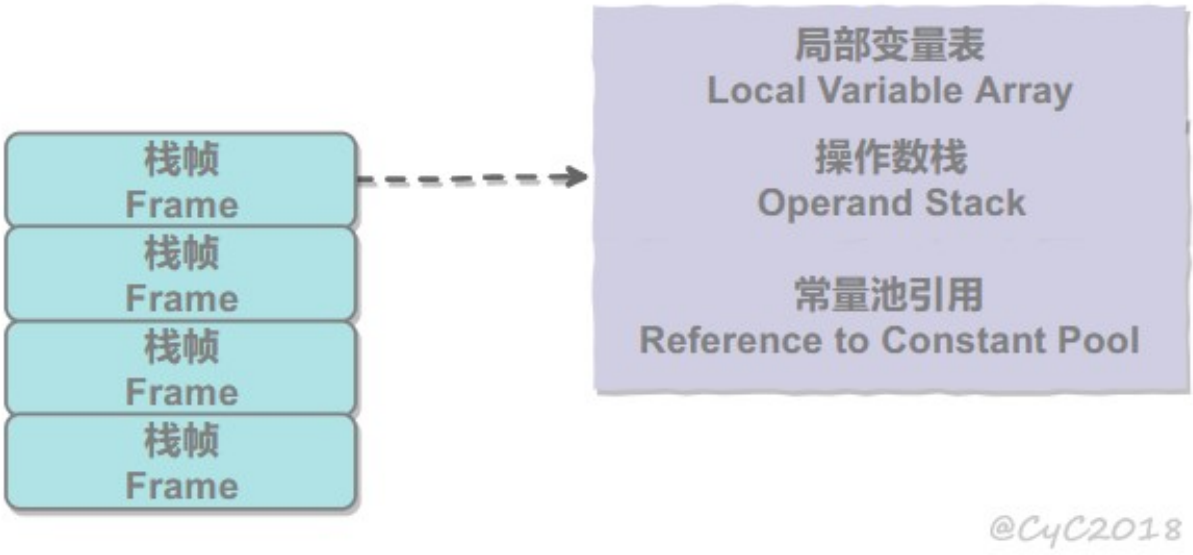


程序计数器

记录正在执行的虚拟机**字节码指令的地址**（如果正在执行的是**本地方法则为空**）。

Java虚拟机栈

每个 Java 方法在执行的同时会创建**一个栈帧**用于存储**局部变量表、操作数栈、常量池引用**等信息。从方法调用直至执行完成的过程，就对应着一个栈帧在 Java 虚拟机栈中入栈和出栈的过程。



可以通过 -Xss 这个虚拟机参数来指定**每个线程的 Java 虚拟机栈内存大小**：

```
java -Xss512M HackTheJava
```

该区域可能抛出以下异常：

- 当**线程请求**的栈**深度超过最大值**，会抛出 StackOverflowError 异常；
- 栈进行动态扩展时如果**无法申请到足够内存**，会抛出 OutOfMemoryError 异常。

java中的基本数据类型一定存储在栈中的吗?

错误，**存储位置取决于其在哪里声明的**

1. 在方法中声明的变量，即该变量是局部变量

每当程序调用方法时，系统都会为该方法建立一个JVM栈帧，其所在方法中声明的变量就放在方法栈中，**当方法结束系统会释放相应栈帧**，其对应在该方法中**声明的变量随着栈的销毁而结束**，这就是局部变量只能在方法中有效的原因。

方法中的基本类型变量：变量名及值（变量名及值是两个概念）是放在JAVA虚拟机栈

方法中的引用类型变量：对象引用在JAVA虚拟机栈中，所指向的对象是放在**堆内存**中的

2.在类中声明的变量是成员变量，也叫全局变量（类变量），放在堆中的（因为全局变量不会随着某个方法执行结束而销毁）

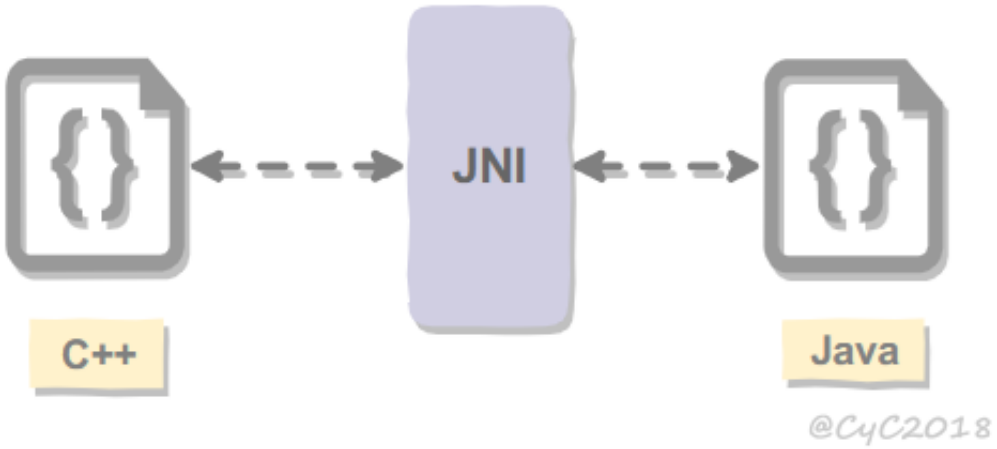
类的基本类型成员：变量名及其值放在**堆内存**中的

类的引用类型成员：引用变量名和对应的对象仍然存储在相应的**堆中**

本地方法栈

本地方法栈与 Java 虚拟机栈类似，它们之间的区别只不过是**本地方法栈为本地方法服务**。

本地方法一般是用其它语言（C、C++ 或汇编语言等）编写的，并且被编译为**基于本机硬件和操作系统的程序**，对待这些方法需要特别处理。



堆

所有（并不是绝对，JIT逃逸分析）**对象**都在这里分配内存，是垃圾收集的主要区域（"GC 堆"）。

现代的垃圾收集器基本都是采用**分代收集算法**，其主要的思想是针对不同类型的对象采取不同的垃圾回收算法。可以**将堆分成两块**：

- 新生代（Young Generation）
- 老年代（Old Generation）

堆**不需要连续内存**，并且可以**动态增加其内存**，增加失败会抛出 OutOfMemoryError 异常。

可以通过 -Xms 和 -Xmx 这两个虚拟机参数来指定一个程序的**堆内存**大小，第一个参数设置初始值，第二个参数设置最大值。

```
java -Xms1M -Xmx2M HackTheJava
```

方法区（1.8后移至本地内存元空间，之前是永生代）

用于存放**已被加载的类信息、常量、静态变量、即时编译器编译（JIT）后的代码**等数据。

和堆一样**不需要连续的内存**，并且可以**动态扩展**，动态扩展失败一样会抛出 OutOfMemoryError 异常。

对这块区域进行垃圾回收的主要目标是对**常量池的回收和对类的卸载**，但是一般比较难实现。

HotSpot 虚拟机把它当成**永久代**来进行垃圾回收。但**很难确定永久代的大小**，因为它受到很多因素影响，并且每次 **Full GC 之后永久代的大小都会改变**，所以经常会抛出 OutOfMemoryError 异常。为了更容易管理方法区，从 **JDK 1.8 开始，移除永久代**，并把方法区移至元空间，它位于**本地内存中**，而不是虚拟机内存中。

运行时常量池

运行时常量池是**方法区的一部分**。

Class 文件中的常量池（**编译器生成的字面量和符号引用**）会在类加载后被放入这个区域。

除了在编译期生成的常量，还允许动态生成，例如 String 类的 intern()（native方法）。

直接内存

在 **JDK 1.4** 中新引入了 NIO 类，它可以使用 **Native 函数库直接分配堆外内存**，然后通过 Java 堆里的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在堆内存和堆外内存来回拷贝数据。

二、垃圾收集

垃圾收集主要是针对**堆和方法区**进行。程序计数器、虚拟机栈和本地方法栈这三个区域属于线程私有的，只存在于线程的生命周期内，**线程结束之后就会消失**，因此不需要对这三个区域进行垃圾回收。

GC主要回答了以下三个问题：

- **哪些内存需要回收?**
- **什么时候回收?**
- **如何回收?**

表1：GC优化需要考虑的JVM参数

类型	参数	描述
堆内存大小	`-Xms`	启动JVM时堆内存的大小
	`-Xmx`	堆内存最大限制
新生代空间大小	**`-XX:NewRatio`**	新生代和老年代的内存比
	`-XX:NewSize`	新生代内存大小
	`-XX:SurvivorRatio`	Eden区和Survivor区的内存比

在进行GC优化时最常用的参数是`-Xms`,`-Xmx`和`-XX:NewRatio`。`-Xms`和`-Xmx`参数通常是必须的，**所以`NewRatio`的值将对GC性能产生重要的影响。**

GC监控

jmap（JVM Memory Map for Java）

jmap用于生成堆快照（**heapdump**）

堆转储（heap dump）是一个用来检查Java内存中的对象和数据的内存文件。该文件可以通过执行JDK中的`jmap`命令来创建。在创建文件的过程中，所有Java程序都将暂停，因此，**不要在系统执行过程中创建该文件**。

通过JVM启动时加入启动参数 –XX:HeapDumpOnOutOfMemoryError参数比较常用

jmap的作用不仅仅是为了获取dump文件，还可以**用于查询finalize执行队列**、**Java堆和永久代**的详细信息，如空间使用率、垃圾回收器等

jstat（JVM Statistics Monitoring Tools）

监控虚拟机的各种运行状态信息，如类的装载、内存、垃圾回收、JIT编译器等

在运行中的Web应用服务器（Web Application Server,WAS）上查看GC状态的最佳方式就是使用`jstat`命令

jps（JVM Process Status Tools）

而他的功能和**ps**的功能类似，可以列举正在运行的**饿虚拟机进程并显示虚拟机执行的主类**以及这些**进程的唯一ID**（LVMID，对应本机来说和PID相同）

jstack（JVM Stack Trace for java）

JVM当前时刻的线程快照，又称**threaddump**文件，它是JVM当前每一条线程正在执行的堆栈信息的集合。生成线程快照的主要目的是为了定位线程出现长时间停顿的原因，如线程死锁、死循环、请求外部时长过长导致线程停顿的原因。通过jstack我们就可以知道哪些进程在后台做些什么？在等待什么资源等！

判断一个对象是否可回收

1. 引用计数算法

为对象添加一个引用计数器，当对象增加一个引用时计数器加 1，引用失效时计数器减 1。引用计数为 0 的对象可被回收。

在两个对象出现循环引用的情况下，此时引用计数器永远不为 0，导致无法对它们进行回收。**正是因为循环引用的存在，因此 Java 虚拟机不使用引用计数算法**。

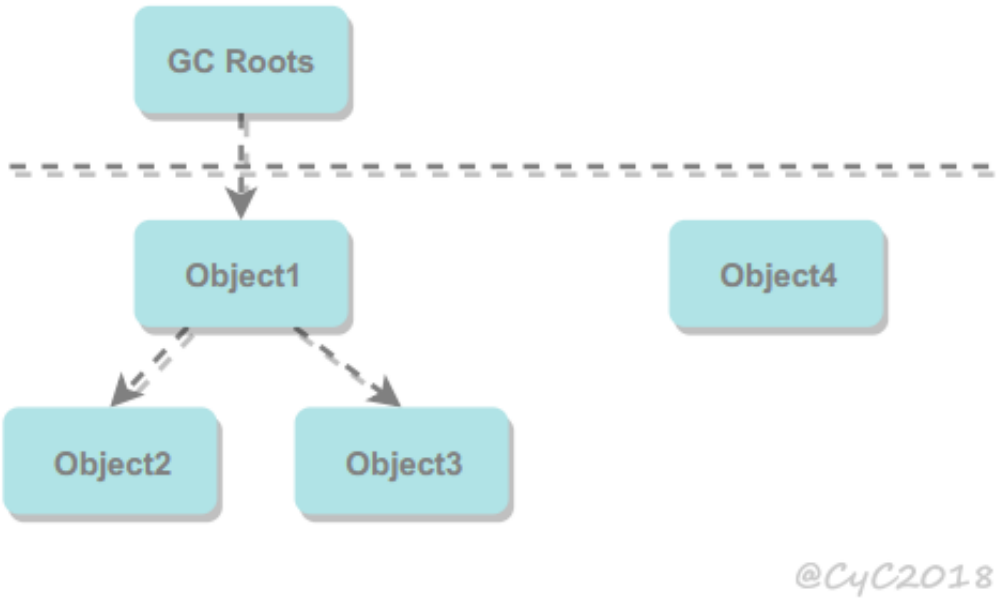
```
public class Test {

    public Object instance = null;

    public static void main(String[] args) {
        Test a = new Test();
        Test b = new Test();
        a.instance = b;
        b.instance = a;
    }
}
```

2. 可达性分析算法

以 GC Roots 为起始点进行搜索，可达的对象都是存活的，不可达的对象可被回收。

Java 虚拟机使用该算法来判断对象是否可被回收，GC Roots 一般包含以下内容：（**各种引用的对象**）

- 虚拟机栈中**局部变量表**中引用的对象
- 本地方法栈中 **JNI** 中引用的对象
- 方法区中类**静态属性**引用的对象
- 方法区中的**常量**引用的对象



3. 方法区的回收

因为方法区主要存放**永久代对象**，而永久代对象的回收率比新生代低很多，所以在方法区上进行回收性价比不高。

主要是对**常量池的回收和对类的卸载**。

为了避免内存溢出，在**大量使用反射和动态代理的场景都需要虚拟机具备类卸载功能**。

类的卸载条件很多，需要满足以下三个条件，**并且满足了条件也不一定会被卸载**：

- 该类**所有的实例**都已经被回收，此时堆中不存在该类的任何实例。
- 加载该类的 **ClassLoader** 已经被回收。
- 该类对应的 **Class 对象**没有在任何地方被引用，也就无法在任何地方通过反射访问该类方法。

4. finalize()

类似 C++ 的析构函数，用于关闭外部资源。但是 try-finally 等方式可以做得更好，并且该方法运行代价很高，不确定性大，**无法保证各个对象的调用顺序，因此最好不要使用**。

当一个对象可被回收时，如果需要执行该对象的 finalize() 方法，那么就有可能**在该方法中让对象重新被引用，从而实现自救。自救只能进行一次**，如果回收的对象之前调用了 finalize() 方法自救，**后面回收时不会再调用该方法**。

引用类型

无论是通过引用计数算法判断对象的引用数量，还是通过可达性分析算法判断对象是否可达，判定对象是否可被回收都与引用有关。

Java 提供了四种强度不同的引用类型。

1. 强引用

被强引用关联的对象不会被回收（**内存不够也不回收**）。

使用 new 一个新对象的方式来创建强引用。

```
Object obj = new Object();
```

2. 软引用（非常适合缓存）

被软引用关联的对象只有在**内存不够的情况下才会被回收**。

使用 SoftReference 类来创建软引用。

```
Object obj = new Object();
SoftReference<Object> sf = new SoftReference<Object>(obj);
obj = null;  // 使对象只被软引用关联
```

3. 弱引用

被弱引用关联的对象一定会被回收，也就是说它**只能存活到下一次垃圾回收发生之前**。

使用 WeakReference 类来创建弱引用。

```
Object obj = new Object();
WeakReference<Object> wf = new WeakReference<Object>(obj);
obj = null;
```

```
String abc = new String("abc");
    final ReferenceQueue<String> referenceQueue = new ReferenceQueue<String>();
    WeakReference<String> abcWeakRef = new WeakReference<String>(abc,
            referenceQueue);
    abc = null;
```

当对象abc被置null（或者内存不足）后，**实际对象被自动回收，引用对象被加到ReferenceQueue队列中**。此时调用abcWeakRef.get()返回null。WeakHashMap 使用 WeakReference 作为 key， 一旦没有指向 key 的强引用, **WeakHashMap 在 GC 后将自动删除相关的 entry**。

4. 幽灵引用（必须与 ReferenceQueue类一起使用，可用来替换finalize）

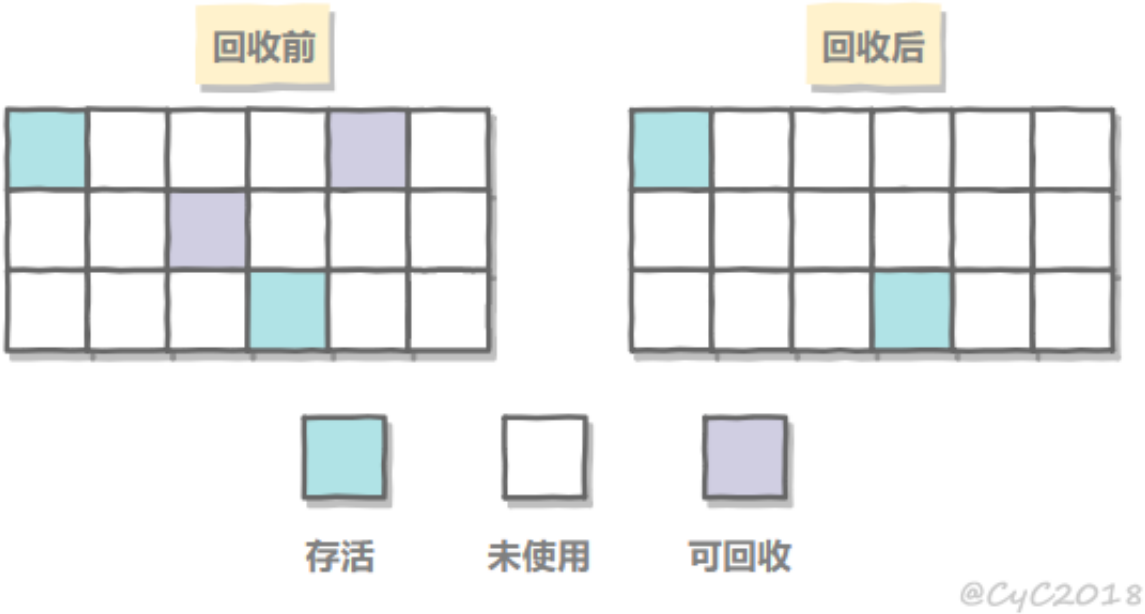
又称为虚引用或者幻影引用，一个对象是否有虚引用的存在，不会对其生存时间造成影响，也**无法通过虚引用得到一个对象，get方法一直返回null**。

为一个对象设置虚引用的**唯一目的是能在这个对象被回收时收到一个系统通知**。

使用幽灵引用，明明GC了，**可实际对象依旧没有被回收。需要手动调用引用对象的clear方法来回收。利用这个特效，可以用来做一些最后的清理工作**。例如ByteBuffer的Cleaner对象，JVM的Reference Handler线程会调用clean()方法。

垃圾收集算法（内存回收的方法论）

1. 标记 - 清除

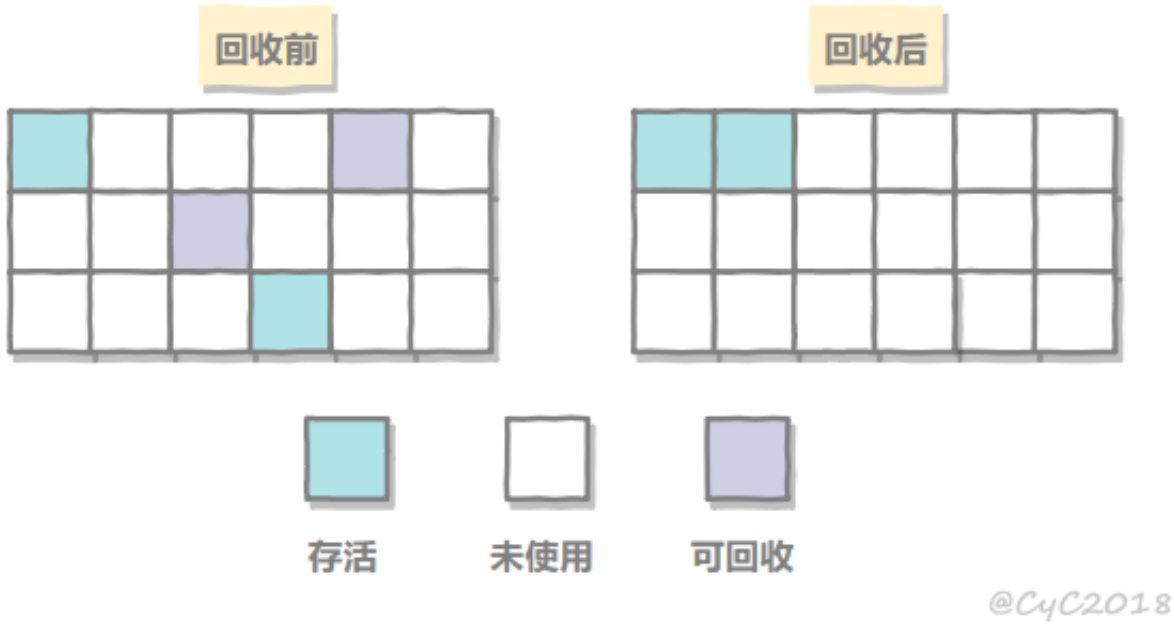


标记要回收的对象，然后清除。

不足：

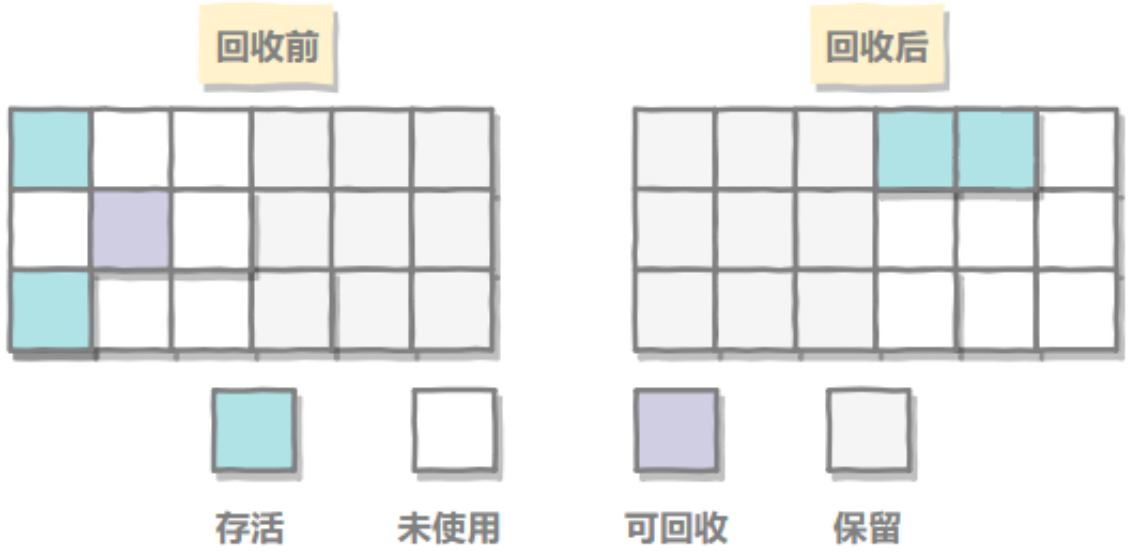
- 标记和清除过程效率都不高；
- 会产生**大量不连续的内存碎片，导致无法给大对象分配内存**。

2. 标记 - 整理



让所有存活的对象都向一端移动，然后直接**清理掉端边界以外的内存**。

3. 复制



将内存划分为大小相等的两块，每次只使用其中一块，当这一块内存用完了就将还存活的对象复制到另一块上面，然后再把使用过的内存空间进行一次清理。

主要不足是只使用了内存的一半。

现在的商业虚拟机都采用这种收集算法回收「新生代」，但是并不是划分为大小相等的两块，而是一块较大的 **Eden** 空间和**两块较小**的 **Survivor** 空间，每次使用 Eden 和其中一块 Survivor。在回收时，将 Eden 和 Survivor 中还存活着的对象全部复制到另一块 Survivor 上，最后清理 Eden 和使用过的那一块 Survivor。

HotSpot 虚拟机的 Eden 和 Survivor 大小比例默认为 8:1，保证了内存的利用率达到 90%。如果每次回收有多于 10% 的对象存活，那么一块 Survivor 就不够用了，此时需要依赖于**老年代进行空间分配担保，也就是借用老年代的空间存储放不下的对象。**

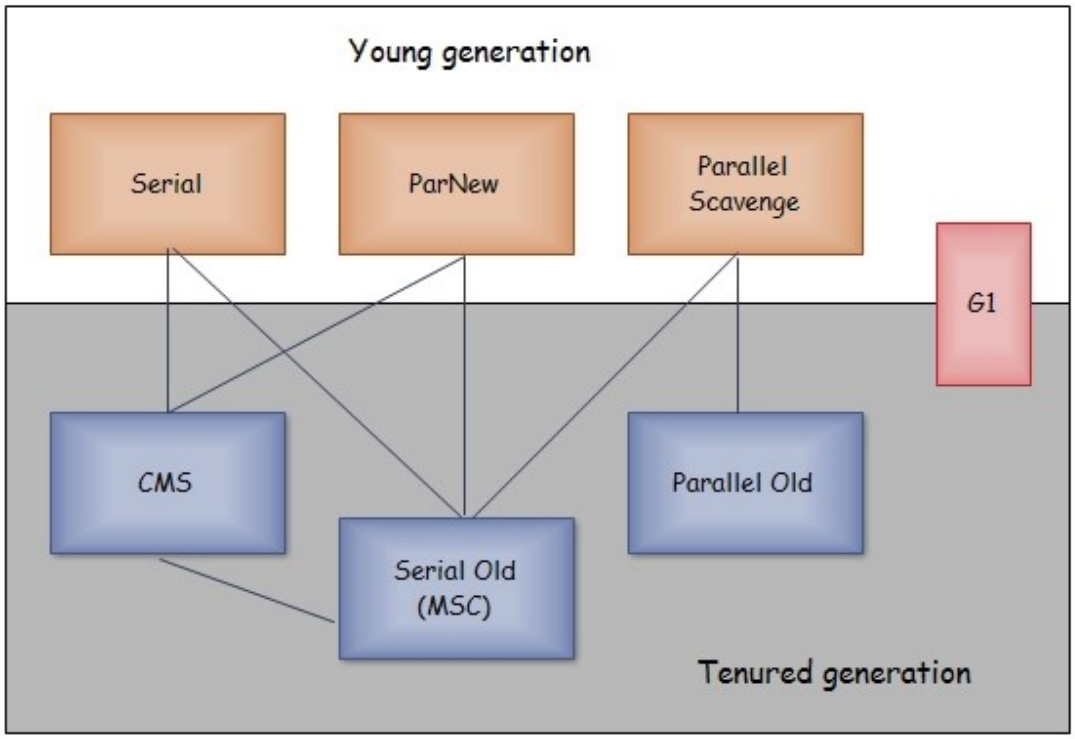
4. 分代收集（对于堆而言的分代！！！）

现在的商业虚拟机采用分代收集算法，它根据对象存活周期将内存划分为几块，不同块采用适当的收集算法。

一般将**堆**分为新生代和老年代。

- **新生代使用：复制算法**
- **老年代使用：标记 - 清除 或者 标记 - 整理 算法**

垃圾收集器（内存回收的具体实现）



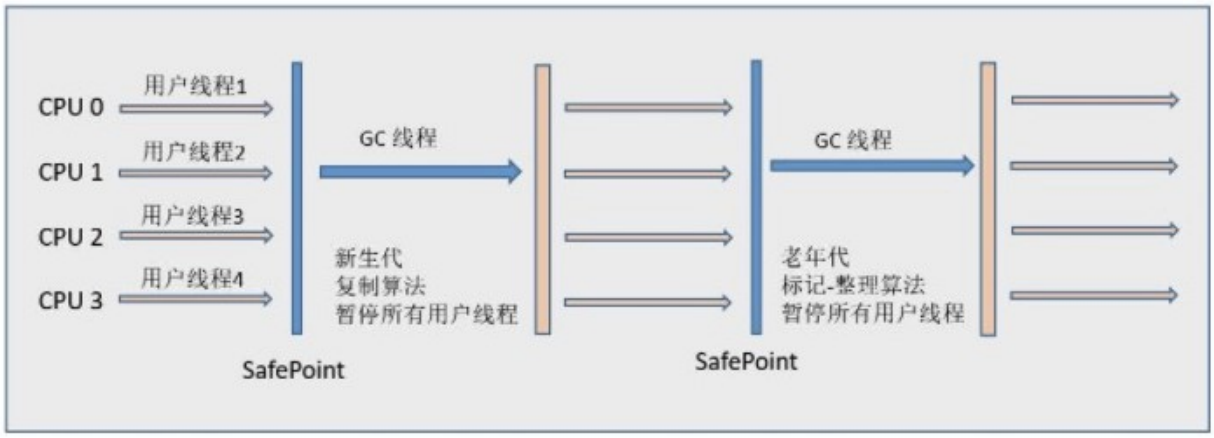
以上是 HotSpot 虚拟机中的 7 个垃圾收集器，连线表示垃圾收集器可以配合使用。

- **单线程与多线程**：单线程指的是垃圾收集器只使用一个线程，而多线程使用多个线程；
- **串行与并行**：串行指的是垃圾收集器与用户程序交替执行，这意味着在执行垃圾收集的时候需要停顿用户程序；**并行指的是垃圾收集器和用户程序同时执行。除了 CMS 和 G1 之外，其它垃圾收集器都是以串行的方式执行。**
- **并行（Parallel）**：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。
- **并发（Concurrent）**：指用户线程与垃圾收集线程同时执行（但不一定是并行的，可能会交替执行），用户程序在继续运行。而垃圾收集程序运行在另一个CPU上。

新生代收集器

1. Serial收集器

下图是Serial和SerialOld配合使用



Serial 翻译为串行，也就是说它以串行的方式执行。

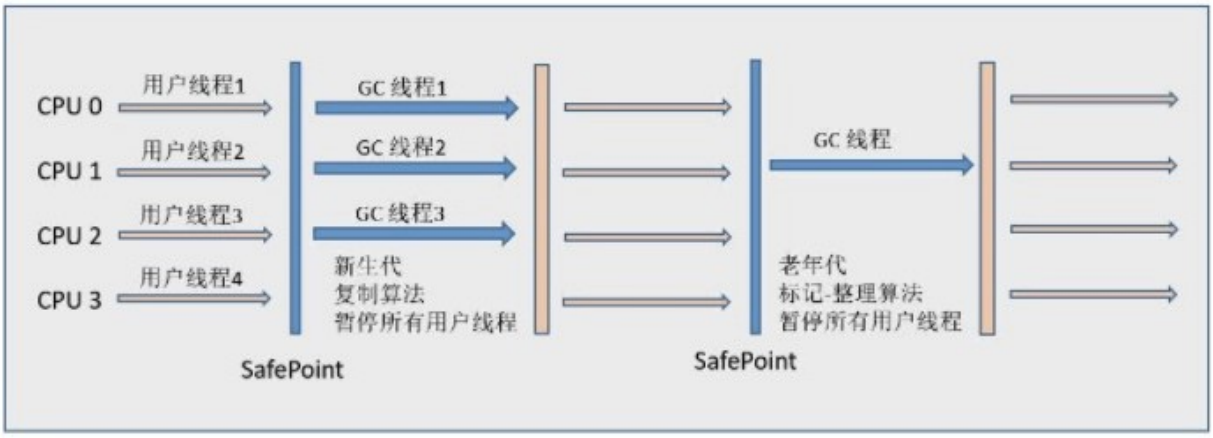
它是单线程的收集器，只会使用一个线程进行垃圾收集工作。

它的**优点是简单高效，在单个 CPU 环境下**，由于没有线程交互的开销，因此拥有**最高的单线程收集效率**。

它是 **Client 场景下的默认新生代收集器，因为在该场景下内存一般来说不会很大**。它收集一两百兆垃圾的停顿时间可以控制在一百多毫秒以内，只要不是太频繁，这点停顿时间是可以接受的。

2. ParNew收集器

ParNew收集器的工作过程如下图（老年代采用Serial Old收集器）：



它是 Serial 收集器的**多线程版本**。

它是 **Server 场景下默认的新生代收集器**，除了性能原因外，主要是**因为除了 Serial 收集器，只有它能与 CMS 收集器配合使用**。

3. Parallel Scavenge收集器（吞吐量优先）

与 ParNew 一样是多线程收集器。

其它收集器目标是尽可能缩短垃圾收集时用户线程的停顿时间，而**它的目标是达到一个可控制的吞吐量**，因此它被称为“**吞吐量优先**”收集器。这里的吞吐量指 **CPU 用于运行用户程序的时间占总时间的比值**。

吞吐量 = 运行用户代码时间 /（运行用户代码时间 + 垃圾收集时间）

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

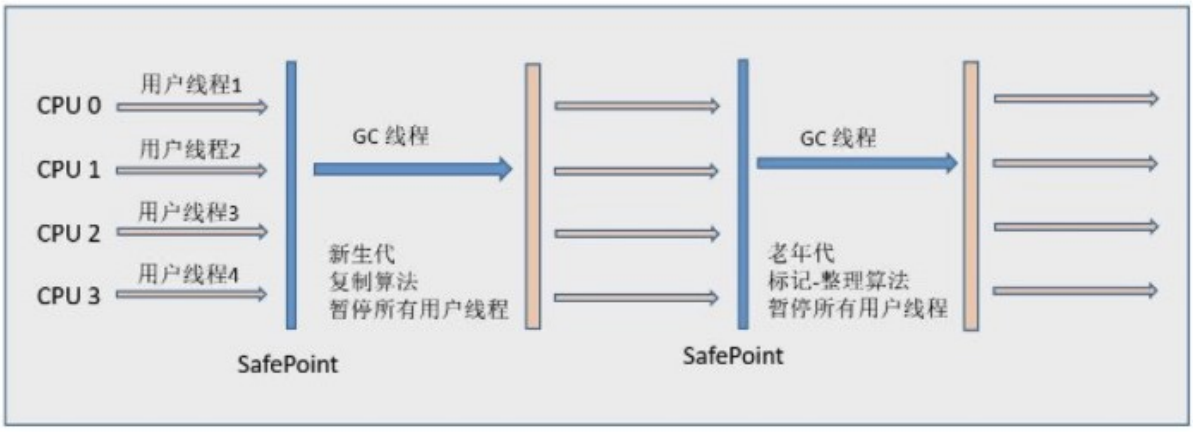
缩短停顿时间是以牺牲吞吐量和新生代空间来换取的：**新生代空间变小，垃圾回收变得频繁，导致吞吐量下降。**

特点：可以通过一个开关参数**XX:+UseAdaptiveSizePolicy**打开 **GC 自适应的调节策略（GC Ergonomics）**，**就不需要手工指定新生代的大小（-Xmn）**、Eden 和 Survivor 区的比例、晋升老年代对象年龄等细节参数了。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合适的停顿时间或者最大的吞吐量。

老年代收集器

4. Serial Old收集器

下图是Serial和SerialOld配合使用

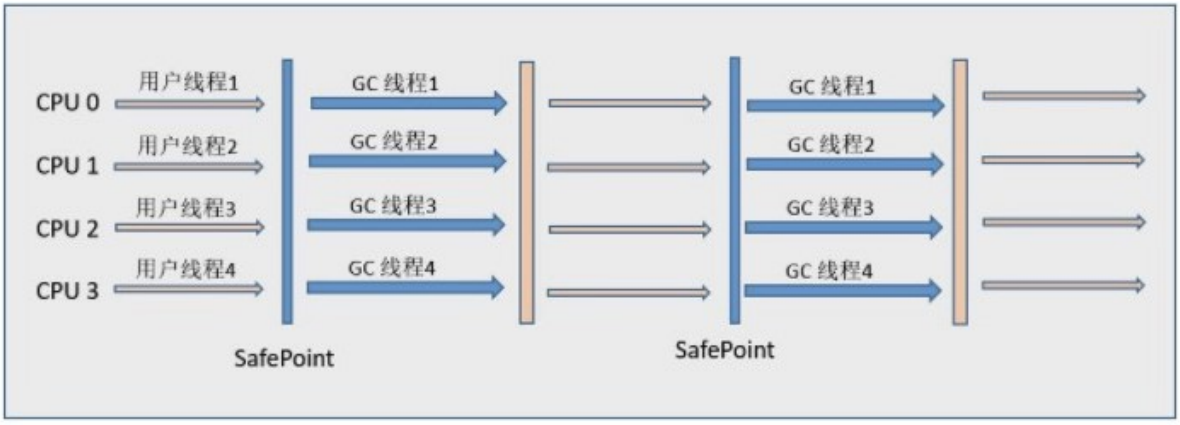


是 Serial 收集器的老年代版本，也是给 **Client 场景下的虚拟机使用。如果用在 Server 场景下，它有两大用途**：

- 在 JDK 1.5 以及之前版本（Parallel Old 诞生以前）中与 Parallel Scavenge 收集器搭配使用。
- 作为 CMS 收集器的**后备预案**，在并发收集发生 Concurrent Mode Failure 时使用。

5. Parallel Old收集器

Parallel Scavenge/Parallel Old收集器配合使用的流程图



是 Parallel Scavenge 收集器的老年代版本。

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。

6. CMS（Concurrent Mark Sweep）收集器

分为以下四个流程：

- **初始标记**：仅仅只是标记一下 GC Roots 能**直接关联**到的对象，**速度很快，需要停顿**。
- **并发标记**：进行 GC Roots Tracing 的过程，它在整个回收过程中**耗时最长，不需要停顿**。
- **重新标记**：为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，**需要停顿**。
- **并发清除**：**不需要停顿**。

在整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，不需要进行停顿。

优点：

CMS是一款优秀的收集器，它的主要**优点**在名字上已经体现出来了：**并发收集**、**低停顿**，因此CMS收集器也被称为**并发低停顿收集器（Concurrent Low Pause Collector）**。

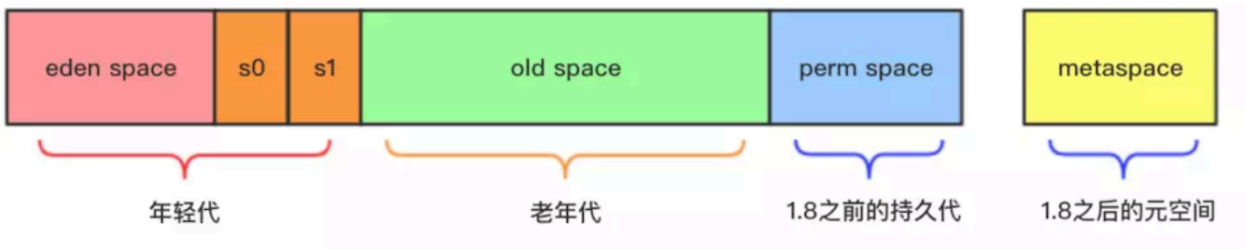
具有以下缺点：

- **吞吐量低**：低停顿时间是以牺牲吞吐量为代价的，导致 CPU 利用率不够高。
- **无法处理浮动垃圾**，可能出现 Concurrent Mode Failure。**浮动垃圾是指并发清除阶段由于用户线程继续运行而产生的垃圾**，这部分垃圾只能到下一次 GC 时才能进行回收。由于浮动垃圾的存在，因此需要预留出一部分内存，**意味着 CMS 收集不能像其它收集器那样等待老年代快满的时候再回收。如果预留的内存不够存放浮动垃圾，就会出现 Concurrent Mode Failure，这时虚拟机将临时启用 Serial Old 来替代 CMS**。
- **标记 - 清除算法导致的空间碎片（没有连续空间放大对象）**，往往出现**老年代空间剩余，但无法找到足够大连续空间来分配当前对象**，不得不提前触发一次 **Full GC**。

7. G1收集器（Garbage-First）

它是一款面向**服务端应用**的垃圾收集器，在**多 CPU 和大内存的场景**下有很好的性能。HotSpot 开发团队赋予它的使命是未来可以**替换掉 CMS 收集器**。

堆被分为新生代和老年代，其它收集器进行收集的范围都是整个新生代或者老年代，**而 G1 可以直接对新生代和老年代一起回收。**



G1 把堆划分成多个**大小相等的独立区域（Region）**，新生代和老年代不再物理隔离。

每个Region被标记了E、S、O和H，说明每个Region在运行时都充当了一种角色，**其中H是以往算法中没有的**，它代表Humongous，这表示这些Region存储的是**巨型对象（humongous object，H-obj）**，当新建对象大小**超过Region大小一半**时，直接在新的**一个或多个连续Region**中分配，并标记为H。

1. 横跨整个堆内存

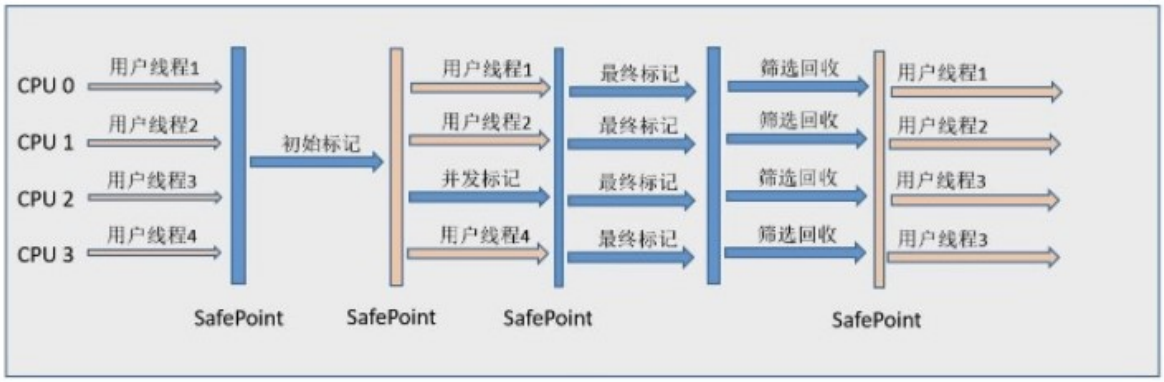
通过引入 Region 的概念，从而将原来的**一整块内存空间划分成多个的小空间**，使得**每个小空间可以单独进行垃圾回收。新生代和老年代不再是物理隔离的了，而都是一部分Region（不需要连续）的集合。**

2.建立可预测的时间模型（维护优先列表）

这种划分方法带来了很大的灵活性，使得**可预测的停顿时间模型**成为可能。**通过记录每个 Region 垃圾回收时间以及回收所获得的空间（这两个值是通过过去回收的经验获得），并维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region（这也就是Garbage-First名称的来由！！！）。**

3. 避免全堆扫描—— Remembered Set

每个 Region 都有一个Remembered Set，用来记录该 **Region 对象的引用对象所在的 Region**。通过使用 Remembered Set，在做**可达性分析**的时候就可以**避免全堆扫描**。



如果不计算维护 Remembered Set 的操作，G1 收集器的运作大致可划分为以下几个步骤：

- **初始标记**（同CMS，停顿）
- **并发标记**（同CMS，无需停顿）
- **最终标记**：为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，虚拟机将这段时间**对象变化记录在线程的 Remembered Set Logs** 里面，最终标记阶段需要把 Remembered Set Logs 的数据合并到 Remembered Set 中。**这阶段需要停顿线程，但是可并**

行执行。

- **筛选回收**：首先对各个 Region 中的**回收价值和成本进行排序**，根据**用户所期望的 GC 停顿时间**来制定回收计划。此阶段其实也可以做到与用户程序一起并发执行（**可以但是没必要**），但是因为只回收一部分 Region，时间是用户可控制的，**而且停顿用户线程将大幅度提高收集效率**。

具备如下特点：

- **空间整合**：整体来看是基于**“标记 - 整理”**算法实现的收集器，从局部（两个 Region 之间）上来看是基于**“复制”**算法实现的，这意味着运行期间**不会产生内存空间碎片**。
- **可预测的停顿**：能让使用者明确指定在一个长度为 M 毫秒的时间片段内，消耗在 GC 上的时间不得超过 N 毫秒。

7个收集器总结

收集器	串行、并行or并发	新生代/老年代	算法	目标	适用场景
Serial	串行	新生代	复制算法	响应速度优先	单CPU环境下的Client模式
Serial Old	串行	老年代	标记-整理	响应速度优先	单CPU环境下的Client模式、CMS的后备预案
ParNew	并行	新生代	复制算法	响应速度优先	**多CPU环境时在Server模式下与CMS配合**
Parallel Scavenge	并行	新生代	复制算法	吞吐量优先	在后台运算而不需要太多交互的任务
Parallel Old	并行	老年代	标记-整理	吞吐量优先	在后台运算而不需要太多交互的任务
CMS	并发	老年代	**标记-清除(Only one)**	响应速度优先	**集中在互联网站或B/S系统服务端上的Java应用**
G1	并发	both	标记-整理+复制算法	响应速度优先	**面向服务端应**用，将来替换CMS

jdk1.8 默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代）

jdk1.9 默认垃圾收集器G1

-XX:+PrintCommandLineFlagsjvm参数可查看默认设置收集器类型

-XX:+PrintGCDetails亦可通过打印的GC日志的新生代、老年代名称判断

三、内存分配与回收策略

Minor GC（新生代GC） 和 Full GC/Major GC （老年代GC）

- Minor GC/young GC：**回收新生代**，因为新生代对象存活时间很短，因此 Minor GC 会频繁执行，执行的速度一般也会比较快。
- Full GC/Major GC：**回收老年代**，出现了Full GC，**经常会伴随至少一次的Minor GC**（但非绝对的，在Parallel Scavenge收集器的收集策略里就有直接进行Full GC的策略选择过程）。Full GC的速度一般会比Minor GC慢10倍以上。
- **并发并行垃圾回收器在触发Full GC之前都会先触发一下Minor GC，这个可以根据参数进行配置。而串行垃圾回收的Full GC默认就是老年代回收。**

Full GC和7类GC收集器的区别

Full GC和Minor GC都指的是GC收集器回收所采用的模式，收集器才是具体实现

1. **Full GC == Major GC指的是对老年代/永久代的stop the world的GC**
2. **Full GC的次数 = 老年代GC时 stop the world的次数**
3. **Full GC的时间 = 老年代GC时 stop the world的总时间**
4. **CMS 不等于Full GC**，我们可以看到CMS分为多个阶段，**只有stop the world的阶段被计算到了Full GC的次数**和时间，而和业务线程并发的GC的次数和时间则不被认为是Full GC。
5. Full GC本身不会先进行Minor GC，我们可以配置，让Full GC之前先进行一次Minor GC，因为老年代很多对象都会引用到新生代的对象，先进行一次Minor GC可以提高老年代GC的速度。比如老年代使用CMS时，设置CMSScavengeBeforeRemark优化，让CMS remark之前先进行一次Minor GC。

内存分配策略

1. 对象优先在Eden分配

大多数情况下，对象在新生代 Eden 区分配，**当 Eden 区空间不够时，发起 Minor GC**。

2. 大对象直接进入老年代

大对象是指需要**连续内存空间的对象**，最典型的大对象是那种**很长的字符串以及数组**。

经常出现大对象会提前触发垃圾收集以获取足够的连续空间分配给大对象。

-XX:PretenureSizeThreshold，大于此值的对象直接在老年代分配，避免在 Eden 区和 Survivor 区之间的大量内存复制。

3. 长期存活的对象进入老年代

为对象定义年龄计数器，对象在 Eden 出生并**经过 Minor GC 依然存活，将移动到 Survivor 中，年龄就增加 1 岁**，增加到一定年龄则移动到老年代中。

-XX:MaxTenuringThreshold 用来定义年龄的阈值。

4. 动态对象年龄判断

虚拟机并不是永远地要求对象的年龄必须达到 MaxTenuringThreshold 才能晋升老年代，如果在 **Survivor 中相同年龄所有对象大小的总和大于 Survivor 空间的一半，则年龄大于或等于该年龄的对象可以直接进入老年代**，无需等到 MaxTenuringThreshold 中要求的年龄。

5. 空间分配担保

在发生 **Minor GC** 之前，**虚拟机先检查老年代最大可用的「连续空间」是否大于新生代「所有对象」总空间**，如果条件成立的话，那么 Minor GC 可以确认是安全的。

如果不成立的话虚拟机会查看 HandlePromotionFailure 设置值**是否允许担保失败**，如果允许那么就会继续检查老年代**最大可用的「连续空间」是否大于历次晋升到老年代对象的平均大小**，如果大于，将尝试着进行一次 Minor GC；如果小于，或者 HandlePromotionFailure 设置不允许冒险，**那么就要进行一次 Full GC**。

Full GC的触发条件

对于 Minor GC，其触发条件非常简单，当 Eden 空间满时，就将触发一次 Minor GC。而 Full GC 则相对复杂，有以下条件：

1. 调用System.gc()

只是建议虚拟机执行 Full GC，但是虚拟机不一定真正去执行。**不建议使用这种方式**，而是让虚拟机管理内存。

2. 老年代空间不足

老年代空间不足的常见场景为前文所讲的大对象直接进入老年代、长期存活的对象进入老年代等。

为了避免以上原因引起的 Full GC，应当尽量不要创建过大的对象以及数组。除此之外，可以**通过 -Xmn 虚拟机参数调大新生代的大小**，让对象尽量在新生代被回收掉，不进入老年代。还可以通过 **-XX:MaxTenuringThreshold 调大对象进入老年代的年龄**，让对象在新生代多存活一段时间。

3. 空间分配担保失败

使用复制算法的 Minor GC 需要老年代的内存空间作担保，如果担保失败会执行一次 Full GC。具体内容请参考上面的第五小节。

4. JDK 1.7之前的永久代空间不足

在 JDK 1.7 及以前，HotSpot 虚拟机中的**方法区是用永久代实现的**，永久代中存放的为一些 Class 的信息、常量、静态变量等数据。

当系统中要**加载的类、反射的类和调用的方法较多时**，永久代可能会被占满，在未配置为采用 CMS GC 的情况下也会执行 Full GC。如果经过 Full GC 仍然回收不了，**那么虚拟机会抛出 java.lang.OutOfMemoryError。**

为避免以上原因引起的 Full GC，可采用的方法为**增大永久代空间或转为使用 CMS GC**。

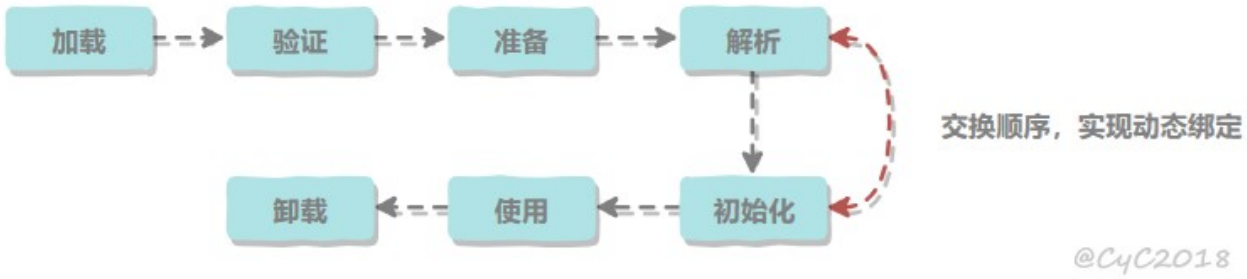
5. Concurrent Mode Failure

执行 CMS GC 的过程中同时有对象要放入老年代，而此时老年代空间不足（可能是 **GC 过程中浮动垃圾过多导致**暂时性的空间不足），便会报 Concurrent Mode Failure 错误，并触发 **Full GC**。

四、 类加载机制

类是在运行期间第一次使用时动态加载的，而不是一次性加载。因为如果一次性加载，那么会占用很多的内存。

类的生命周期



包括以下 7 个阶段：

- **加载（Loading）**
- **验证（Verification）**
- **准备（Preparation）**
- **解析（Resolution）**
- **初始化（Initialization）**
- 使用（Using）
- 卸载（Unloading）

类加载过程

包含了加载、验证、准备、解析和初始化这 5 个阶段。

1. 加载

加载是类加载的一个阶段，注意不要混淆。

加载过程完成以下三件事：

- 通过类的**完全限定名称**获取定义该类的**二进制字节流。**
- 将该字节流表示的**静态存储结构转换为方法区的运行时存储结构。**
- 在内存中生成一个**代表该类的 Class 对象**，作为方法区中该类各种数据的访问入口。

其中二进制字节流可以从以下方式中获取：

- **从 ZIP 包读取**，成为 JAR、EAR、WAR 格式的基础。
- **从网络中获取**，最典型的应用是 Applet。
- **运行时计算生成**，例如动态代理技术，在 java.lang.reflect.Proxy 使用 ProxyGenerator.generateProxyClass 的代理类的二进制字节流。
- **由其他文件生成**，例如由 JSP 文件生成对应的 Class 类。

2. 验证

确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

3. 准备

类变量是被 static 修饰的变量，准备阶段**为类变量分配内存并设置初始值**，使用的是**方法区**的内存。

实例变量不会在这阶段分配内存，它会在对象实例化时随着对象一起被分配在堆中。应该注意到，**实例化不是类加载的一个过程**，类加载发生在所有实例化操作之前，并且类加载只进行一次，实例化可以进行多次。

初始值一般为 0 值，例如下面的类变量 value 被初始化为 0 而不是 123。

```
public static int value = 123;
```

如果**类变量是常量**，那么它将初始化为表达式所定义的值而不是 0。例如下面的常量 value 被初始化为 123 而不是 0。

```
public static final int value = 123;
```

4. 解析

将**常量池的符号引用替换为直接引用**的过程。

其中解析过程**在某些情况下可以在初始化阶段之后再开始**，这是为了支持 **Java 的动态绑定**。

5. 初始化

初始化阶段才真正开始执行类中定义的 Java 程序代码。初始化阶段是**虚拟机执行类构造器 <clinit>() 方法**的过程。在**准备阶段，类变量已经赋过一次系统要求的初始值，而在初始化阶段，根据程序员通过程序制定的主观计划去初始化类变量和其它资源**。

<clinit>() 是**由编译器自动收集类中所有类变量的赋值动作和静态语句块中的语句合并产生的**，编译器收集的顺序由语句在**源文件中出现的顺序决定**。特别注意的是，**静态语句块只能访问到定义在它之前的类变量**，定义在它之后的类变量**只能赋值，不能访问**。例如以下代码：

```
public class Test {
    static {
        i = 0;                // 给变量赋值可以正常编译通过
        System.out.print(i);  // 这句编译器会提示“非法向前引用”
    }
    static int i = 1;
}
```

由于父类的 <clinit>() 方法先执行，也就意味着**父类中定义的静态语句块的执行要优先于子类**。例如以下代码：

```
static class Parent {
    public static int A = 1;
    static {
        A = 2;
    }
}

static class Sub extends Parent {
    public static int B = A;
```

```
}

public static void main(String[] args) {
     System.out.println(Sub.B);  // 2
}
```

接口中**不可以使用静态语句块**，但**仍然有类变量初始化的赋值操作**，因此接口与类一样都会生成 <clinit>() 方法。**但接口与类不同的是**，执行接口的 <clinit>() 方法**不需要先执行父接口的** <clinit>() 方法。只有当父接口中定义的变量使用时，父接口才会初始化。另外，**接口的实现类在初始化时也一样不会执行接口的** <clinit>() 方法。

虚拟机会保证一个类的 <clinit>() 方法在多线程环境下被正确的加锁和同步，如果多个线程同时初始化一个类，只会有一个线程执行这个类的 <clinit>() 方法，其它线程都会阻塞等待，直到活动线程执行 <clinit>() 方法完毕。如果在一个类的 <clinit>() 方法中有耗时的操作，就可能造成多个线程阻塞，在实际过程中此种阻塞很隐蔽。

类初始化时机

1. 主动引用

虚拟机规范中并没有强制约束何时进行加载，但是规范严格规定了有且**只有下列五种情况必须对类进行初始化（加载、验证、准备、解析**都会随之发生）：

- 遇到 new、getstatic、putstatic、invokestatic 这四条字节码指令时，如果类没有进行过初始化，则必须先触发其初始化。最常见的生成这 4 条指令的场景是：**1. 使用 new 关键字实例化对象的时候；2. 读取或设置一个类的静态字段（被 final 修饰、已在编译期把结果放入常量池的静态字段除外）的时候；3. 以及调用一个类的静态方法的时候。**
- 使用 java.lang.reflect 包的方法**对类进行反射调用的时候**，如果类没有进行初始化，则需要先触发其初始化。
- 当初始化一个类的时候，如果发现其**父类还没有进行过初始化，则需要先触发其父类的初始化。**
- 当**虚拟机启动时**，用户需要指定一个**要执行的主类（包含 main() 方法的那个类），虚拟机会先初始化这个主类**；
- 当使用 JDK 1.7 的动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果为 REF_getStatic, REF_putStatic, REF_invokeStatic 的**方法句柄**，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化；

2. 被动引用

以上 5 种场景中的行为称为对一个类进行主动引用。除此之外，所有引用类的方式都不会触发初始化，称为被动引用。被动引用的常见例子包括：

- **通过子类引用父类的静态字段**，不会导致子类初始化。

```
System.out.println(SubClass.value);  // value 字段在 SuperClass 中定义
```

- **通过数组定义来引用类**，不会触发此类的初始化。该过程会对数组类进行初始化，数组类是一个由虚拟机自动生成的、直接继承自 Object 的子类，其中包含了数组的属性和方法。

```
SuperClass[] sca = new SuperClass[10];
```

- **常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类**，因此不会触发定义常量的类的初始化。

```
System.out.println(ConstClass.HELLOWORLD);
```

类与类加载器

两个类相等，需要类本身相等，并且使用同一个类加载器进行加载。这是因为每一个类加载器都拥有一个独立的类名称空间。

这里的相等，包括类的 Class 对象的 equals() 方法、isAssignableFrom() 方法、isInstance() 方法的返回结果为 true，也包括使用 instanceof 关键字做对象所属关系判定结果为 true。

类加载器分类

从 Java 虚拟机的角度来讲，只存在以下两种不同的类加载器：

- **启动类加载器（Bootstrap ClassLoader）**，使用 C++ 实现，是虚拟机自身的一部分；
- 所有其它类的加载器，使用 Java 实现，独立于虚拟机，**继承自抽象类 java.lang.ClassLoader**。

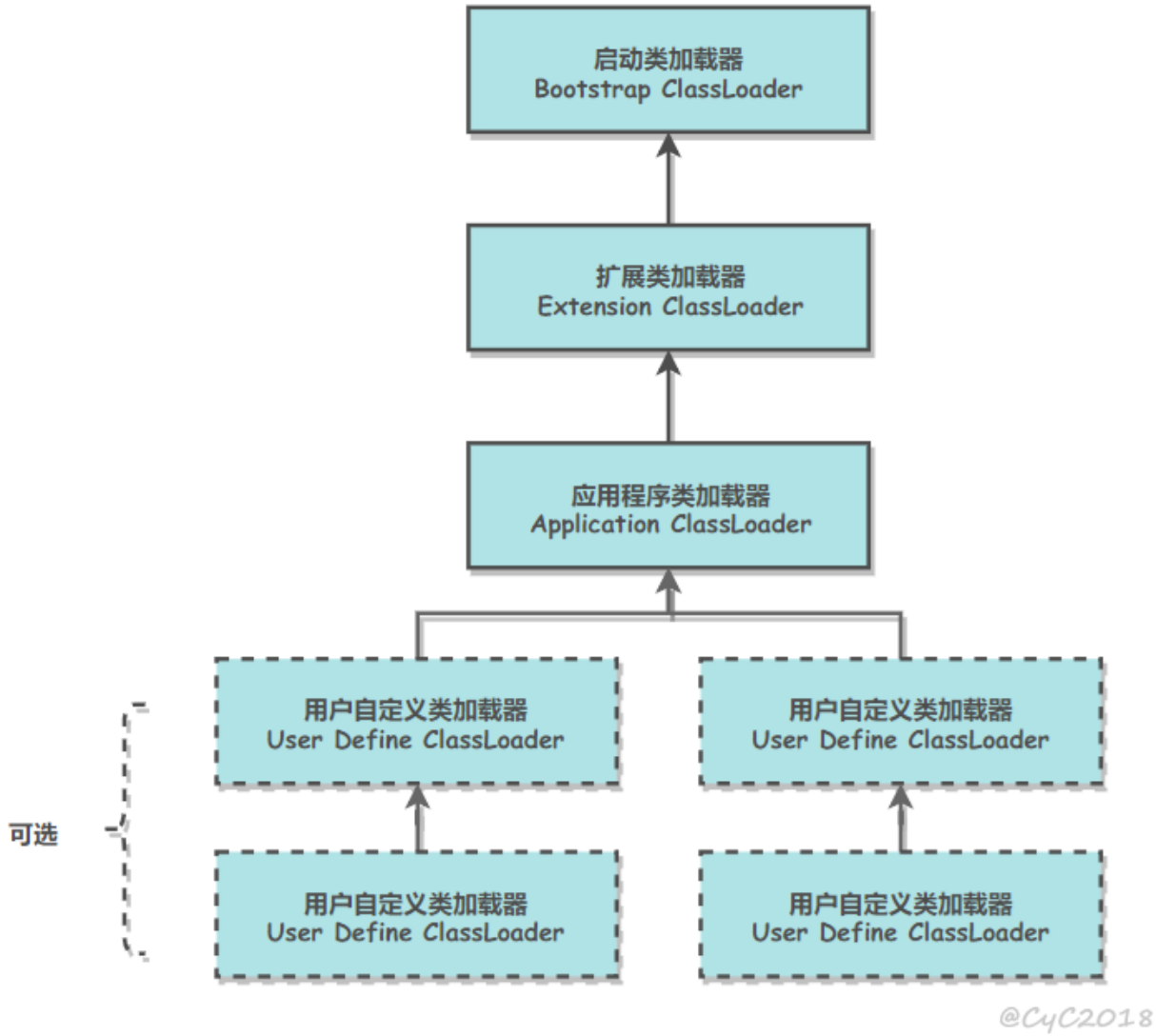
从 Java 开发人员的角度看，类加载器可以划分得更细致一些：

- **启动类加载器（Bootstrap ClassLoader）**此类加载器负责将存放在 **<JRE_HOME>\lib** 目录中的，或者被 **-Xbootclasspath 参数**所指定的路径中的，并且是**虚拟机识别的（仅按照文件名识别**，如 rt.jar，名字不符合的类库即使放在 lib 目录中也不会被加载）类库加载到虚拟机内存中。启动类加载器**无法被 Java 程序直接引用**，用户在编写自定义类加载器时，**如果需要把加载请求委派给启动类加载器，直接使用 null 代替即可**。
- **扩展类加载器（Extension ClassLoader）**这个类加载器是由 **ExtClassLoader**（sun.misc.Launcher$ExtClassLoader）实现的。它负责将 **<JAVA_HOME>/lib/ext 或者被 java.ext.dir 系统变量所指定路径**中的所有类库加载到内存中，**开发者可以直接使用扩展类加载器**。
- **应用程序类加载器（Application ClassLoader）**这个类加载器是由 **AppClassLoader**（sun.misc.Launcher$AppClassLoader）实现的。由于这个类加载器是 ClassLoader 中的 **getSystemClassLoader()** 方法的返回值，**因此一般称为系统类加载器。它负责加载用户类路径（ClassPath）上所指定的类库**，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，**一般情况下这个就是程序中默认的类加载器**。

双亲委派模型

应用程序是由三种类加载器互相配合从而实现类加载，除此之外还可以加入自己定义的类加载器。

下图展示了类加载器之间的层次关系，称为**双亲委派模型（Parents Delegation Model）**。该模型要求除了顶层的启动类加载器外，**其它的类加载器都要有自己的父类加载器**。类加载器之间的父子关系**一般通过组合关系（Composition）来实现，而不是继承关系（Inheritance）**



1. 工作过程

一个类加载器首先将类加载请求转发到父类加载器，只有当父类加载器无法完成时才尝试自己加载。

2. 好处

使得 Java 类随着它的类加载器一起具有一种带有优先级的层次关系，从而使得基础类得到统一。

例如 java.lang.Object 存放在 **rt.jar** 中，如果编写另外一个 java.lang.Object 并放到 ClassPath 中，程序可以编译通过。由于双亲委派模型的存在，所以在 rt.jar 中的 Object 比在 ClassPath 中的 Object 优先级更高，这是因为 rt.jar 中的 Object 使用的是**启动类加载器**，而 ClassPath 中的 Object 使用的是**应用程序类加载器**。rt.jar 中的 Object 优先级更高，那么程序中所有的 Object 都是这个 Object。

3. 实现

以下是抽象类 java.lang.ClassLoader 的代码片段，其中的 loadClass() 方法运行过程如下：**先检查类是否已经加载过，如果没有则让父类加载器去加载**。当父类加载器加载失败时抛出 ClassNotFoundException，此时尝试自己去加载。

```
public abstract class ClassLoader {
    // The parent class loader for delegation
```

```
    private final ClassLoader parent;

    public Class<?> loadClass(String name) throws ClassNotFoundException {
        return loadClass(name, false);
    }

    protected Class<?> loadClass(String name, boolean resolve) throws
ClassNotFoundException {
        synchronized (getClassLoadingLock(name)) {
            // First, check if the class has already been loaded
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                try {
                    if (parent != null) {
                        c = parent.loadClass(name, false);
                    } else {
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    // ClassNotFoundException thrown if class not found
                    // from the non-null parent class loader
                }

                if (c == null) {
                    // If still not found, then invoke findClass in order
                    // to find the class.
                    c = findClass(name);
                }
            }
            if (resolve) {
                resolveClass(c);
            }
            return c;
        }
    }

    protected Class<?> findClass(String name) throws ClassNotFoundException {
        throw new ClassNotFoundException(name);
    }
}
```

自定义类加载器实现

FileSystemClassLoader 是自定义类加载器，继承自 java.lang.ClassLoader，用于加载文件系统上的类。它首先根据类的全名在文件系统上查找类的字节代码文件（.class 文件），然后读取该文件内容，最后通过 **defineClass() 方法（ClassLoader的方法）**来把这些**字节代码转换成 java.lang.Class 类的实例**。

java.lang.ClassLoader 的 loadClass() 实现了双亲委派模型的逻辑，**自定义类加载器一般不去重写它，但是需要重写 findClass() 方法。**

```
public class FileSystemClassLoader extends ClassLoader {

    private String rootDir;

    public FileSystemClassLoader(String rootDir) {
        this.rootDir = rootDir;
    }

    protected Class<?> findClass(String name) throws ClassNotFoundException {
        byte[] classData = getClassData(name);
        if (classData == null) {
            throw new ClassNotFoundException();
        } else {
            return defineClass(name, classData, 0, classData.length);
        }
    }

    private byte[] getClassData(String className) {
        String path = classNameToPath(className);
        try {
            InputStream ins = new FileInputStream(path);
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            int bufferSize = 4096;
            byte[] buffer = new byte[bufferSize];
            int bytesNumRead;
            while ((bytesNumRead = ins.read(buffer)) != -1) {
                baos.write(buffer, 0, bytesNumRead);
            }
            return baos.toByteArray();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }

    private String classNameToPath(String className) {
        return rootDir + File.separatorChar
                + className.replace('.', File.separatorChar) + ".class";
    }
}
```

破坏双亲委派

为什么要破坏双亲委派?

因为在某些情况下**父类加载器需要委托子类加载器去加载class文件**。受到加载范围的限制，父类加载器无法加载到需要的文件。

以Driver接口为例，由于Driver接口定义在jdk当中的，**而其实现由各个数据库的服务商来提供**，比如MySQL的就写了`MySQL Connector`，那么问题就来了，DriverManager（也由jdk提供）要加载各个实现了Driver接口的实现类，然后进行管理，**但是DriverManager由启动类加载器加载**，只能记载JAVA_HOME的lib下文件，**而其实现是由服务商提供的，由系统类加载器加载**，这个时候就需要**启动类加载器来委托子类来加载Driver实现**，从而破坏了双亲委派，这里仅仅是举了破坏双亲委派的其中一个情况。