一、内存区域

1、jvm运行时数据区主要分为两个部分，一部分是线程共享的，有堆、方法区（常量池（运行时、字符串）），另一部分是线程独有的，比如程序计数器、栈，其中栈又分为两种：虚拟机栈、本地方法栈

- 堆：jvm管理的内存最大的一块，用于存放对象实例，比如说通过new的方式生成的对象
- 方法区：方法区是一个逻辑上的概念，具体上的实现是遵循了jvm规范的，比如说jdk1.7的永久代（位于jvm管理内存）、jdk1.7之后的元空间（位于操作系统管理的），用于存储被虚拟机加载的**类信息、字段信息、方法信息、常量、静态变量、即时编译器编译后的代码缓存等数据。**
    - 运行时常量池：用于存放编译期生成的各种字面量和符号引用的 **常量池表(Constant Pool Table)**，一个表存了很多符号。
    - **字符串常量池**：由于字符串是最容易使用的，所以 JVM 为了提升性能和减少内存消耗针对字符串（String 类）专门开辟的一块区域，主要目的是为了避免字符串的重复创建。
        - JDK1.7 之前，字符串常量池存放在永久代
        - JDK1.7 字符串常量池和静态变量从永久代移动了 Java 堆中。
- 程序计数器：使用的是程序计数器，是一块较小的内存空间，
    1. 主要就是存放当前线程执行指令，下一条要执行的指令，从而实现代码的流程控制
    2. 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪了
- 虚拟机栈：栈也是一片内存空间，**为虚拟机执行 Java 方法 （也就是字节码）服务**，栈是由一个个栈帧组成，每调用一个方法就会创建一个栈帧，而每个栈帧中都拥有：局部变量表、操作数栈、动态链接、方法返回地址。
- 本地方法栈：**为虚拟机使用到的 Native 方法服务**，本地方法被执行的时候，在本地方法栈也会创建一个栈帧，用于存放该本地方法的局部变量表、操作数栈、动态链接、出口信息。

2、**内存分配的两种方式** （补充内容，需要掌握）：

- 指针碰撞：
    - 适用场合：堆内存规整（即没有内存碎片）的情况下。
    - 原理：用过的内存全部整合到一边，没有用过的内存放在另一边，中间有一个分界指针，只需要向着没用过的内存方向将该指针移动对象内存大小位置即可。
    - 使用该分配方式的 GC 收集器：Serial, ParNew
- 空闲列表：
    - 适用场合：堆内存不规整的情况下。
    - 原理：虚拟机会维护一个列表，该列表中会记录哪些内存块是可用的，在分配的时候，找一块儿足够大的内存块儿来划分给对象实例，最后更新列表记录。
    - 使用该分配方式的 GC 收集器：CMS

3、**内存分配并发问题（补充内容，需要掌握）**

在创建对象的时候有一个很重要的问题，就是线程安全，因为在实际开发过程中，创建对象是很频繁的事情，作为虚拟机来说，必须要保证线程是安全的，通常来讲，虚拟机采用两种方式来保证线程安全：

- **CAS+失败重试：** CAS 是乐观锁的一种实现方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。**虚拟机采用 CAS 配上失败重试的方式保证更新操作的原子性。**

- **TLAB：** 为每一个线程预先在 Eden 区分配一块儿内存，JVM 在给线程中的对象分配内存时，首先在 TLAB 分配，当对象大于 TLAB 中的剩余内存或 TLAB 的内存已用尽时，再采用上述的 CAS 进行内存分配

4、对象的访问定位有两种方式：句柄和直接指针

- 句柄：Java堆被划分为两部分：一部分是存放真正的对象数据的Java堆，另一部分是一个称为句柄池（Handle Pool）的内存区域，对象的引用由句柄组成，每个句柄包含两个指针：一个指向句柄池中的句柄（栈指向句柄），另一个指向真正的对象数据（句柄指向真正的内存地址）。
    - 当Java线程需要访问对象时，首先通过栈内的对象引用，找到句柄池，句柄池找到对应的句柄，然后再由句柄中的指针定位到对象数据。
    - 这样做的好处是对象的引用地址是稳定的，因为真正的对象数据可以被移动，但是句柄池中的句柄不会变化。
    - 句柄方式的缺点是需要额外的一次指针定位，增加了访问对象的开销，对性能有一定的影响。
- 直接指针：栈内的引用存放的就是对象真实的内存地址，访问对象时访问引用，直接定位到对象
    - 使用直接指针访问方式最大的好处就是速度快，它节省了一次指针定位的时间开销

这两种对象访问方式各有优势。使用句柄来访问的最大好处是 reference 中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而 reference 本身不需要修改。使用直接指针访问方式最大的好处就是速度快，它节省了一次指针定位的时间开销

二、垃圾回收

1、可以通过引用计数法和可达性算法进行判断

- 引用计数法：给对象中添加一个引用计数器，每当有一个地方引用它，计数器就加 1，当引用失效，计数器就减 1，任何时候计数器为 0 的对象就是不可能再被使用的，当进行垃圾回收时，为0的就会被清除。虽然**这个方法实现简单，效率高，但是目前主流的虚拟机中并没有选择这个算法来管理内存，其最主要的原因是它很难解决对象之间循环引用的问题，存在内存泄漏的问题**
- 可达性算法：通过一系列的称为 **“GC Roots”** 的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连的话，则证明此对象是不可用的，需要被回收。
    - 虚拟机栈、本地方法栈中引用的对象
    - 方法区中常量引用的对象、类静态属性引用的对象
    - 被同步锁持有的对象

2、前面谈到了无论是通过引用计数法判断对象引用数量，还是通过可达性分析法判断对象的引用链是否可达，判定对象的存活都与“引用”有关。

1．强引用（StrongReference）- 不可或缺

以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。比如我们通过=赋值的对象，强引用对象即使当内存空间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

2．软引用（SoftReference）- 可有可无

我们可以通过创建SoftReference对象，当进行垃圾回收时，如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。

- 只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。
- 软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，JAVA 虚拟机就会把这个软引用加入到与之关联的引用队列中。

好处：**软引用可以加速 JVM 对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出（OutOfMemory）等问题的产生。**

3．弱引用（WeakReference）- 可有可无

我们可以通过创建WeakReference对象，和软引用非常相似。

- 弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

4．虚引用（PhantomReference） - 形同虚设

我们可以通过创建PhantomReference对象，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

- **虚引用主要用来跟踪对象被垃圾回收的活动。**
- **虚引用与软引用和弱引用的一个区别在于：** 虚引用必须和引用队列（ReferenceQueue）联合使用。
    - 当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

3、假如在字符串常量池中存在字符串 "abc"，如果当前没有任何 String 对象引用该字符串常量的话，就说明常量 "abc" 就是废弃常量，如果这时发生内存回收的话而且有必要的话，"abc" 就会被系统清理出常量池了。

4、判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足下面 3 个条件才能算是 **“无用的类”**：

- 该类所有的实例都已经被回收，也就是 Java 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 `java.lang.Class` 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

虚拟机可以对满足上述 3 个条件的无用类进行回收，这里说的仅仅是“可以”，而并不是和对象一样不使用了就会必然被回收。

5，6、标记-清除、分区复制、标记-整理、分代收集

【标记-清除算法】：分为“标记”和“清除”两个阶段，标记阶段遍历整个堆标记出所有不需要回收的对象，在标记完成后统一回收掉所有没有被标记的对象。

- 两个阶段过程效率都不高，而且标记清除后会产生大量不连续的内存碎片

【分区复制算法】：将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。

- 解决了内存碎片的问题
- 但是可用内存缩小为原来的一半，而且如果存活对象数量比较大，复制性能会变得很差。

【标记-整理算法】：标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。

【分代收集算法】：根据对象存活周期的不同将内存分为几块。一般将 Java 堆分为新生代和老年代、永久代（元空间），这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

- 新生代中，每次收集都会有大量对象死去，所以可以选择”标记-复制“算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。
- 老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。

7、假设我们使用的是 HotSpot 虚拟机，并且采用了分代垃圾回收策略，其中新生代（Young Generation）使用复制算法，老年代（Old Generation）使用标记整理算法。

1. **初始状态**：
    - 新生代中的 Eden区、Survivor区From和Survivor区To都是空的，老年代中也是空的。
    - 永久代（或元空间）中存放着类的元数据信息和其他静态数据。
2. **创建对象**：
    - 当我们在应用程序中创建新的对象时，这些对象首先被分配到新生代的 Eden区。
3. **新生代的垃圾回收（Minor GC）**：
    - 当 Eden区满时，触发新生代的垃圾回收。
    - 垃圾回收器会对Eden区、Survivor区From和Survivor区To进行垃圾回收，首先，它会从根对象（如全局变量、活动栈、静态变量等）出发，遍历所有的可达对象，并将它们标记为活动对象。
    - 然后，所有活动对象将被复制到Survivor区To，并且为每个对象的年龄加1。年龄加1后，如果其中有对象的年龄达到一定阈值（通常为15岁）或者空间达到一定阈值时，则它们会被晋升到老年代。
    - 清空Eden区和Survivor区From。
4. **停顿（Stop-the-World）现象**：
    - 在执行新生代的垃圾回收（Minor GC）时，会发生停顿现象。这意味着所有的应用程序线程都会被暂停，垃圾回收线程会独占CPU资源进行垃圾回收操作。
    - 停顿时间的长短取决于垃圾回收器的实现和配置，通常在新生代垃圾回收时停顿时间比较短，但在老年代的垃圾回收（Major GC或Full GC）时，停顿时间可能会更长。
5. **老年代的垃圾回收（Major GC或Full GC）**：
    - 当老年代中的空间满了，或者由于某些其他原因触发老年代的垃圾回收。
    - 垃圾回收器会对老年代进行垃圾回收，首先，它会从根对象出发，遍历所有的可达对象，并将它们标记为活动对象。
    - 然后，对老年代中的对象进行标记整理，将未被标记的垃圾对象进行回收，然后将存活对象紧凑排列，消除内存碎片。

- 对象晋升：从新生代晋升到老年代的对象和在老年代中存活的对象都会经过老年代的垃圾回收。

6. **永久代（或元空间）的垃圾回收**：
   - 永久代（或元空间）一般用于存放静态数据和类的元数据信息，它的垃圾回收通常在应用程序结束时或类加载器被回收时进行。
   - 在 Java 8 之前，永久代是有限的，因此如果存在过多的类加载和卸载，可能导致永久代内存溢出。
   - 在 Java 8 及以后的版本，永久代被元空间取代，它的内存由操作系统管理，不再有固定大小的限制。

8、常见的垃圾回收器有以下几种，一种是串行化的垃圾回收，一种是以吞吐量为主的垃圾回收，另一种是以响应时间为主的垃圾收集器：

串行化：

1. **Serial收集器**（Serial Garbage Collector）：
   - 为单线程环境设计的垃圾回收器，使用**标记-复制算法**进行新生代垃圾回收。
   - 在新生代使用Serial收集器，老年代使用Serial Old收集器的组合称为Serial+Serial Old收集器组合。

吞吐量：

1. **ParNew收集器**（Parallel New Garbage Collector）：
   - 是Serial收集器的多线程版本，也使用**标记-复制算法**进行新生代垃圾回收。
   - 在新生代使用ParNew收集器，老年代使用Parallel Old收集器的组合称为ParNew+Parallel Old收集器组合。
2. **Parallel Scavenge 收集器**：
   - 是一种以高吞吐量为目标的垃圾回收器，使用**标记-复制算法**进行新生代垃圾回收。
   - 它使用多线程进行垃圾回收，旨在充分利用多核处理器的优势，以达到高吞吐量。
   - 主要适用于后台任务和数据处理等对吞吐量要求较高的应用场景。
   - 在新生代使用Parallel Scavenge收集器，老年代使用Parallel Old收集器的组合称为Parallel Scavenge+Parallel Old收集器组合。

响应时间：

1. **CMS收集器**（Concurrent Mark-Sweep Garbage Collector）：
   - 是以获取最短回收停顿时间为目标的收集器，使用**标记-清除算法**进行老年代垃圾回收。
   - 并发标记阶段和用户线程一起运行，减少了停顿时间，但会增加垃圾回收过程的CPU开销。
   - 在新生代使用ParNew收集器，老年代使用CMS收集器的组合称为ParNew+CMS收集器组合。

吞吐量和响应时间：

1. **G1收集器**（Garbage-First Garbage Collector）：
   - 是一种面向服务器端应用的垃圾回收器，将堆内存划分为多个区域，并使用不同的算法进行垃圾回收。
   - G1收集器通过优先处理垃圾最多的区域来达到回收最多垃圾的目的，从而提供更稳定的吞吐量和响应时间。
2. **ZGC收集器**（Z Garbage Collector）：
   - 是 JDK 11 中引入的一种低延迟垃圾回收器，专注于减少垃圾回收带来的停顿时间。
   - 它适用于大堆、多核服务器上的应用程序，可以在数毫秒的范围内实现较低的停顿时间。

- ZGC收集器使用**标记-整理算法**来进行垃圾回收。
- 在新生代使用标记-复制算法，老年代使用标记-整理算法。

根据垃圾回收算法的不同，常见的垃圾收集器可以分为以下几种：

1. **复制（Copying）算法**：
   - 垃圾收集器：Serial收集器、ParNew收集器
   - 特点：将内存分为两块，每次只使用其中一块，当一块内存用完时，将存活对象复制到另一块内存中，然后清空之前的内存。适用于新生代，由于大部分对象都是朝生夕灭的，因此适合采用复制算法。
2. **标记-清除（Mark-Sweep）算法**：
   - 垃圾收集器：CMS收集器、G1收集器
   - 特点：分为标记和清除两个阶段。首先标记出所有存活对象，然后清除未标记的垃圾对象。由于标记和清除过程的效率较低，可能产生内存碎片，因此在老年代中使用较多。
3. **标记-整理（Mark-Compact）算法**：
   - 垃圾收集器：Parallel Old收集器
   - 特点：标记出所有存活对象，然后将存活对象向一端移动，清理掉边界以外的垃圾对象。适用于老年代，由于老年代中存活对象较多，采用复制算法会浪费大量内存。
4. **分代收集算法**：
   - 垃圾收集器：Parallel Scavenge收集器、G1收集器
   - 特点：将堆内存分为不同的代，通常分为新生代和老年代，根据不同代的特点采用不同的垃圾回收算法，以提高垃圾回收效率。新生代使用复制算法，老年代使用标记-整理或标记-清除算法。

不同的垃圾收集器使用不同的算法，针对不同的应用场景和内存特点，选择合适的垃圾收集器可以优化应用程序的性能和稳定性。

好的，请看下面输出的内容：

根据不同的区域（新生代、老年代、永久代/元空间），常见的垃圾收集器可以分为以下几种：

新生代（Young Generation）垃圾收集器：

1. **Serial收集器**：
   - 单线程收集器，使用**复制算法**，适用于单核处理器或较小的应用程序。
   - 参数：`-XX:+UseSerialGC`
2. **ParNew收集器**：
   - Serial收集器的多线程版本，使用**复制算法**，适用于多核处理器。
   - 参数：`-XX:+UseParNewGC`
3. **Parallel Scavenge收集器**：
   - 使用**复制算法**，追求高吞吐量，适用于后台任务和数据处理等对吞吐量要求较高的应用。
   - 参数：`-XX:+UseParallelGC`

老年代（Old Generation）垃圾收集器：

1. **Serial Old收集器**：
   - 单线程收集器，使用**标记-整理算法**，适用于单核处理器或吞吐量要求不高的应用。
   - 参数：`-XX:+UseSerialOldGC`
2. **Parallel Old收集器**：
   - 多线程收集器，使用**标记-整理算法**，适用于多核处理器和吞吐量要求较高的应用。

    - 参数：`-XX:+UseParallelOldGC`
3. **CMS收集器**：
    - 以获取最短回收停顿时间为目标的收集器，使用**标记-清除算法**，适用于对响应时间要求较高的应用。
    - 参数：`-XX:+UseConcMarkSweepGC`

永久代/元空间（Permanent Generation/Metaspace）垃圾收集器：

1. **Serial收集器**：
    - Serial收集器用于新生代，新生代中的字符串常量池和静态变量存放在永久代/元空间中。
    - 参数：`-XX:+UseSerialGC`
2. **G1收集器**：
    - G1收集器用于堆内存划分为多个区域，其中包含了永久代/元空间的概念，用于存储类的元数据信息。
    - 参数：`-XX:+UseG1GC`

不同区域的垃圾收集器根据其特点和适用场景，可以灵活地选择和配置，以满足应用程序的性能和稳定性需求。

并行和并发概念补充：

- **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，可能会交替执行），用户程序在继续运行，而垃圾收集器运行在另一个 CPU 上。

9、好的，简洁介绍 CMS 和 G1 收集器：

CMS（Concurrent Mark-Sweep）收集器：

- 目标是降低垃圾回收停顿时间（响应时间），适用于高并发应用。
- 使用标记-清除算法，在并发标记和并发清除阶段与用户线程并发执行，减少停顿时间。

G1（Garbage-First）收集器：

- 目标是降低停顿时间和高吞吐量的平衡，适用于大堆内存应用。
- 将堆内存划分为多个区域，优先回收垃圾最多的区域，降低停顿时间。

总体来说，CMS 和 G1 收集器都专注于降低垃圾回收停顿时间，适用于服务器端应用。CMS 通过并发标记和清除实现低延迟，而 G1 通过分区和优先回收垃圾最多的区域来平衡延迟和吞吐量。

10、**Minor GC**（Young Generation GC）和 **Full GC**（Old Generation GC）是两种不同类型的垃圾回收。

Minor GC：

- Minor GC 是针对新生代的垃圾回收，主要清理新生代中的垃圾对象。
- 在 Minor GC 过程中，只有新生代的 Eden 区和 Survivor 区参与垃圾回收，通常采用复制算法。
- Minor GC 的目标是尽可能快速地回收新生代的垃圾对象，保证新生代的空间可以被重复利用，将存活的对象晋升到老年代或幸存者区。

Full GC：

- Full GC 是针对整个堆（包括新生代和老年代）的垃圾回收，也被称为 Major GC 或 Old Generation GC。

- 在 Full GC 过程中，整个堆的所有区域都参与垃圾回收，包括新生代、老年代和持久代（在 JDK1.7 之前）或元空间（在 JDK1.7 之后）。
- Full GC 通常使用标记-清除、标记-整理或复制算法进行垃圾回收。
- Full GC 的目标是彻底清理整个堆，回收尽可能多的内存空间，减少内存碎片，从而避免频繁的垃圾回收。

主要区别：

- Minor GC 针对新生代，回收的是新生代中的垃圾对象；而 Full GC 针对整个堆，回收的是整个堆中的垃圾对象。
- Minor GC 通常发生频率较高，而 Full GC 通常较慢，会导致较长的停顿时间。
- Minor GC 通常发生在新生代的空间不足时，而 Full GC 发生在老年代空间不足或执行 `System.gc()` 方法时，或是为了进行内存整理。

选择合适的垃圾回收器和调整相应的参数，以及对内存和对象的合理管理，是优化应用程序性能的重要因素。

三、类加载

1、类从被加载到虚拟机内存中开始到卸载出内存为止的整个生命周期可以简单概括为以下 7 个阶段：

1. **加载（Loading）**：将类的字节码文件加载到内存，并创建对应的 Class 对象，描述类的结构信息。
2. **验证（Verification）**：对加载的字节码进行验证，确保其符合 JVM 规范和安全性要求。
3. **准备（Preparation）**：为类的静态变量分配内存空间，并设置默认初始值，例如 int 类型为 0，引用类型为 null。
4. **解析（Resolution）**：将符号引用解析为直接引用，即将类、字段、方法等符号引用转换为对应的直接内存指针。
5. **初始化（Initialization）**：执行类的初始化代码，包括静态变量赋值和静态代码块的执行。在此阶段，类的静态变量会被赋予程序中指定的值。
6. **使用（Using）**：对象可以被程序使用，调用其方法，访问其成员变量。
7. **卸载（Unloading）**：当类不再被使用，且没有任何引用时，垃圾回收器会回收类所占用的内存空间，从虚拟机内存中卸载类。

这样的生命周期包括了类从加载到被使用再到最终卸载的整个过程。连接阶段（验证、准备和解析）是在加载阶段之后进行的，而初始化阶段是类被首次主动使用时进行的。

2、**Java 对象的创建过程**（五步）：

```
1. **类加载**：加载类的字节码文件到内存，并在方法区创建一个对应的Class对象，用于描述类的结构信息。
2. **分配内存**：在堆内存中为新创建的对象分配内存空间。
3. **初始化零值**：将分配的内存空间初始化为零值，即对象的成员变量在此阶段都为默认值（例如，int 类型默认为 0，引用类型默认为 null）。
4. **设置对象头**：虚拟机会根据对象的类型设置对象头信息，包括对象的哈希码、GC 分代年龄等。
5. **执行<init>方法**：在构造器方法`<init>`中进行对象的初始化工作，即根据构造器的实际内容为对象的成员变量赋值。
```

3、双亲委派机制是 Java 类加载器的一种工作机制，它的核心思想是当一个类加载器收到类加载请求时，首先会将该请求委派给父类加载器去尝试加载。只有在父类加载器无法加载时，才由当前类加载器自行加载。

好处：

- 安全性：避免恶意类通过伪装加载到核心 API 类，保证核心 API 类的安全性。
- 避免重复加载：如果父类加载器已经加载了一个类，那么子类加载器再次加载同一类时，可以直接返回父类加载器已经加载的类，避免重复加载，节省内存空间。

4、打破双亲委派机制一般是为了实现特定的需求，例如加载自定义的类或资源。

操作：

- 在自定义类加载器中重写 `loadClass` 方法：在重写的 `loadClass` 方法中，根据需要自定义类的加载逻辑，可以通过调用父类加载器的 `loadClass` 方法，或是直接调用 `defineClass` 方法加载类字节码。
- 使用 `Thread.currentThread().setContextClassLoader` ：通过设置线程的上下文类加载器来改变类加载器的搜索路径，从而实现加载自定义类或资源。

具体案例：

- 一些应用服务器（如Tomcat）为了支持不同应用程序之间的隔离，会使用自定义的类加载器实现独立的类加载环境，打破双亲委派机制。
- OSGi框架也是通过自定义类加载器实现模块化加载和隔离的，也是为了打破双亲委派机制。