Java学习笔记—复习

|  |  |
| --- | --- |
|  | 🢂 内容概览 |
|  | Why：此文档用来做什么？它存在的意义是什么？为解决什么问题？   |  | | --- | |  |   What：当前包含了那些内容？   |  | | --- | |  |   How：此文档应如何参考？   |  | | --- | |  |   Who：此文档适用于那些人员阅读参考？   |  | | --- | |  | |

目录

[1 JVM 1](#_Toc444025370)

[1.1 JVM运行时内存 1](#_Toc444025371)

[1.1.1 程序计数器 1](#_Toc444025372)

[1.1.2 虚拟机栈（-Xss） 1](#_Toc444025373)

[1.1.3 本地方法栈（-Xoss） 1](#_Toc444025374)

[1.1.4 Java堆（-Xmx，-Xms，-Xmn） 1](#_Toc444025375)

[1.1.5 方法区（-XX:PermSize / -XX:MaxPermSize） 1](#_Toc444025376)

[1.1.5.1 Class文件常量池 1](#_Toc444025377)

[1.1.5.2 运行时常量池 1](#_Toc444025378)

[1.1.6 直接内存（-XX:MaxDirectMemorySize） 1](#_Toc444025379)

[1.2 GC机制 1](#_Toc444025380)

[1.2.1 GC需完成的3件事： 1](#_Toc444025381)

[1.2.2 What：哪些内存需要回收？ 1](#_Toc444025382)

[1.2.3 堆内存回收 1](#_Toc444025383)

[1.2.3.1 What：如何判断对象需要回收？ 1](#_Toc444025384)

[1.2.3.2 What：哪些对象可以作为GC Roots对象？ 1](#_Toc444025385)

[1.2.3.3 引用的分类 1](#_Toc444025386)

[1.2.3.4 标记对象流程（2次标记） 1](#_Toc444025387)

[1.2.4 方法区内存回收 1](#_Toc444025388)

[1.2.4.1 常量回收 1](#_Toc444025389)

[1.2.4.2 类信息回收 1](#_Toc444025390)

[1.2.5 常用GC算法 1](#_Toc444025391)

[1.2.5.1 标记-清除算法（理论） 1](#_Toc444025392)

[1.2.5.2 复制算法（理论） 1](#_Toc444025393)

[1.2.5.3 标记-整理算法（理论） 1](#_Toc444025394)

[1.2.5.4 分代收集算法（理论） 1](#_Toc444025395)

[1.2.6 hotspot GC回收策略 1](#_Toc444025396)

[1.2.7 垃圾收集器（7个） 1](#_Toc444025397)

[1.2.8 GC过程和GC日志的理解 1](#_Toc444025398)

[1.3 类加载机制 1](#_Toc444025399)

[1.3.1 基本概念 1](#_Toc444025400)

[1.3.2 类加载规则 1](#_Toc444025401)

[1.3.3 类加载时机 1](#_Toc444025402)

[1.3.4 类加载器 1](#_Toc444025403)

[1.3.5 类加载机制 1](#_Toc444025404)

[1.3.6 类加载流程 1](#_Toc444025405)

[1.3.7 自定义类加载器 1](#_Toc444025406)

[1.3.8 自定义类加载器的应用 1](#_Toc444025407)

[1.4 对象创建过程 1](#_Toc444025408)

[1.5 常用虚拟机参数总结 1](#_Toc444025409)

[2 数据库 1](#_Toc444025410)

[2.1 脏读、幻读、重复读 1](#_Toc444025411)

[2.2 事务隔离级别 1](#_Toc444025412)

[2.3 事务传播特性 1](#_Toc444025413)

[2.3.1 配置方式 1](#_Toc444025414)

[2.3.1.1 基于aop 1](#_Toc444025415)

[2.3.1.2 基于注解 1](#_Toc444025416)

# Java基本特性

## NIO

## 反射&代理

### What：如何理解反射？

能够在运行时对类&对象的结构进行解析，并提供操作接口；

### What：如何理解代理？

1. 代理代理分为两种：静态代理和动态代理；
2. 静态代理：是指在编译时通过代码植入实现的代理模式；

缺点：每实现对一个接口的代理，就必须实现相应的代理类，造成代理类泛滥；

1. 动态代理：是指在运行时，通过反射等技术实现代理模式；常用的动态代理方式有两种：java原生动态代理&CGlib动态代理；

### What：java原生代理和CGlib代理有什么区别？

1. java动态代理基于接口实现，类似于静态代理模式，属于其一种改良；基于Proxy类和InvocationHanlder接口实现；

缺点：必须基于接口实现代理；

1. CGlib动态代理使用继承实现，继承并重写父类方法；故父类和相应方法不能是final的；基于MethodInterceptor实现；

缺点：对于final类或者final方法无法实现代理；

# Spring

## Spring理解

### FactoryBean和ApplicationContext的区别

BeanFacotry是spring中比较原始的Factory。如XMLBeanFactory就是一种典型的BeanFactory。原始的BeanFactory无法支持spring的许多插件，如AOP功能、Web应用等。

ApplicationContext接口,它由BeanFactory接口派生而来，因而提供BeanFactory所有的功能。ApplicationContext以一种更向面向框架的方式工作以及对上下文进行分层和实现继承，ApplicationContext包还提供了以下的功能：

• MessageSource, 提供国际化的消息访问

• 资源访问，如URL和文件

• 事件传播

• 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层

1. 利用MessageSource进行国际化：BeanFactory是不支持国际化功能的，因为BeanFactory没有扩展Spring中MessageResource接口。相反，由于ApplicationContext扩展了MessageResource接口，因而具有消息处理的能力(i18N)，具体spring如何使用国际化，以后章节会详细描述。
2. 强大的事件机制(Event) ；基本上牵涉到事件(Event)方面的设计，就离不开观察者模式。不明白观察者模式的朋友，最好上网了解下。因为，这种模式在java开发中是比较常用的，又是比较重要的。 ApplicationContext的事件机制主要通过ApplicationEvent和ApplicationListener这两个接口来提供的，和java swing中的事件机制一样。即当ApplicationContext中发布一个事件的时，所有扩展了ApplicationListener的Bean都将会接受到这个事件，并进行相应的处理。

Spring提供了部分内置事件，主要有以下几种：

ContextRefreshedEvent ：ApplicationContext发送该事件时，表示该容器中所有的Bean都已经被装载完成，此ApplicationContext已就绪可用

ContextStartedEvent：生命周期 beans的启动信号

ContextStoppedEvent: 生命周期 beans的停止信号

ContextClosedEvent：ApplicationContext关闭事件，则context不能刷新和重启，从而所有的singleton bean全部销毁(因为singleton bean是存在容器缓存中的)

虽然，spring提供了许多内置事件，但用户也可根据自己需要来扩展spriong中的事物。注意，要扩展的事件都要实现ApplicationEvent接口。

1. 底层资源的访问；ApplicationContext扩展了ResourceLoader(资源加载器)接口，从而可以用来加载多个Resource，而BeanFactory是没有扩展ResourceLoader
2. 对Web应用的支持；与BeanFactory通常以编程的方式被创建不同的是，ApplicationContext能以声明的方式创建，如使用ContextLoader。当然你也可以使用ApplicationContext的实现之一来以编程的方式创建ApplicationContext实例 。
3. 其它区别；

* BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化，这样，我们就不能发现一些存在的Spring的配置问题。而ApplicationContext则相反，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误。
* BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册；

## Bean的管理

### Spring支持的作用域有哪些？

Spring框架支持如下五种不同的作用域：

1. singleton：在Spring IOC容器中仅存在一个Bean实例，Bean以单实例的方式存在；
2. prototype：一个bean可以定义多个实例；
3. request：每次HTTP请求都会创建一个新的Bean。该作用域仅适用于WebApplicationContext环境；
4. session：一个HTTP Session定义一个Bean。该作用域仅适用于WebApplicationContext环境；
5. globalSession：同一个全局HTTP Session定义一个Bean。该作用域同样仅适用于WebApplicationContext环境；

bean默认的scope属性是singleton，单例实现是非线程安全的。

### Spring框架中bean的生命周期

1. Spring容器读取XML文件中bean的定义并实例化bean。
2. Spring根据bean的定义设置属性值。
3. 如果该Bean实现了BeanNameAware接口，Spring将bean的id传递给setBeanName()方法。
4. 如果该Bean实现了BeanFactoryAware接口，Spring将beanfactory传递给setBeanFactory()方法。
5. 如果任何bean BeanPostProcessors 和该bean相关，Spring调用postProcessBeforeInitialization()方法。
6. 如果该Bean实现了InitializingBean接口，调用Bean中的afterPropertiesSet方法。如果bean有初始化函数声明，调用相应的初始化方法。
7. 如果任何bean BeanPostProcessors 和该bean相关，调用postProcessAfterInitialization()方法。
8. 如果该bean实现了DisposableBean，调用destroy()方法。

### 哪些是最重要的bean生命周期方法？如何重写？

有两个重要的bean生命周期方法。第一个是setup方法，该方法在容器加载bean的时候被调用。第二个是teardown方法，该方法在bean从容器中移除的时候调用。

2）bean标签有两个重要的属性(init-method 和 destroy-method)，你可以通过这两个属性定义自己的初始化方法和析构方法。

3）Spring也有相应的注解：@PostConstruct 和 @PreDestroy。

### 什么是bean自动装配？有哪些方式？有哪些缺点？

Spring容器可以自动配置相互协作beans之间的关联关系。这意味着Spring可以自动配置一个bean和其他协作bean之间的关系，通过检查BeanFactory 的内容里没有使用和< property>元素。自动装配提供五种不同的模式供Spring容器用来自动装配beans之间的依赖注入:

1. no：默认的方式是不进行自动装配，通过手工设置ref 属性来进行装配bean；
2. byName：通过参数名自动装配，Spring容器查找beans的属性，这些beans在XML配置文件中被设置为byName。之后容器试图匹配、装配和该bean的属性具有相同名字的bean；
3. byType：通过参数的数据类型自动自动装配，Spring容器查找beans的属性，这些beans在XML配置文件中被设置为byType。之后容器试图匹配和装配和该bean的属性类型一样的bean。如果有多个bean符合条件，则抛出错误；
4. constructor：这个同byType类似，不过是应用于构造函数的参数。如果在BeanFactory中不是恰好有一个bean与构造函数参数相同类型，则抛出一个严重的错误；
5. autodetect：如果有默认的构造方法，通过constructor的方式自动装配，否则使用 byType的方式自动装配；

缺点：

1. 重写：你仍然需要使用 和< property>设置指明依赖，这意味着总要重写自动装配;
2. 原生数据类型:你不能自动装配简单的属性，如原生类型、字符串和类;
3. 模糊特性：自动装配总是没有自定义装配精确，因此，如果可能尽量使用自定义装配;

### 相关注解

@Configuration它用来标记类，说明作为beans的定义，可以被Spring IOC容器使用。

@Bean注解，它表示该方法定义的Bean要被注册进Spring应用上下文中。

@Required表明bean的属性必须在配置时设置，可以在bean的定义中明确指定也可通过自动装配设置。如果bean的属性未设置，则抛出BeanInitializationException异常。

@Autowired 注解提供更加精细的控制，包括自动装配在何处完成以及如何完成。它可以像@Required一样自动装配setter方法、构造器、属性或者具有任意名称和/或多个参数的PN方法。

@Qualifier当有多个相同类型的bean而只有其中的一个需要自动装配时，将@Qualifier 注解和@Autowire 注解结合使用消除这种混淆，指明需要装配的bean。

## 事务管理

### 事务管理的方式有哪些？

1. 编程式事务管理：这意味着你可以通过编程的方式管理事务，这种方式带来了很大的灵活性，但很难维护。
2. 声明式事务管理：这种方式意味着你可以将事务管理和业务代码分离。你只需要通过注解或者XML配置管理事务。

## IOC

### Spring IoC容器是什么？

Spring IOC负责创建对象、管理对象(通过依赖注入)、整合对象、配置对象以及管理这些对象的生命周期。

优点：

1. 解耦，可扩展；
2. 可测试；
3. 支持单例、延迟加载等；

### Spring中的依赖注入是什么？有哪些类型？使用场景如何？

依赖注入作为控制反转(IOC)的一个层面，可以有多种解释方式。在这个概念中，你不用创建对象而只需要描述如何创建它们。你不必通过代码直接的将组件和服务连接在一起，而是通过配置文件说明哪些组件需要什么服务。之后IOC容器负责衔接。类型如下：

1. 构造器依赖注入：构造器依赖注入在容器触发构造器的时候完成，该构造器有一系列的参数，每个参数代表注入的对象。
2. Setter方法依赖注入：首先容器会触发一个无参构造函数或无参静态工厂方法实例化对象，之后容器调用bean中的setter方法完成Setter方法依赖注入。
3. 可以同时使用两种方式的依赖注入，最好的选择是使用构造器参数实现强制依赖注入，使用setter方法实现可选的依赖关系。

## AOP

Aop分为：前置增强、后置增强、环绕增强、抛出增强、引入增强；

### aop实现方式有哪些？

1. 编程式

ProxyFactory：setTarget，setAdvice，getProxy

MethodBeforeAdvice：前置

AfterReturningAdvice：后置

MethodInterceptor：环绕

ThrowsAdvice：抛出

DelegatingIntroductionInterceptor：引入

1. 声明式

配置ProxyFactoryBean

# JVM

## JVM运行时内存

1. 程序计数器：线程私有
2. 虚拟机栈：线程私有
3. 本地方法栈：线程私有
4. 堆
5. 方法区
6. 直接内存块；

### 程序计数器

1. 当前线程执行字节码的行号指示器；
2. 字节码解释器通过改变程序计数器的值来选取下一条要执行的字节码指令；
3. 若正在执行的是java方法，计数器存放正在执行的字节码指令的内存地址；如果正在执行的是native方法，则值为空；
4. 生命周期与线程相同；

### 虚拟机栈（-Xss）

1. 描述java方法执行的内存模型；方法调用、参数入栈出栈；
2. 每个方法执行时都会创建一个栈桢，用于存放局部变量表，操作数栈，动态连接，方法出口等信息；
3. 每个方法从调用直至执行完成的过程，对应这一个栈桢在虚拟机中从入栈到出栈的过程；
4. 申请范围超过栈深度，抛出StackOverFlowException；申请内存不足，抛出OutOfMemoryException；

|  |
| --- |
| **问：虚拟机栈在方法执行过程中是如何运转的？** |

|  |
| --- |
| **问：栈桢的结构是怎样的？** |

### 本地方法栈（-Xoss）

1. 描述Java本地方法执行的内存模型；
2. 申请范围超过栈深度，抛出StackOverFlowException；申请内存不足，抛出OutOfMemoryException；
3. 虚拟机规范对实现无强制规定，可与虚拟机栈合并；
4. hotspot虚拟机本地方法栈和虚拟机栈合并；

### Java堆（-Xmx，-Xms，-Xmn）

1. 存放对象的主要区域；
2. GC管理的主要区域；
3. 从内存回收的角度看，根据分带收集算法可分为新生代和老年代；新生代可以分为Eden、from survivor和to survivor三个区；
4. 从内存分配的角度看，线程共享的Java堆可能划分出多个私有的分配缓冲区（Thread Local Allocation Buffer，TLAB）；

|  |
| --- |
| **问：为什么会有TLAB？其作用是什么？** |

1. 堆内存的划分方式与内容无关，只是为了更好的回收内存，或者更快的分配内存；
2. 堆可以存在与逻辑上连续，物理上无需连续的内存空间中；
3. 无法分配内存，则抛出OutOfMemoryError异常；

|  |
| --- |
| **问：堆的生命周期是怎样的？是否是贯穿虚拟机运行始末？** |

### 方法区（-XX:PermSize / -XX:MaxPermSize）

1. 存储已被虚拟机加载的类信息（类实例）、常量、静态常量、即时编译器编译的代码等数据；
2. 也是GC回收区域，回收内存包括对常量池的内存回收和对类型的卸载；
3. 内存不足，将抛出OutOfMemoryError；

|  |
| --- |
| **问：方法区的生命周期是怎样的？** |

|  |
| --- |
| **问：类加载后的类对象存放在哪个区域？** |

|  |
| --- |
| **问：静态对象存放在哪个区域？** |

#### Class文件常量池

虚拟机必须为每个被装载的类型维护一个常量池。常量池就是该类型所用常量的一个有序集合，包括直接常量（String ,integer和floating point常量）和对其他类型、字段和方法的符号引用。池中的数据项就像数组一样是通过索引访问的。因为常量池存储了相应类型所用到的所有类型、字段和方法的符号引用，所以它在Java程序的动态连接中起着核心作用。

|  |
| --- |
| http://static.oschina.net/uploads/img/201412/17092921_nWj0.jpg |

#### 运行时常量池

1. 方法区的一部分；
2. 存放编译时生成的各种字面量（常量）和符号引用；
3. 存放翻译出来的直接引用；
4. 若无法分配内存，则抛出OutOfMemoryError；

|  |
| --- |
| **问：什么是符号引用？什么是直接引用？** |

|  |
| --- |
| **问：运行时常量池和Class文件常量池有什么区别？** |

|  |
| --- |
| 问：如何使用String的intern方法？ |

### 直接内存（-XX:MaxDirectMemorySize）

1. 不属于运行时数据区的一部分；
2. NIO可以通过Native方法直接分配堆外内存；
3. 不受JVM参数管理，可能抛出OutOfMemoryError；
4. 直接内存大小默认与堆大小一致；

## GC机制

### GC需完成的3件事：

1. what：哪些内存需要回收？（堆、方法区）
2. when：什么时候回收？
3. how：如何回收？

### What：哪些内存需要回收？

1. 堆（不可达对象）；
2. 方法区（常量池内存、类型卸载）的内存回收；

### 堆内存回收

#### What：如何判断对象需要回收？

##### 引用计数（理论）

通过给每个对象维护一个引用计数器，记录当前对象被引用的数量；引用数量为0时，对象即可被回收；

优点：效率高，实现简单；

缺点：无法解决循环引用问题；

##### 可达性分析（主流）

通过一系列称为GCRoots的对象作为起始点，当一个对象到GCRoots不可达时，证明对象不可用；

#### What：哪些对象可以作为GC Roots对象？

1. 虚拟机栈中的引用对象；（栈中的本地变量表）
2. 方法区中类静态属性引用的对象；
3. 方法区中常量引用的对象；
4. 本地方法栈中JNI引用的对象；（Native方法）

#### 引用的分类

引用分为如下4种：

1. 强引用：代码中直接定义的引用，引用存在，对象不回收；
2. 软引用：SoftReference，非必需对象，在发生溢出前，回收此类对象；
3. 弱引用：WeakReference，生存到下一次回收之前；
4. 虚引用：PhantomReference，对对象的生存无影响，用作GC时，收到系统通知；

#### 标记对象流程（2次标记）

1. 无GCRoots引用，判断是否执行finalize()方法；
2. 若未执行finalize()方法，则放入F-Queue中，等待调用，不保证finalize()执行完；
3. 扫描F-Queue对象，若恢复引用，则移出队列，不进行回收；
4. finalize()方法由低优先级线程Finalizer执行；
5. finalize方法只能被调用一次；
6. finalize方法不建议使用；

### 方法区内存回收

#### 常量回收

判断常量是否被引用；

#### 类信息回收

1. 该类的所有实例均被回收；
2. 加载类的ClassLoader被回收；
3. 对应的Class对象没有被引用，没有反射访问Class的方法；

### 常用GC算法

#### 标记-清除算法（理论）

##### 思路：

* 整个GC过程分为标记过程和清除过程两步；
* 标记过程：进行可达性分析, 标记待清除的对象；分析过程中不允许对象被访问，会产生GC停顿；
* 清除过程：执行清除操作，回收对象内存；因为堆内存的不连续性，会产生内存碎片；

##### 缺点：

* 效率低；
* 碎片多；

##### 适用场景

* 不关注GC停顿的应用；
* 少量对象回收；

#### 复制算法（理论）

##### 思路：

* 将内存分为大小相等的两块内存；当一块内存用完时，将其中存活的对象拷贝到另一块内存；然后将原先内存全部清理；如此往复；

##### 优点：

* 内存连续；
* 操作效率高；

##### 缺点：

* 内存消耗多，使用率不高；
* 对于存活时间较长的对象，回收效率不高；

##### 适用场景：

* 存活时间短的对象回收；

#### 标记-整理算法（理论）

##### 思路：

* 整个GC过程分为标记过程和整理过程两步；
* 标记过程：进行可达性分析, 标记待清除的对象；分析过程中不允许对象被访问，会产生GC停顿；
* 向一端移动存活对象，直接清理边界以外内存；

##### 优点：

* 内存碎片少；

##### 缺点：

* 效率低；

##### 适用场景：

* 不关注GC停顿的应用；

#### 分代收集算法（理论）

##### 思路：

* 将整个GC收集过程分为新生代和老年代两代；
* 新生代采用复制算法，使得对存活时间较短的对象尽快回收；
* 老年代采用标记-清除、标记-整理算法，回收存活时间较长的对象；

##### 优点：

* 能够对存活时长不同的对象使用不同的回收策略，使得整体性能最大化；
* 能够快速回收存活时间较短的对象；存活时间短的对象一般占多数，能够提高回收效率；

### hotspot GC回收策略

1. 可达性分析过程，必须保证程序停顿到某一镜像；即分析过程中，虚拟机中所有对象的引用关系不能改变；
2. JVM使用准确式GC，保证系统知道哪些位置存放引用，而非遍历所有位置；
3. Hotspot中使用一组称为OopMap的数据结构，在类加载时，计算该类的对象在什么偏移量上是什么类型；以便后续使用时，不必再次对类进行解析；
4. JIT编译过程中，也会在特定位置记录栈，寄存器的位置引用；
5. 安全点：hotspot只有在特定位置才生成OopMap，GC到达这些位置时才会进行；Safepoint在”长时间执行”的位置，”长时间”最明显的特征是指令序列的复用；
6. 如何确保所有的线程都在Safepoint中断：

* 抢先式中断；
* 主动式中断；

1. 安全区域：在一段代码执行的区间，引用关系不发生改变，则GC可不关注这些线程。当线程要离开区域时，判断GC是否完成，以决定线程是否继续执行；

|  |
| --- |
| **问：什么是准确式GC?** |

|  |
| --- |
| **问：为什么要设置Safepoint？** |

### 垃圾收集器（7个）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **收集器** | **清理算法** | **回收代** | **搭配收集器** | **原理** |
| Serial | 复制算法 | 新生代 | CMS  SerialOld | 使用一个cpu/一个线程完成GC； |
| ParNew | 复制算法 | 新生代 | CMS  SerialOld | Serial的多线程版本； |
| Parallel Scavenge | 复制算法 | 新生代 | SerialOld  ParallelOld | 多线程，以提高吞吐为目标，而非缩短停顿时间；  有GC自适应调节策略； |
| CMS | 标记-清除算法 | 老年代 | Serial  ParNew | Concurrent Mark Sweep，并发标记，并发清除；以获得最1短停顿时间为目标；  回收过程：   1. 初始标记：标记GCRoots能够直接关联到的对象；（停顿） 2. 并发标记：执行可达性分析； 3. 重新标记：修正并发标记过程中产生的变动；（停顿） 4. 并发清除：执行清除操作； |
| SerialOld | 标记-整理算法 | 老年代 | Serial  ParNew | 单线程 |
| ParallelOld | 标记-整理算法 | 老年代 | ParallelOld | 多线程，以提高吞吐为目标，而非缩短停顿时间； |
| G1 | 标记-整理 | 皆可 |  | 1. 面向server模式，替换CMS收集器； 2. 并行&并发； 3. 分代收集； 4. 空间整合； 5. 可预测停顿：能够指定单位时间内停顿的时长； 6. 划分内存为多个region，避免回收整个内存； 7. 维护region回收价值列表，优先回收价值高的region； 8. 回收过程：  * 初始标记：标记GC Roots直接关联的对象；（停顿） * 并发标记：标记存活对象；（并发） * 重新标记：修正并发标记过程中变动；（停顿，并发） * 筛选回收：根据回收价值成本对region排序，根据设定时间回收最有价值的部分； |

|  |  |  |  |
| --- | --- | --- | --- |
| **收集器** | **适用场景** | **优点** | **缺点** |
| Serial | Client模式下的虚拟机； | 消耗资源小； | 停顿时间可能稍长； |
| ParNew | Server模式下与CMS搭配 | GC速度提高，减少停顿时间； |  |
| Parallel Scavenge | 后台运算而不需要太多交互的任务； | 提高吞吐；  GC自适应调节策略； | 停顿时间可能稍长； |
| Serial Old | Client模式下的虚拟机；  Server模式下作为CMS的后备方案； | 消耗资源小； | 停顿时间可能稍长； |
| Parallel Old | 配合Parallel Scavenge使用，适合于注重吞吐的场合； | 提高吞吐； | 停顿时间可能稍长； |
| CMS | 适合交互频繁，停顿少的情况； | 停顿较少 | 1. CPU消耗高； 2. 无法处理浮动垃圾（GC过程中产生的垃圾）； 3. 易产生Concurrent Mode Failure，而导致Full GC；（清理过程中，产生的对象超过老年代内存剩余量） 4. 碎片较多，造成额外的FullGC；（大对象分配时，不容易找到适合的内存） |
| G1 | Server模式，替换CMS； |  |  |

|  |
| --- |
| 问：如何理解Parallel Scavenge的GC自适应调节策略？ |

|  |
| --- |
| **注：**  🡪 没有完美的收集器；  🡪 没有放之四海而皆准的道理和准则；  🡪 拥抱变化；  🡪 没有不限范围的万能钥匙； |

### GC过程和GC日志的理解

## 类加载机制

### 基本概念

1. Jvm执行一个程序无论多么复杂，都是一个独立的JVM进程；
2. 类加载器负责加载类文件，包括系统类加载器和用户自定义类加载器；
3. 类加载过程：类文件加载和连接；

### 类加载规则

1. 若当前类未加载，则进行加载；
2. 若当前类的父类未加载，则进行加载；🡪JVM总是最先初始化父类
3. 若当前类中有初始化语句，则依次执行；

### 类加载时机

1. 创建类的实例：new操作，反射，反序列化；
2. 调用类变量/类方法（static）；
3. 使用反射创建类的实例：Class.forName(className);
4. 初始化某个类的子类；
5. 直接使用java运行某个类；

|  |
| --- |
| 注：final类型变量如果在编译时值就可以确定下来（常量，不调用方法），则在编译时变量就已经被值替换，相当于宏变量，故调用static final类变量可能不会触发类加载； |

|  |
| --- |
| 注：调用ClassLoader类的loadClass()方法来加载某个类时，该方法只是加载该类，并没有进行初始化；使用Class的forName()方法才会初始化该类； |

### 类加载器

1. 类加载器负责加载jvm运行过程中需要用到的类，并在内存中生成相应的Class的实例；已经加载过的类将不会被再次加载；
2. 一个类使用类的全名和类加载器实例作为唯一标识；🡪两个类加载器加载同名的类，产生的实例是不同的；
3. JVM启动时，会生成由三个类加载器构成的初始类加载器层次结构：

* Boostrap ClassLoader：根类加载器（引导类加载器），负责加载java核心类；该类由JVM自己实现，并非是ClassLoader的子类；即$JAVA\_HOME/jre/lib下面的jar；（一般非JAVA实现）
* Extension ClassLoader：扩展类加载器；负责加载扩展核心类以外的功能，用于扩展jvm功能；即：$JAVA\_HOME/jre/lib/ext路径下的类或者java.ext.dirs指定的目录中的jar；
* System ClassLoader：系统类加载器；负责加载java –classpath选项、java.class.path系统属性、CLASSPATH环境变量中指定的jar包和类路径；可以通过ClassLoader的getSystemClassLoader方法获取系统类加载器；

1. 如果没有特别指定，用户自定义类加载器都以类加载器作为父加载器；

### 类加载机制

1. 全盘负责：当类加载器加载某一个类时，该类所依赖和引用的所有其它Class也由该类加载器加载；除非显式使用另外的类加载器加载；
2. 父类委托：先让父类加载器加载该Class，如果父类加载器无法加载，才尝试自己从类加载器中加载；
3. 缓存机制：所有加载过的类都被缓存起来，当程序中需要加载某个类时。类加载器先从缓存搜索该类是否存在，如果不存在，才会进行加载；

### 类加载流程

1. 检测当前类是否加载过（即在缓存中是否有该类实例）；如果有，直接到第8步；
2. 检测当前类加载器有无父类加载器；不存在，到第4步；存在，到第5步；
3. 请求使用父类加载器加载目标类；成功，直接到第8步；失败，到第5步；
4. 请求使用根类加载其加载目标类；成功，到第8步；失败，到第7步；
5. 当前类加载器尝试寻找待加载的Class文件；找到，到第6步；未找到，到第7步；
6. 加载Class文件；成功，到第8步；
7. 抛出：ClassNotFoundException；
8. 返回对应的Class对象；

### 自定义类加载器

ClassLoader包含两个关键方法，可以通过重写如下方法来定义自己的类加载器，推荐重写findClass方法；

1. loadClass(String name , Boolean resolve)：根据指定类名加载类文件；调用步骤：

* 调用findLoadedClass（String）检查是否已经加载类；如果加载，则直接返回；
* 调用父类加载器的loadClass方法；如果父类加载为NULL，则调用根类加载器；
* 调用findClass(String)方法查找类；

1. findClass(String name)：根据指定类名查找类；
2. defineClass()：负责读取指定的字节码文件到byte[]中，并将byte[]转换为Class对象，并校验有效性；该方法无法重写；

### 自定义类加载器的应用

1. 在执行代码前自动验证数字签名；
2. 实现代码混淆器，避免反编译代码；
3. 根据用户需求动态加载类；
4. 根据应用需求把其它数据以字节码形式加载到应用中；

## 对象创建过程

## 常用虚拟机参数总结

|  |  |  |
| --- | --- | --- |
| **虚拟机参数** | **用途** | **备注** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# 数据库

## 脏读、幻读、重复读

1. **脏读:** 指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据， 那么另外一个事务读到的这个数据是脏数据，依据脏数据所做的操作可能是不正确的；
2. **不可重复读:** 指在一个事务内，多次读同一数据。在这个事务还没有结束时，另外一个事务也访问该同一数据。 那么，在第一个事务中的两次读数据之间，由于第二个事务的修改，那么第一个事务两次读到的数据可能是不一样的。这样就发生了在一个事务内两次读到的数据是不一样的，因此称为是不可重复读；
3. **幻觉读:** 指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象发生了幻觉一样；（分页）

## 事务隔离级别

1. ISOLATION\_DEFAULT： 这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔离级别.另外四个与JDBC的隔离级别相对应；
2. ISOLATION\_READ\_UNCOMMITTED： 这是事务最低的隔离级别，它充许令外一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读，不可重复读和幻像读；
3. ISOLATION\_READ\_COMMITTED： 保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据；（oracle默认）
4. ISOLATION\_REPEATABLE\_READ： 这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻像读。它除了保证一个事务不能读取另一个事务未提交的数据外，还保证了避免下面的情况产生(不可重复读)；（mysql默认）
5. ISOLATION\_SERIALIZABLE 这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。除了防止脏读，不可重复读外，还避免了幻像读；

## 事务传播特性

### 配置方式

#### 基于aop

|  |
| --- |
| <tx:advice id="txAdvice" transaction-manager="txManager">     <tx:attributes>       <tx:method name="add\*" propagation="REQUIRED" isolation="READ\_COMMITTED"/>增加记录的方法       <tx:method name="get\*" propagation="REQUIRED" isolation="READ\_COMMITTED"/>获取记录的方法       <tx:method name="delete\*" propagation="REQUIRED" isolation="READ\_COMMITTED"/>删除的方法       <tx:method name="update\*" propagation="REQUIRED" isolation="SERIALIZABLE"/>更改记录的方法     </tx:attributes>   </tx:advice> |

#### 基于注解

|  |
| --- |
| *<!-- 启用对事务注解的支持 -->* <**tx:annotation-driven transaction-manager="transactionManager"**/>  @Transactional(propagation=Propagation.NOT\_SUPPORTED,readOnly=true) |

1. REQUIRED：业务方法需要在一个事务中运行。如果方法运行时，已经处在一个事务中，那么加入到该事务，否则为自己创建一个新的事务；（spring默认为required）
2. NOT\_SUPPORTED：声明方法不需要事务。如果方法没有关联到一个事务，容器不会为它开启事务。如果方法在一个事务中被调用，该事务会被挂起，在方法调用结束后，原先的事务便会恢复执行；
3. REQUIRESNEW：属性表明不管是否存在事务，业务方法总会为自己发起一个新的事务。如果方法已经运行在一个事务中，则原有事务会被挂起，新的事务会被创建，直到方法执行结束，新事务才算结束，原先的事务才会恢复执行；
4. MANDATORY：该属性指定业务方法只能在一个已经存在的事务中执行，业务方法不能发起自己的事务。如果业务方法在没有事务的环境下调用，容器就会抛出例外；
5. SUPPORTS：这一事务属性表明，如果业务方法在某个事务范围内被调用，则方法成为该事务的一部分。如果业务方法在事务范围外被调用，则方法在没有事务的环境下执行。
6. Never：指定业务方法绝对不能在事务范围内执行。如果业务方法在某个事务中执行，容器会抛出例外，只有业务方法没有关联到任何事务，才能正常执行；
7. NESTED：如果一个活动的事务存在，则运行在一个嵌套的事务中. 如果没有活动事务, 则按REQUIRED属性执行.它使用了一个单独的事务， 这个事务拥有多个可以回滚的保存点。内部事务的回滚不会对外部事务造成影响。它只对DataSourceTransactionManager事务管理器 起效；外部事务回滚，内部事务也需要回滚；