#### 超链接函数

超链接函数的使用：链接到指定的文件，点击打开指定的文件，如= HYPERLINK（"C:\Users\nl\Documents\11.doc","打开文件11"）

#### 文本函数

TEXT，可以将根据指定的数值格式将数字转成文本，第一个参数是内容，第二个是格式，如TEXT($A3-B$1,"yy-mm-dd")

# Servlet&JVM优化

## Servlet

JavaEE企业级规范|标准。开发基于网络请求-应答模型的应用，代码应该遵循Servlet标准规范。标准中定义，请求的类型必须是ServletRequest，服务器响应类型必须是ServletResponse，服务器中的服务组件类型必须是Servlet，服务器提供的唯一服务方法一定是void service(ServletRequest, ServletResponse)方法。

遵循这套规范，提供服务端容器的，称为中间件提供商（Tomcat）；遵循这套规范开发提供服务的，称为服务提供者（编写Servlet相关代码的）。

### Servlet结构

#### Servlet

是Servlet规范的顶级接口。核心关注方法是服务方法。

public void service(ServletRequest request, ServletResponse response) throws IOException, ServletException;

这个方法是Servlet规范中唯一的服务方法。

#### GenericServlet

Servlet规范提供的Servlet接口适配器类型。如果开发中，只针对service方法做强化处理开发，建议继承此类型

适配器模式：是连接两个不同类型的中间桥梁。可以通过这个桥梁，将两个不想关的类型连接起来。

interface Servlet{}

class GenericServlet implements Servlet{}

class MyServlet extends GenericServlet{}

class Main(){

public static void main(String[] args){

Servlet s = (Servlet) new MyServlet();

}

}

#### HttpServlet

是GenericServlet的子类型，是依赖于Http协议的Servlet实现。因为在这个类型中的唯一方法service中，直接进行方法参数的强制类型转换。

这个类提供了一个受保护的service方法。方法签名是：

protected void service(HttpServlertRequest, HttpServletResponse)throws IOException, ServletException;

此方法和http协议耦合，只处理基于http协议的请求-应答模型。方法中提供了模板实现，根据不同的请求方式，调用不同的处理逻辑，如：get请求，调用doGet方法；post请求调用doPost方法。这种设计好处在于，适用于Rest开发模式下的请求处理。可以根据不同的请求方式，提供不同的处理业务。

web.xml

<servlet>

<servlet-name>userServlet</servlet-name>

<servlet-class>UserServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>userServlet</servlet-name>

<url-pattern>/user/\*</url-pattern>

</servlet-mapping>

public class UserServlet extends HttpServlet{

public void doPost(){ // 调用service中的新增方法 }

public void doPut(){ // 调用service中的修改方法 }

public void doDelete(){ // 调用service中的删除方法 }

public void doGet(){

String pathInfo = request.getPathInfo(); // 就是web.xml中配置的url-pattern内的\*

if(pathInfo.equals("...")){ ... }

}

}

### Filter

过滤器。Servlet标准规范中的一个技术体系。

核心功能是：为Servlet提供一个安全的服务环境。并设置统一的配置信息。Filter执行的时候，Servlet是未调用状态。服务未提供。在服务提供前，完成所有的安全校验和统一配置，可以节省服务器资源消耗。

主要处理内容：安全校验、设置字符集、静态资源处理

实现接口Filter，核心方法是void doFilter(ServletRequest, ServletResponse,FilterChain)

过滤器（Filter） VS 拦截器（Interceptor）

拦截器是做什么的？是给服务增加额外功能的代码。如：日志、事务、授权、登录成功后的资源初始化、退出成功后的资源回收。

理论上，在过滤器中可以实现的逻辑，在拦截器中都可以实现。反之不可。

如果将过滤需要实现的逻辑，附加到拦截器上，会造成服务器资源的额外消耗。因为每个拦截器执行，意味着，服务已经开始执行了。

### Listener

监听器。监听各种事件，处理不同的事件。

常用Listener：ServletContextListener监听ServletContext初始化和销毁的监听器, HttpSessionListener 监听HttpSession初始化和销毁的监听器, HttpSessionAttributeListener 监听HttpSession中的Attribute变更的监听器, HttpSessionBindingListener 监听HttpSession中的Attribute绑定的监听器, ServletRequestListener 监听ServletRequest请求对象创建销毁的监听器, ServletRequestAttributeListener 监听ServletRequest中Attribute变更的监听器。

开发应用的时候，自开发监听器的场景比较少。使用较多。如：ContextLoaderListener。

Listener一般都会配合<init-param>实现参数的定制。

### MVC模式

MVC：分层，模型层（module）、视图层（view）、控制层（controller）。

一般情况下，MVC框架指哪一个层次的开发框架？控制层框架。如：SpringMVC、Struts

三个层次中，最不好划分的就是模型层。在WEB应用中，除视图逻辑和控制器外都是模型层代码。

MVC框架特点：一定有一个核心控制器（核心Servlet），如：SpringMVC中的DispatcherServlet。在传统的MVC框架中，核心控制器是使用Servlet开发实现的。核心控制器是框架与客户端的唯一接触点（单点接触）。单点接触的好处是：服务代码和客户端请求解耦。MVC框架中都会提供一个服务顶级接口，和若干适配器类型。让开发者可以快速开发。如：SpringMVC中的接口Controller。

框架侵入性（使用某技术的时候，对应的开发的代码是否可以只依赖JVM运行）：可以只依赖JVM运行的，框架技术成为无侵入性。有侵入性的框架，有人成为侵入性重量级。

框架量级（重量级|轻量级）：指封装级别。必须依赖企业级中间件容器的框架成为重量级框架，如：EJB。不依赖企业级中间件容器的是轻量级框架，如：SpringMVC、Spring、MyBatis。

### 企业开发中异常处理

异常怎么处理？

异常的处理：代码中，绝对不能直接throws。

异常的传递：代码中，除自开发的根意外，不能只try..catch。

代码举例：

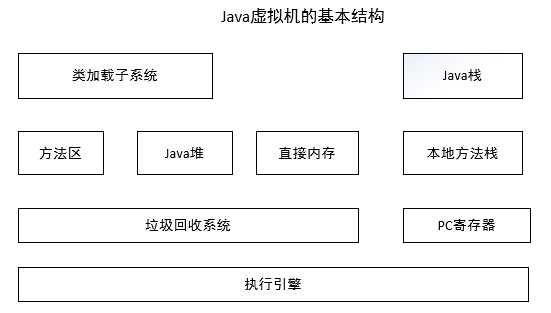
Mapper|DAO: 如果是手写的，代码必须try..catch，处理异常，并将catch捕获的异常封装后（也可以不封装）再抛出。通知调用者，当前方法出现了异常。

service：代码必须try..catch，且将捕获的异常封装后再抛出。

controller：只try..catch，除非自定义了ExceptionHandler。

## JVM优化

### JVM简单结构图



#### 类加载子系统与方法区：

类加载子系统负责从文件系统或者网络中加载Class信息，加载的类信息存放于一块称为方法区的内存空间。除了类的信息外，方法区中可能还会存放运行时常量池信息，包括字符串字面量和数字常量（这部分常量信息是Class文件中常量池部分的内存映射）。

#### Java堆

java堆在虚拟机启动的时候建立，它是java程序最主要的内存工作区域。几乎所有的java对象实例都存放在java堆中。堆空间是所有线程共享的，这是一块与java应用密切相关的内存空间。

#### 直接内存

java的NIO库允许java程序使用直接内存。直接内存是在java堆外的、直接向系统申请的内存空间。通常访问直接内存的速度会优于java堆。因此出于性能的考虑，读写频繁的场合可能会考虑使用直接内存。由于直接内存在java堆外，因此它的大小不会直接受限于Xmx指定的最大堆大小，但是系统内存是有限的，java堆和直接内存的总和依然受限于操作系统能给出的最大内存。

#### 垃圾回收系统

垃圾回收系统是java虚拟机的重要组成部分，垃圾回收器可以对方法区、java堆和直接内存进行回收。其中，java堆是垃圾收集器的工作重点。和C/C++不同，java中所有的对象空间释放都是隐式的，也就是说，java中没有类似free()或者delete()这样的函数释放指定的内存区域。对于不再使用的垃圾对象，垃圾回收系统会在后台默默工作，默默查找、标识并释放垃圾对象，完成包括java堆、方法区和直接内存中的全自动化管理。

#### Java栈

每一个java虚拟机线程都有一个私有的java栈，一个线程的java栈在线程创建的时候被创建，java栈中保存着帧信息，java栈中保存着局部变量、方法参数，同时和java方法的调用、返回密切相关。

#### 本地方法栈

本地方法栈和java栈非常类似，最大的不同在于java栈用于方法的调用，而本地方法栈则用于本地方法的调用，作为对java虚拟机的重要扩展，java虚拟机允许java直接调用本地方法（通常使用C编写）

#### PC寄存器

PC（Program Counter）寄存器也是每一个线程私有的空间，java虚拟机会为每一个java线程创建PC寄存器。在任意时刻，一个java线程总是在执行一个方法，这个正在被执行的方法称为当前方法。如果当前方法不是本地方法，PC寄存器就会指向当前正在被执行的指令。如果当前方法是本地方法，那么PC寄存器的值就是undefined

#### 执行引擎

执行引擎是java虚拟机的最核心组件之一，它负责执行虚拟机的字节码，现代虚拟机为了提高执行效率，会使用即时编译(just in time)技术将方法编译成机器码后再执行。

Java HotSpot Client VM(-client)，为在客户端环境中减少启动时间而优化的执行引擎；本地应用开发使用。（如：eclipse）

Java HotSpot Server VM(-server)，为在服务器环境中最大化程序执行速度而设计的执行引擎。应用在服务端程序。（如：tomcat）

Java HotSpot Client模式和Server模式的区别

当虚拟机运行在-client模式的时候,使用的是一个代号为C1的轻量级编译器, 而-server模式启动的虚拟机采用相对重量级,代号为C2的编译器. C2比C1编译器编译的相对彻底,服务起来之后,性能更高

JDK安装目录/jre/lib/（x86、i386、amd32、amd64）/jvm.cfg

文件中的内容，-server和-client哪一个配置在上，执行引擎就是哪一个。如果是JDK1.5版本且是64位系统应用时，-client无效。

--64位系统内容

-server KNOWN

-client IGNORE

--32位系统内容

-server KNOWN

-client KNOWN

注意：在部分JDK1.6版本和后续的JDK版本(64位系统)中，-client参数已经不起作用了，Server模式成为唯一

### 堆结构及对象分代

#### 什么是分代，分代的必要性是什么

Java虚拟机根据对象存活的周期不同，把堆内存划分为几块，一般分为新生代、老年代和永久代（对HotSpot虚拟机而言），这就是JVM的内存分代策略。

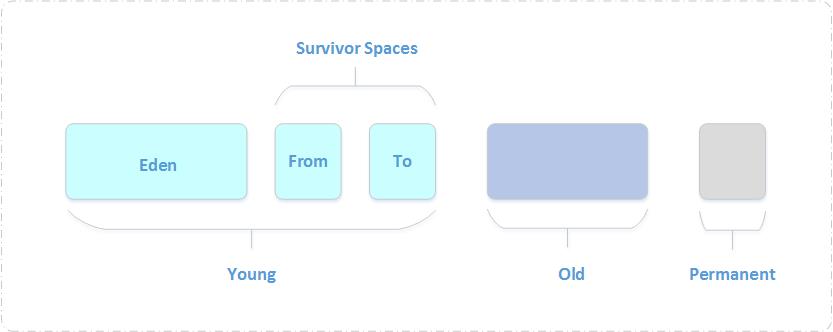
堆内存是虚拟机管理的内存中最大的一块，也是垃圾回收最频繁的一块区域，我们程序所有的对象实例都存放在堆内存中。给堆内存分代是为了提高对象内存分配和垃圾回收的效率。试想一下，如果堆内存没有区域划分，所有的新创建的对象和生命周期很长的对象放在一起，随着程序的执行，堆内存需要频繁进行垃圾收集，而每次回收都要遍历所有的对象，遍历这些对象所花费的时间代价是巨大的，会严重影响我们的GC效率。

有了内存分代，情况就不同了，新创建的对象会在新生代中分配内存，经过多次回收仍然存活下来的对象存放在老年代中，静态属性、类信息等存放在永久代中，新生代中的对象存活时间短，只需要在新生代区域中频繁进行GC，老年代中对象生命周期长，内存回收的频率相对较低，不需要频繁进行回收，永久代中回收效果太差，一般不进行垃圾回收，还可以根据不同年代的特点采用合适的垃圾收集算法。分代收集大大提升了收集效率，这些都是内存分代带来的好处。

#### 分代的划分

Java虚拟机将堆内存划分为新生代、老年代和永久代，永久代是HotSpot虚拟机特有的概念（JDK1.8之后为metaspace替代永久代），它采用永久代的方式来实现方法区，其他的虚拟机实现没有这一概念，而且HotSpot也有取消永久代的趋势，在JDK 1.7中HotSpot已经开始了“去永久化”，把原本放在永久代的字符串常量池移出。永久代主要存放常量、类信息、静态变量等数据，与垃圾回收关系不大，新生代和老年代是垃圾回收的主要区域。

内存简图如下：



##### 新生代（Young Generation）

新生成的对象优先存放在新生代中，新生代对象朝生夕死，存活率很低，在新生代中，常规应用进行一次垃圾收集一般可以回收70% ~ 95% 的空间，回收效率很高。

HotSpot将新生代划分为三块，一块较大的Eden（伊甸）空间和两块较小的Survivor（幸存者）空间，默认比例为8：1：1。划分的目的是因为HotSpot采用复制算法来回收新生代，设置这个比例是为了充分利用内存空间，减少浪费。新生成的对象在Eden区分配（大对象除外，大对象直接进入老年代），当Eden区没有足够的空间进行分配时，虚拟机将发起一次Minor GC。

GC开始时，对象只会存在于Eden区和From Survivor区，To Survivor区是空的（作为保留区域）。GC进行时，Eden区中所有存活的对象都会被复制到To Survivor区，而在From Survivor区中，仍存活的对象会根据它们的年龄值决定去向，年龄值达到年龄阀值（默认为15，新生代中的对象每熬过一轮垃圾回收，年龄值就加1，GC分代年龄存储在对象的header中）的对象会被移到老年代中，没有达到阀值的对象会被复制到To Survivor区。接着清空Eden区和From Survivor区，新生代中存活的对象都在To Survivor区。接着， From Survivor区和To Survivor区会交换它们的角色，也就是新的To Survivor区就是上次GC清空的From Survivor区，新的From Survivor区就是上次GC的To Survivor区，总之，不管怎样都会保证To Survivor区在一轮GC后是空的。GC时当To Survivor区没有足够的空间存放上一次新生代收集下来的存活对象时，需要依赖老年代进行分配担保，将这些对象存放在老年代中。

##### 老年代（Old Generationn）

在新生代中经历了多次（具体看虚拟机配置的阀值）GC后仍然存活下来的对象会进入老年代中。老年代中的对象生命周期较长，存活率比较高，在老年代中进行GC的频率相对而言较低，而且回收的速度也比较慢。

##### 永久代（Permanent Generationn）

永久代存储类信息、常量、静态变量、即时编译器编译后的代码等数据，对这一区域而言，Java虚拟机规范指出可以不进行垃圾收集，一般而言不会进行垃圾回收。

### 垃圾回收算法及分代垃圾收集器

#### 垃圾收集器的分类

##### 次收集器

Scavenge GC，指发生在新生代的GC，因为新生代的Java对象大多都是朝生夕死，所以Scavenge GC非常频繁，一般回收速度也比较快。当Eden空间不足以为对象分配内存时，会触发Scavenge GC。

一般情况下，当新对象生成，并且在Eden申请空间失败时，就会触发Scavenge GC，对Eden区域进行GC，清除非存活对象，并且把尚且存活的对象移动到Survivor区。然后整理Survivor的两个区。这种方式的GC是对年轻代的Eden区进行，不会影响到年老代。因为大部分对象都是从Eden区开始的，同时Eden区不会分配的很大，所以Eden区的GC会频繁进行。因而，一般在这里需要使用速度快、效率高的算法，使Eden去能尽快空闲出来。

当年轻代堆空间紧张时会被触发

相对于全收集而言，收集间隔较短

##### 全收集器

Full GC，指发生在老年代的GC，出现了Full GC一般会伴随着至少一次的Minor GC（老年代的对象大部分是Scavenge GC过程中从新生代进入老年代），比如：分配担保失败。Full GC的速度一般会比Scavenge GC慢10倍以上。当老年代内存不足或者显式调用System.gc()方法时，会触发Full GC。

当老年代或者持久代堆空间满了，会触发全收集操作

可以使用System.gc()方法来显式的启动全收集

全收集一般根据堆大小的不同，需要的时间不尽相同，但一般会比较长。

##### 垃圾回收器的常规匹配

serial

ParNew

Parallel Scavenge

Serial Old

Parallel Old

CMS

G1

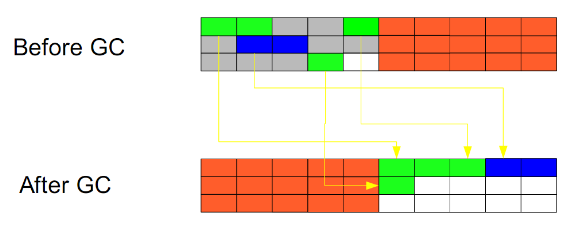
#### 常见垃圾回收算法

##### 引用计数（Reference Counting）

比较古老的回收算法。原理是此对象有一个引用，即增加一个计数，删除一个引用则减少一个计数。垃圾回收时，只用收集计数为0的对象。此算法最致命的是无法处理循环引用的问题。

##### 复制（Copying）

此算法把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。此算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不会出现“碎片”问题。当然，此算法的缺点也是很明显的，就是需要两倍内存空间。简图如下：



##### 标记-清除（Mark-Sweep）

此算法执行分两阶段。第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。此算法需要暂停整个应用，同时，会产生内存碎片。简图如下：



##### 标记-整理（Mark-Compact）

此算法结合了“标记-清除”和“复制”两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，把清除未标记对象并且把存活对象“压缩”到堆的其中一块，按顺序排放。此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。简图如下：

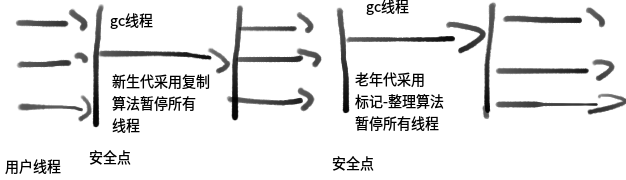


#### 分代垃圾收集器

##### 串行收集器（Serial）

Serial收集器是Hotspot运行在Client模式下的默认新生代收集器, 它的特点是：只用一个CPU（计算核心）/一条收集线程去完成GC工作, 且在进行垃圾收集时必须暂停其他所有的工作线程(“Stop The World” -后面简称STW)。可以使用-XX:+UseSerialGC打开。

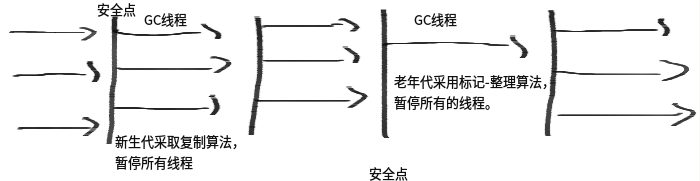
虽然是单线程收集, 但它却简单而高效, 在VM管理内存不大的情况下(收集几十M~一两百M的新生代), 停顿时间完全可以控制在几十毫秒~一百多毫秒内。



##### 并行收集器（ParNew）

ParNew收集器其实是前面Serial的多线程版本, 除使用多条线程进行GC外, 包括Serial可用的所有控制参数、收集算法、STW、对象分配规则、回收策略等都与Serial完全一样(也是VM启用CMS收集器-XX: +UseConcMarkSweepGC的默认新生代收集器)。

由于存在线程切换的开销, ParNew在单CPU的环境中比不上Serial, 且在通过超线程技术实现的两个CPU的环境中也不能100%保证能超越Serial. 但随着可用的CPU数量的增加, 收集效率肯定也会大大增加(ParNew收集线程数与CPU的数量相同, 因此在CPU数量过大的环境中, 可用-XX:ParallelGCThreads=<N>参数控制GC线程数)。



##### Parallel Scavenge收集器

与ParNew类似, Parallel Scavenge也是使用复制算法, 也是并行多线程收集器. 但与其他收集器关注尽可能缩短垃圾收集时间不同, Parallel Scavenge更关注系统吞吐量:

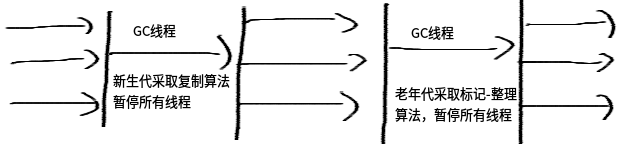
系统吞吐量=运行用户代码时间/(运行用户代码时间+垃圾收集时间)

停顿时间越短就越适用于用户交互的程序-良好的响应速度能提升用户的体验;而高吞吐量则适用于后台运算而不需要太多交互的任务-可以最高效率地利用CPU时间,尽快地完成程序的运算任务. Parallel Scavenge提供了如下参数设置系统吞吐量:

|  |  |
| --- | --- |
| Parallel Scavenge参数 | 描述 |
| -XX:MaxGCPauseMillis | (毫秒数) 收集器将尽力保证内存回收花费的时间不超过设定值, 但如果太小将会导致GC的频率增加. |
| -XX:GCTimeRatio | (整数:0 < GCTimeRatio < 100) 是垃圾收集时间占总时间的比率 |
| XX:+UseAdaptiveSizePolicy | 启用GC自适应的调节策略: 不再需要手工指定-Xmn、-XX:SurvivorRatio、-XX:PretenureSizeThreshold等细节参数, VM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最大的吞吐量 |

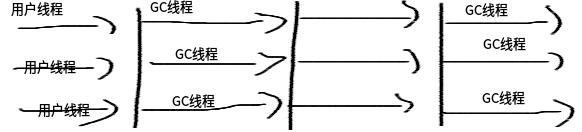
##### Serial Old收集器

Serial Old是Serial收集器的老年代版本, 同样是单线程收集器,使用“标记-整理”算法



##### Parallel Old收集器

Parallel Old是Parallel Scavenge收集器的老年代版本, 使用多线程和“标记－整理”算法, 吞吐量优先, 主要与Parallel Scavenge配合在注重吞吐量及CPU资源敏感系统内使用；



##### CMS收集器（Concurrent Mark Sweep）

CMS(Concurrent Mark Sweep)收集器是一款具有划时代意义的收集器, 一款真正意义上的并发收集器, 虽然现在已经有了理论意义上表现更好的G1收集器, 但现在主流互联网企业线上选用的仍是CMS(如Taobao、微店).

CMS是一种以获取最短回收停顿时间为目标的收集器(CMS又称多并发低暂停的收集器), 基于”标记-清除”算法实现, 整个GC过程分为以下4个步骤:

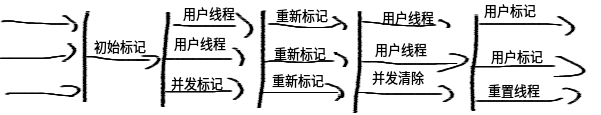
1. 初始标记(CMS initial mark)

2. 并发标记(CMS concurrent mark: GC Roots Tracing过程)

3. 重新标记(CMS remark)

4. 并发清除(CMS concurrent sweep: 已死对象将会就地释放, 注意:此处没有压缩)

其中1，3两个步骤(初始标记、重新标记)仍需STW. 但初始标记仅只标记一下GC Roots能直接关联到的对象, 速度很快; 而重新标记则是为了修正并发标记期间因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录, 虽然一般比初始标记阶段稍长, 但要远小于并发标记时间.



CMS特点：

1. CMS默认启动的回收线程数=(CPU数目+3)/4

当CPU数>4时, GC线程一般占用不超过25%的CPU资源, 但是当CPU数<=4时, GC线程可能就会过多的占用用户CPU资源, 从而导致应用程序变慢, 总吞吐量降低.

2.无法处理浮动垃圾, 可能出现Promotion Failure、Concurrent Mode Failure而导致另一次Full GC的产生: 浮动垃圾是指在CMS并发清理阶段用户线程运行而产生的新垃圾. 由于在GC阶段用户线程还需运行, 因此还需要预留足够的内存空间给用户线程使用, 导致CMS不能像其他收集器那样等到老年代几乎填满了再进行收集. 因此CMS提供了-XX:CMSInitiatingOccupancyFraction参数来设置GC的触发百分比(以及-XX:+UseCMSInitiatingOccupancyOnly来启用该触发百分比), 当老年代的使用空间超过该比例后CMS就会被触发(JDK 1.6之后默认92%). 但当CMS运行期间预留的内存无法满足程序需要, 就会出现上述Promotion Failure等失败, 这时VM将启动后备预案: 临时启用Serial Old收集器来重新执行Full GC(CMS通常配合大内存使用, 一旦大内存转入串行的Serial GC, 那停顿的时间就是大家都不愿看到的了).

3.最后, 由于CMS采用”标记-清除”算法实现, 可能会产生大量内存碎片. 内存碎片过多可能会导致无法分配大对象而提前触发Full GC. 因此CMS提供了-XX:+UseCMSCompactAtFullCollection开关参数, 用于在Full GC后再执行一个碎片整理过程. 但内存整理是无法并发的, 内存碎片问题虽然没有了, 但停顿时间也因此变长了, 因此CMS还提供了另外一个参数-XX:CMSFullGCsBeforeCompaction用于设置在执行N次不进行内存整理的Full GC后, 跟着来一次带整理的(默认为0: 每次进入Full GC时都进行碎片整理).

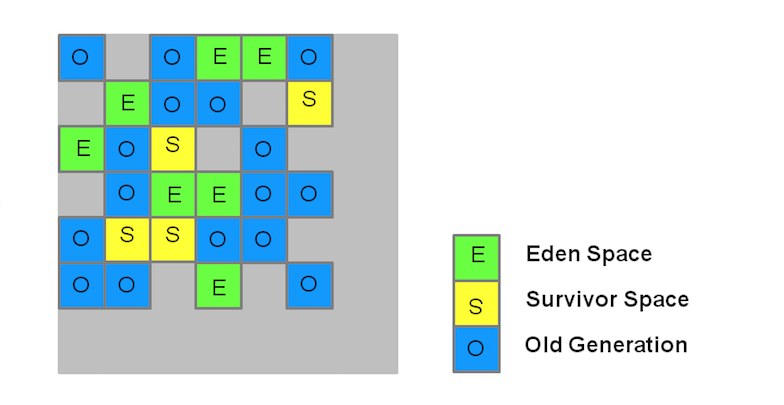
##### 分区收集- G1收集器

G1(Garbage-First)是一款面向服务端应用的收集器, 主要目标用于配备多颗CPU的服务器治理大内存.

- G1 is planned as the long term replacement for the Concurrent Mark-Sweep Collector (CMS).

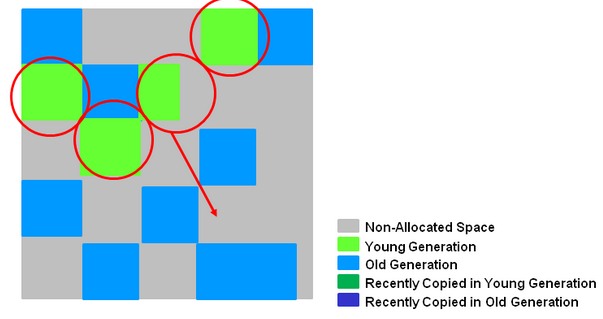
-XX:+UseG1GC启用G1收集器.

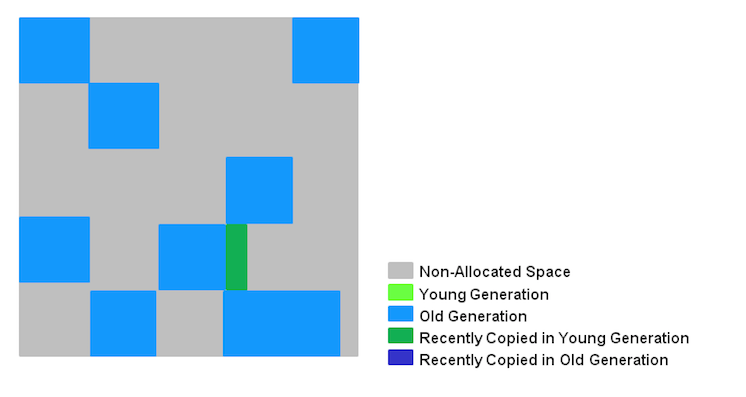
与其他基于分代的收集器不同, G1将整个Java堆划分为多个大小相等的独立区域(Region), 虽然还保留有新生代和老年代的概念, 但新生代和老年代不再是物理隔离的了, 它们都是一部分Region(不需要连续)的集合.如：



每块区域既有可能属于O区、也有可能是Y区, 因此不需要一次就对整个老年代/新生代回收. 而是当线程并发寻找可回收的对象时, 有些区块包含可回收的对象要比其他区块多很多. 虽然在清理这些区块时G1仍然需要暂停应用线程, 但可以用相对较少的时间优先回收垃圾较多的Region. 这种方式保证了G1可以在有限的时间内获取尽可能高的收集效率.

G1的新生代收集跟ParNew类似: 存活的对象被转移到一个/多个Survivor Regions. 如果存活时间达到阀值, 这部分对象就会被提升到老年代.如图：





其特定是：

一整块堆内存被分为多个Regions.

存活对象被拷贝到新的Survivor区或老年代.

年轻代内存由一组不连续的heap区组成, 这种方法使得可以动态调整各代区域尺寸.

Young GC会有STW事件, 进行时所有应用程序线程都会被暂停.

多线程并发GC.

G1老年代GC特点如下:

并发标记阶段

1 在与应用程序并发执行的过程中会计算活跃度信息.

2 这些活跃度信息标识出那些regions最适合在STW期间回收(which regions will be best to reclaim during an evacuation pause).

3 不像CMS有清理阶段.

再次标记阶段

1 使用Snapshot-at-the-Beginning(SATB)算法比CMS快得多.

2 空region直接被回收.

拷贝/清理阶段(Copying/Cleanup Phase)

1 年轻代与老年代同时回收.

2 老年代内存回收会基于他的活跃度信息.

### JVM优化

#### JDK常用JVM优化相关命令

|  |  |  |
| --- | --- | --- |
| bin | 描述 | 功能 |
| jps | 打印Hotspot VM进程 | VMID、JVM参数、main()函数参数、主类名/Jar路径 |
| jstat | 查看Hotspot VM 运行时信息 | 类加载、内存、GC[可分代查看]、JIT编译  命令格式：jstat -gc 10340 250 20 |
| jinfo | 查看和修改虚拟机各项配置 | -flag name=value |
| jmap | heapdump: 生成VM堆转储快照、查询finalize执行队列、Java堆和永久代详细信息 | jmap -dump:live,format=b,file=heap.bin [VMID] |
| jstack | 查看VM当前时刻的线程快照: 当前VM内每一条线程正在执行的方法堆栈集合 | Thread.getAllStackTraces()提供了类似的功能 |
| javap | 查看经javac之后产生的JVM字节码代码 | 自动解析.class文件, 避免了去理解class文件格式以及手动解析class文件内容 |
| jcmd | 一个多功能工具, 可以用来导出堆, 查看Java进程、导出线程信息、 执行GC、查看性能相关数据等 | 几乎集合了jps、jstat、jinfo、jmap、jstack所有功能 |
| jconsole | 基于JMX的可视化监视、管理工具 | 可以查看内存、线程、类、CPU信息, 以及对JMX MBean进行管理 |
| jvisualvm | JDK中最强大运行监视和故障处理工具 | 可以监控内存泄露、跟踪垃圾回收、执行时内存分析、CPU分析、线程分析… |

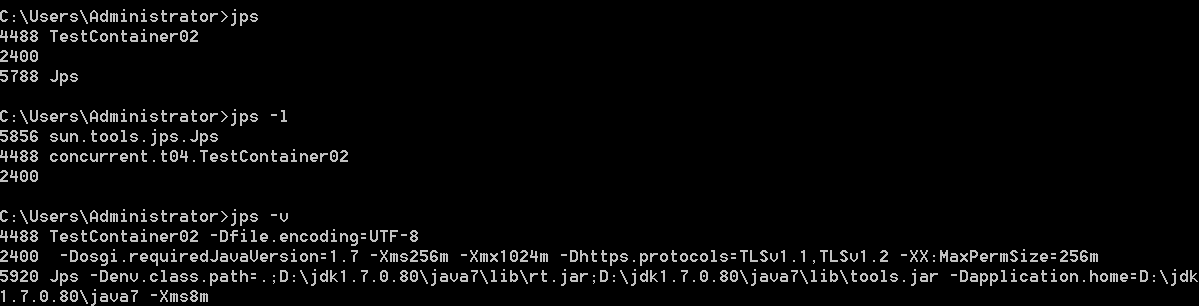
##### jps

jps - l

显示线程id和执行线程的主类名

jps -v

显示线程id和执行线程的主类名和JVM配置信息



##### jstat

jstat -参数 线程id 执行时间（单位毫秒） 执行次数

jstat -gc 4488 30 10

|  |
| --- |
| S0C S1C S0U S1U EC EU  10752.0 10752.0 0.0 0.0 65536.0 18351.3  OC OU MC MU YGC YGCT FGC FGCT GCT  173568.0 0.0 21504.0 2659.6 0 0.000 0 0.000 0.000 |

SXC - survivor 初始空间大小，单位字节。

SXU - survivor 使用空间大小， 单位字节。

EC - eden 初始空间大小

EU - eden 使用空间大小

OC - old 初始空间大小

OU - old 使用空间大小

MC - permanent 初始空间大小

MU - permanent 使用空间大小

CCSC - 压缩类空间， copy算法的备份空间。

CCSU - 压缩类空间， copy算法的备份空间。

YGC - youngGC 收集次数

YGCT - youngGC收集使用时长， 单位秒

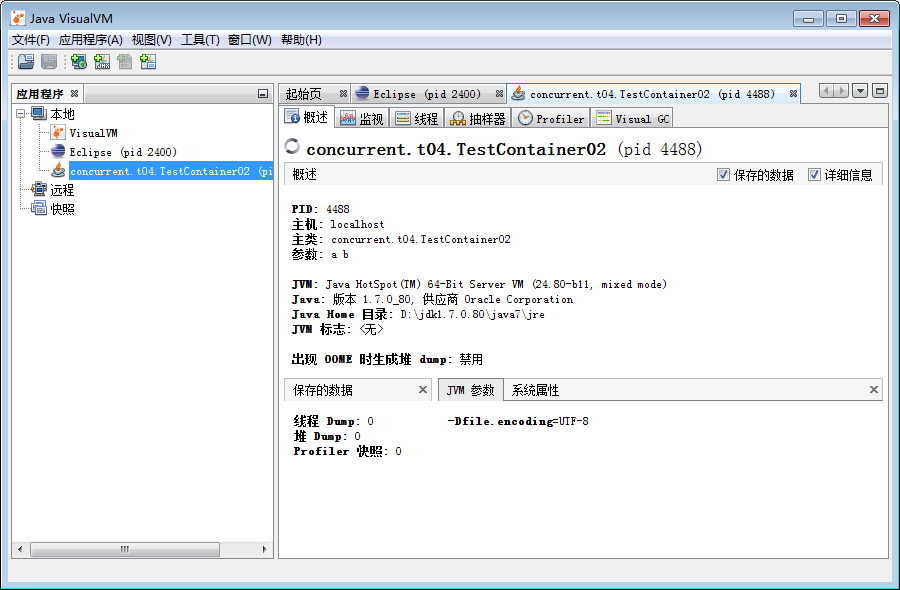
FGC - fullGC收集次数

FGCT - fullGC收集使用时长

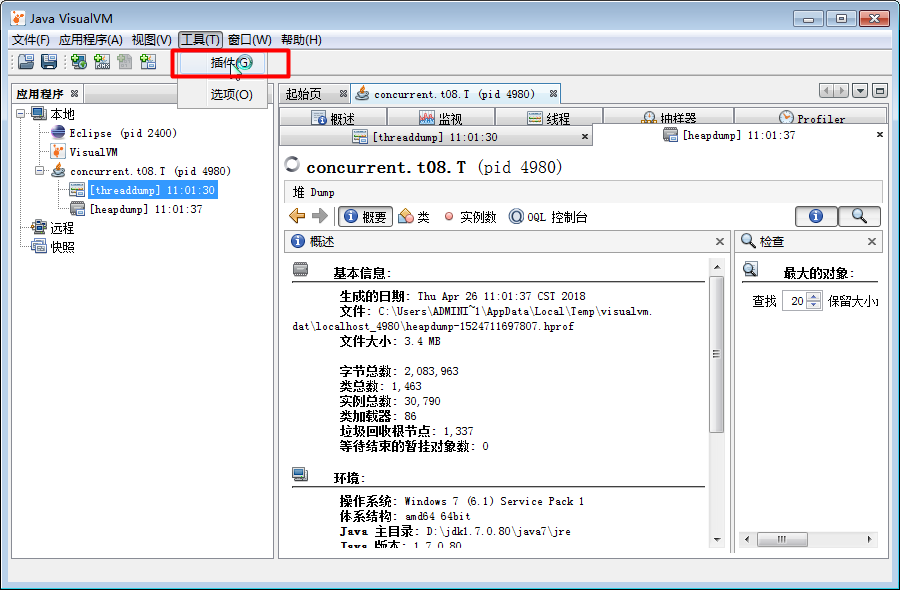
GCT - 总计收集使用总时长 YGCT+FGCT

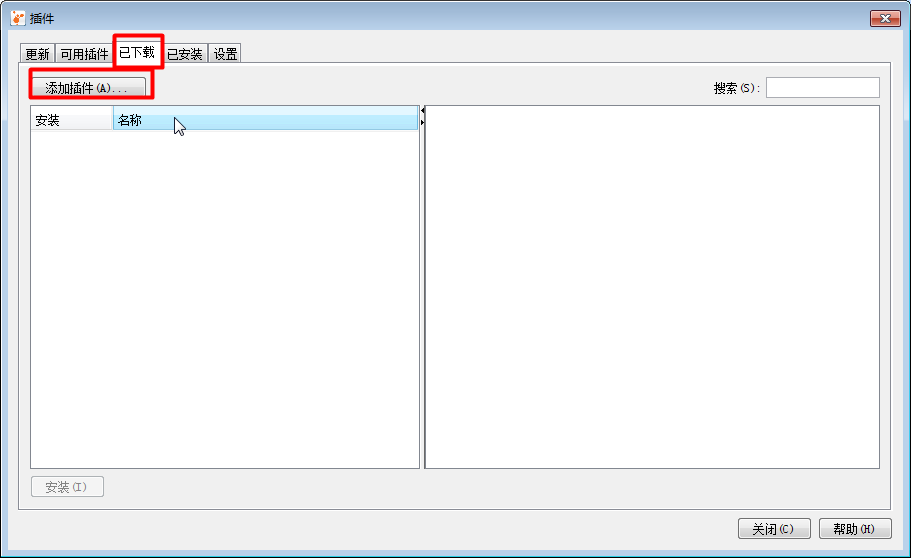
##### jvisualvm

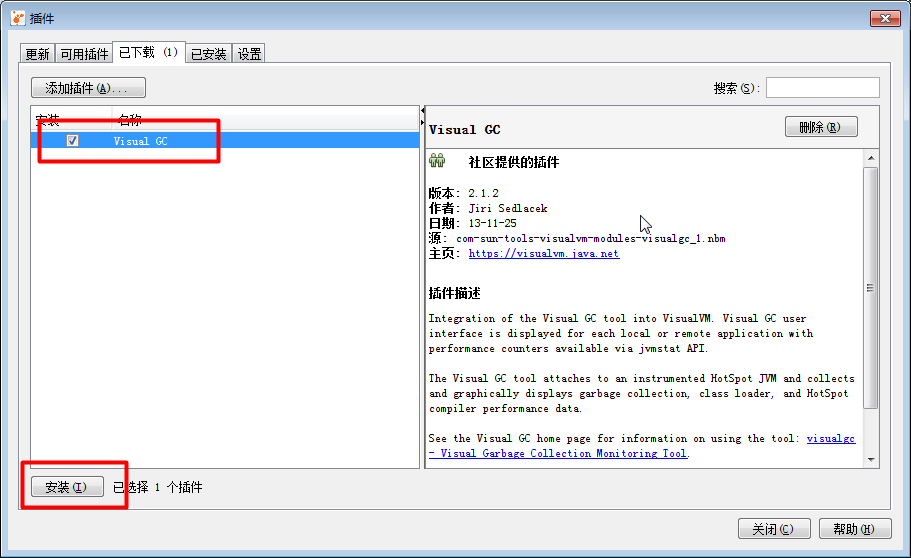
一个JDK内置的图形化VM监视管理工具

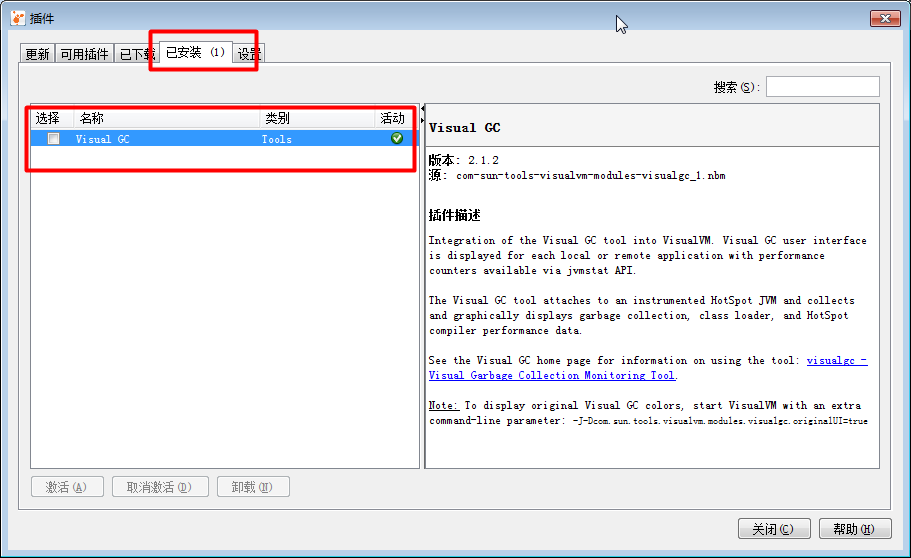


##### visualgc插件

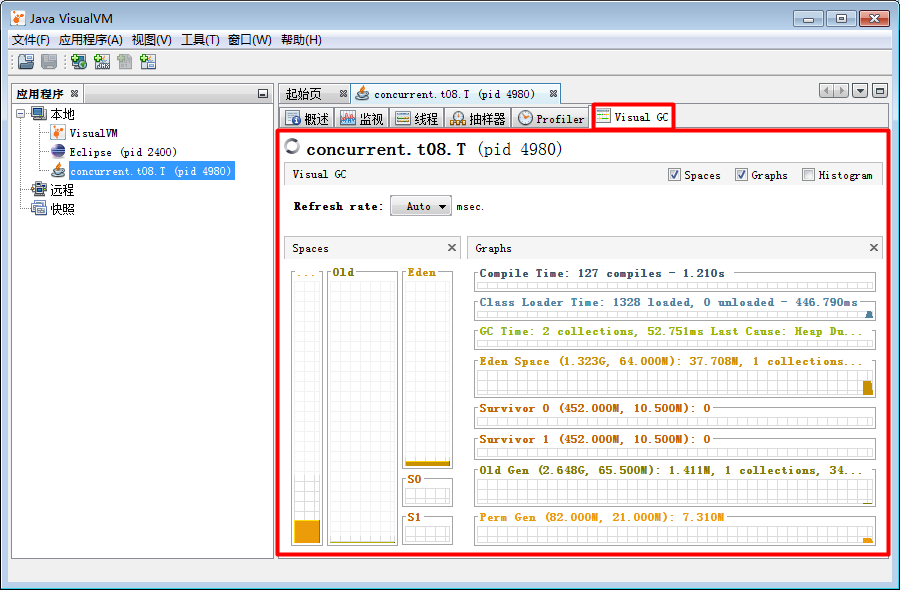








重启jvisualvm工具



#### JVM常见参数

配置方式：java [options] MainClass [arguments]

options - JVM启动参数。 配置多个参数的时候，参数之间使用空格分隔。

参数命名： 常见为 -参数名

参数赋值： 常见为 -参数名=参数值 | -参数名:参数值

##### 内存设置

-Xms:初始堆大小，JVM启动的时候，给定堆空间大小。

-Xmx:最大堆大小，JVM运行过程中，如果初始堆空间不足的时候，最大可以扩展到多少。

-Xmn：设置年轻代大小。整个堆大小=年轻代大小+年老代大小+元空间大小。元空间一般固定大小为64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。

-Xss： 设置每个线程的Java栈大小。JDK5.0以后每个线程Java栈大小为1M，以前每个线程堆栈大小为256K。根据应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。

-XX:NewSize=n:设置年轻代大小

-XX:NewRatio=n:设置年轻代和年老代的比值。如:为3，表示年轻代与年老代比值为1：3，年轻代占整个年轻代+年老代和的1/4

-XX:SurvivorRatio=n:年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如：3，表示Eden：Survivor=3：2，一个Survivor区占整个年轻代的1/5

-XX:MaxPermSize=n:设置持久代大小。不推荐手工设置。

-XX:MaxTenuringThreshold：设置垃圾最大年龄。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概率。

##### 内存设置经验分享

JVM中最大堆大小有三方面限制：相关操作系统的数据模型（32-bt还是64-bit）限制；系统的可用虚拟内存限制；系统的可用物理内存限制。32位系统 下，一般限制在1.5G~2G；64为操作系统对内存无限制。

Tomcat配置方式： 编写catalina.bat|catalina.sh，增加JAVA\_OPTS参数设置。windows和linux配置方式不同。windows - set "JAVA\_OPTS=%JAVA\_OPTS% 自定义参数"；linux - JAVA\_OPTS="$JAVA\_OPTS 自定义参数"

常见设置：

-Xmx3550m -Xms3550m -Xmn2g -Xss128k 适合开发过程的测试应用。要求物理内存大于4G。

-Xmx3550m -Xms3550m -Xss128k -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxPermSize=160m -XX:MaxTenuringThreshold=0 适合高并发本地测试使用。且大数据对象相对较多（如IO流）

环境： 16G物理内存，高并发服务，重量级对象中等（线程池，连接池等），常用对象比例为40%（运行过程中产生的对象40%是生命周期较长的）

-Xmx10G -Xms10G -Xss1M -XX:NewRatio=3 -XX:SurvivorRatio=4 -XX:MaxPermSize=2048m -XX:MaxTenuringThreshold=5

java -Xmx10G -Xms10G -Xss1M -XX:NewRatio=3 -XX:SurvivorRatio=4 -XX:MaxPermSize=2048m -XX:MaxTenuringThreshold=5 主类名 类型启动参数

##### 收集器设置

收集器配置的时候，次收集器和全收集器必须匹配。具体匹配规则参考[3.](#_垃圾回收器的常规匹配)[1.3](#_垃圾回收器的常规匹配)

-XX:+UseSerialGC:设置串行收集器，年轻带收集器， 次收集器

-XX:+UseParallelGC:设置并行收集器

-XX:+UseParNewGC:设置年轻代为并行收集。可与CMS收集同时使用。JDK5.0以上，JVM会根据系统配置自行设置，所以无需再设置此值。

-XX:+UseParallelOldGC:设置并行年老代收集器，JDK6.0支持对年老代并行收集。

-XX:+UseConcMarkSweepGC:设置年老代并发收集器，测试中配置这个以后，-XX:NewRatio的配置失效，原因不明。所以，此时年轻代大小最好用-Xmn设置。

-XX:+UseG1GC:设置G1收集器

##### 垃圾回收统计信息

类似日志的配置信息。会有控制台相关信息输出。 商业项目上线的时候，不允许使用。一定使用loggc

-XX:+PrintGC

-XX:+Printetails

-XX:+PrintGCTimeStamps

-Xloggc:filename

##### 并行收集器设置

-XX:ParallelGCThreads=n:设置并行收集器收集时最大线程数使用的CPU数。并行收集线程数。

-XX:MaxGCPauseMillis=n:设置并行收集最大暂停时间，单位毫秒。可以减少STW时间。

-XX:GCTimeRatio=n:设置垃圾回收时间占程序运行时间的百分比。公式为1/(1+n)并发收集器设置

-XX:+CMSIncrementalMode:设置为增量模式。适用于单CPU情况。

-XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。

-XX:CMSFullGCsBeforeCompaction=n：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次GC以后对内存空间进行压缩、整理。

-XX:+UseCMSCompactAtFullCollection：打开对年老代的压缩。可能会影响性能，但是可以消除碎片

##### 收集器设置经验分享

关于收集器的选择JVM给了三种选择：串行收集器、并行收集器、并发收集器，但是串行收集器只适用于小数据量的情况，所以这里的选择主要针对并行收集器和并发收集器。默认情况下，JDK5.0以前都是使用串行收集器，如果想使用其他收集器需要在启动时加入相应参数。JDK5.0以后，JVM会根据当前系统配置进行判断。

常见配置：

并行收集器主要以到达一定的吞吐量为目标，适用于科学计算和后台处理等。

-Xmx3800m -Xms3800m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20

使用ParallelGC作为并行收集器， GC线程为20（CPU核心数>=20时），内存问题根据硬件配置具体提供。建议使用物理内存的80%左右作为JVM内存容量。

-Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20 -XX:+UseParallelOldGC

指定老年代收集器，在JDK5.0之后的版本，ParallelGC对应的全收集器就是ParallelOldGC。可以忽略

-Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:MaxGCPauseMillis=100

指定GC时最大暂停时间。单位是毫秒。每次GC最长使用100毫秒。可以尽可能提高工作线程的执行资源。

-Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:MaxGCPauseMillis=100 -XX:+UseAdaptiveSizePolicy

UseAdaptiveSizePolicy是提高年轻代GC效率的配置。次收集器执行效率。

并发收集器主要是保证系统的响应时间，减少垃圾收集时的停顿时间。适用于应用服务器、电信领域、互联网领域等。

-Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC

指定年轻代收集器为ParNew，年老代收集器ConcurrentMarkSweep，并发GC线程数为20（CPU核心>=20），并发GC的线程数建议使用（CPU核心数+3）/4或CPU核心数【推荐使用】。

-Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseConcMarkSweepGC -XX:CMSFullGCsBeforeCompaction=5 -XX:+UseCMSCompactAtFullCollection

CMSFullGCsBeforeCompaction=5执行5次GC后，运行一次内存的整理。

UseCMSCompactAtFullCollection执行老年代内存整理。可以避免内存碎片，提高GC过程中的效率，减少停顿时间。如果内存容量足够，不推荐开启。效率较低。

##### 简单总结

年轻代大小选择

响应时间优先的应用：尽可能设大，直到接近系统的最低响应时间限制（根据实际情况选择）。在此种情况下，年轻代收集发生的频率也是最小的。同时，减少到达年老代的对象。

吞吐量优先的应用：尽可能的设置大，可能到达Gbit的程度。因为对响应时间没有要求，垃圾收集可以并行进行，一般适合8CPU以上的应用。

年老代大小选择

响应时间优先的应用：年老代使用并发收集器，所以其大小需要小心设置，一般要考虑并发会话率和会话持续时间等一些参数。如果堆设置小了，可以会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案，一般需要参考以下数据获得：

并发垃圾收集信息

持久代并发收集次数

传统GC信息

花在年轻代和年老代回收上的时间比例

减少年轻代和年老代花费的时间，一般会提高应用的效率

吞吐量优先的应用：一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而年老代存放长期存活对象。

较小堆引起的碎片问题，因为年老代的并发收集器使用标记、清除算法，所以不会对堆进行压缩。当收集器回收时，他会把相邻的空间进行合并，这样可以分配给较大的对象。但是，当堆空间较小时，运行一段时间以后，就会出现“碎片”，如果并发收集器找不到足够的空间，那么并发收集器将会停止，然后使用传统的标记、整理方式进行回收。如果出现“碎片”，可能需要进行如下配置：

-XX:+UseCMSCompactAtFullCollection：使用并发收集器时，开启对年老代的压缩。

-XX:CMSFullGCsBeforeCompaction=0：上面配置开启的情况下，这里设置多少次Full GC后，对年老代进行压缩

##### 测试代码

|  |
| --- |
| package jvm;  import java.io.IOException;  import java.lang.management.GarbageCollectorMXBean;  import java.lang.management.ManagementFactory;  import java.util.List;  public class Test {  public static void main(String[] args) {  List<GarbageCollectorMXBean> l =  ManagementFactory.*getGarbageCollectorMXBeans*();  for(GarbageCollectorMXBean b : l) {  System.*out*.println(b.getName());  }  try {  System.*in*.read();  } catch (IOException e) {  e.printStackTrace();  }  }  } |

# Tomcat深度优化

## Tomcat优化重要性

相信Web开发的小伙伴的日常开发基本离不开Tomcat，Tomcat作为一个免费的开放源代码的Web 应用服务器，它的性能已经相当出色了，但是有些时候想要发挥tomcat最佳的性能还是需要一定的优化配置工作。本课程就简单的总结一下Tomcat的优化，以便以后开发工作调优等情况下的知识储备。

tomcat中的脚本文件：$CATALINA\_HOME/bin/\*.sh [bat]

setclasspath.sh - 设置若干环境变量和类路径环境变量

catalina.sh - tomcat中的核心脚本，真正启动、停止tomcat的脚本文件

startup.sh - 简化的启动脚本文件。

shutdown.sh - 简化的停止脚本文件。

tomcat中的配置文件：$CATALINA\_HOME/conf/\*.xml

context.xml - 配置tomcat上下文环境的。如：可以配置一个连接池，在java代码中通过JNDI来获取连接池访问数据库。

server.xml

tomcat-users.xml - 配置tomcat提供的默认管理应用登录用户及权限的。配置好用户和权限后，可以通过Server Status查看Tomcat服务器状态数据。

## Tomcat优化

### 内存优化

通过对JVM配置的优化，提升Tomcat性能。

#### 常用配置简介

1、-Xmx1024m  设置JVM最大可用内存为1024MB

2、-Xms1024m  设置JVM最小内存为1024m。此值可以设置与-Xmx相同，以避免每次垃圾回收完成后JVM重新分配内存。

3、-XX:NewSize  设置年轻代大小

4、XX:MaxNewSize 设置最大的年轻代大小

5、-XX:PermSize  设置永久代大小。Java8以后版本无效

6、-XX:MaxPermSize 设置最大永久代大小。 Java8以后版本无效

7、-XX:NewRatio=4:设置年轻代（包括Eden和两个Survivor区）与终身代的比值（除去永久代）。设置为4，则年轻代与终身代所占比值为1：4，年轻代占整个堆栈的1/5

8、-XX:MaxTenuringThreshold=0：设置垃圾最大年龄，默认为：15。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概论。

9、-XX:+DisableExplicitGC这个将会忽略手动调用GC的代码使得 System.gc()的调用就会变成一个空调用，完全不会触发任何GC

#### windows下修改:

在bin/catalina.bat文件中增加下述配置

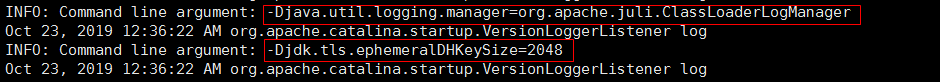
set JAVA\_OPTS=-Dfile.encoding=UTF-8 -server -Xms256m -Xmx512m -XX:NewSize=128m -XX:MaxNewSize=256m -XX:PermSize=64m -XX:MaxPermSize=64m -XX:MaxTenuringThreshold=10 -XX:NewRatio=2 -XX:+DisableExplicitGC

#### Linux下修改：

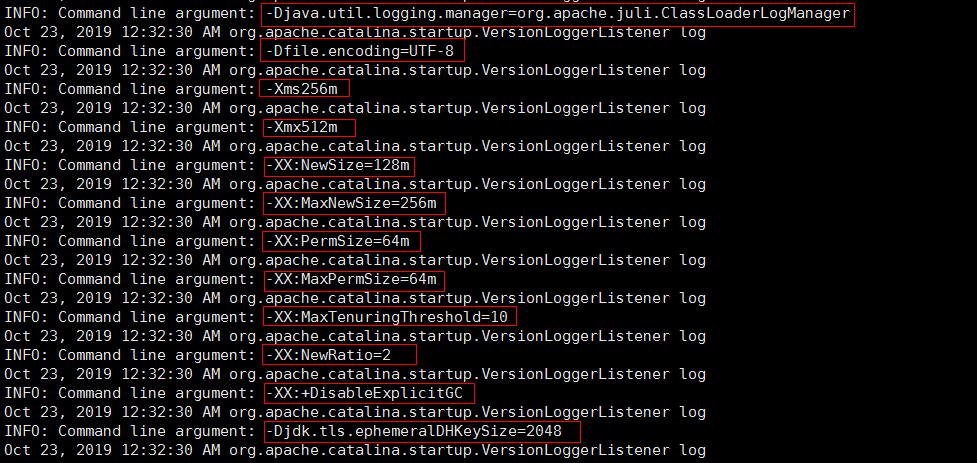
在bin/catalina.sh文件中增加下述配置

JAVA\_OPTS="-Dfile.encoding=UTF-8 -server -Xms256m -Xmx512m -XX:NewSize=128m -XX:MaxNewSize=256m -XX:PermSize=64m -XX:MaxPermSize=64m -XX:MaxTenuringThreshold=10 -XX:NewRatio=2 -XX:+DisableExplicitGC"

未设置启动参数的Tomcat启动日志：



设置启动参数的Tomcat启动日志：



### Connector运行模式

Tomcat中的Connector有三种执行模式，分别是BIO、NIO和APR。

#### 模式简介

BIO：同步阻塞的链接模式。默认的模式,性能非常低下,没有经过任何优化处理和支持.

NIO：同步非阻塞的链接模式。Java nio是一个基于缓冲区、并能提供非阻塞I/O操作的Java API，因此nio也被看成是non-blocking I/O的缩写。它拥有比传统I/O操作(bio)更好的并发运行性能。

APR：apr是从操作系统级别解决异步IO问题，大幅度提高服务器的并发处理性能，也是Tomcat生产环境运行的首选方式，而apr的本质就是使用jni技术调用操作系统底层的IO接口，所以需要提前安装所需要的依赖。

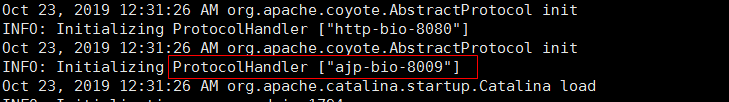
#### 关闭AJP

AJP是为 Tomcat 与 HTTP 服务器之间通信而定制的协议，能提供较高的通信速度和效率。如果tomcat前端放的是apache的时候，会使用到AJP这个连接器。若tomcat未与apache配合使用，因此不使用此连接器，因此需要注销掉该连接器。

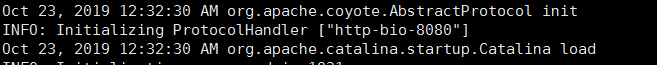
修改位置在conf/server.xml配置文件。



未关闭AJP的启动日志内容：

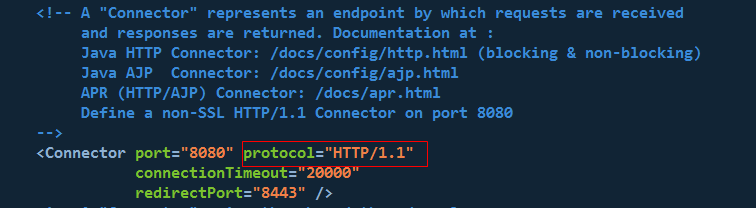


关闭AJP的启动日志内容：

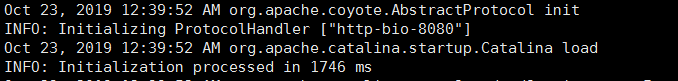


#### BIO模式

这是Tomcat的默认模式，是同步阻塞的链接模式。其配置位置在conf/server.xml配置文件中的Connector标签。

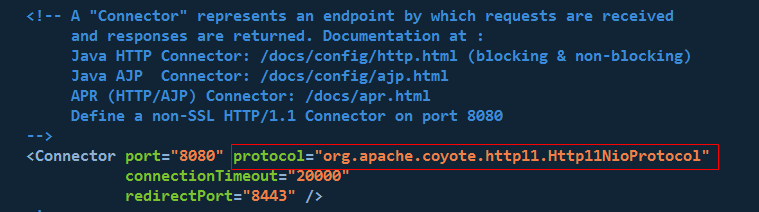


Tomcat启动日志：

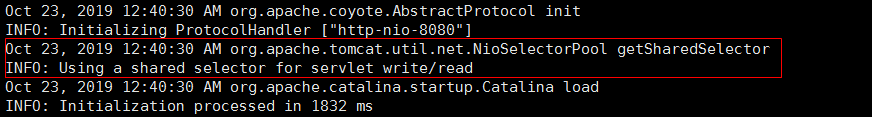


#### NIO模式

修改Tomcat链接模式为同步非阻塞，这种链接方式可以大大提高Tomcat与客户端交互的响应效率。其配置位置在conf/server.xml配置文件中Connector标签的protocol属性。将protocol属性的值修改为：org.apache.coyote.http11.Http11NioProtocol



Tomcat启动日志：



#### APR模式

##### 安装APR需要的依赖

openssl - 依赖开源ssl加密协议 devel - 完整包

apr - 依赖的apr系统底层库

expat - 扩展依赖，也是访问系统底层的。

yum -y install openssl-devel

yum -y install apr-devel

yum -y install expat-devel

##### 升级openssl版本

因现在版本的Tomcat-native需要openssl版本高于1.0.2。而自动安装的openssl版本为1.0.1。所以需要升级openssl版本，具体如下：

tar -xf openssl-1.0.2g.tar.gz

cd openssl-1.0.2g

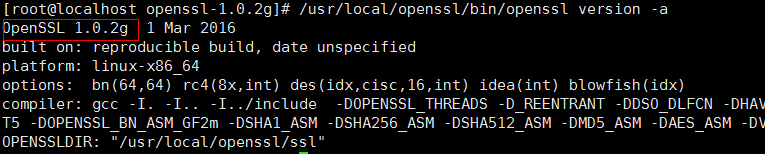
./config --prefix=/usr/local/openssl -fPIC no-gost no-shared no-zlib

make depend

make install

测试版本：

/usr/local/openssl/bin/openssl version -a



##### 安装APR组件

tar -zxf apr-1.7.0.tar.gz

cd apr-1.7.0

./configure --prefix=/usr/local/apr

make

make install

tar -zxf apr-util-1.6.1.tar.gz

cd apr-util-1.6.1

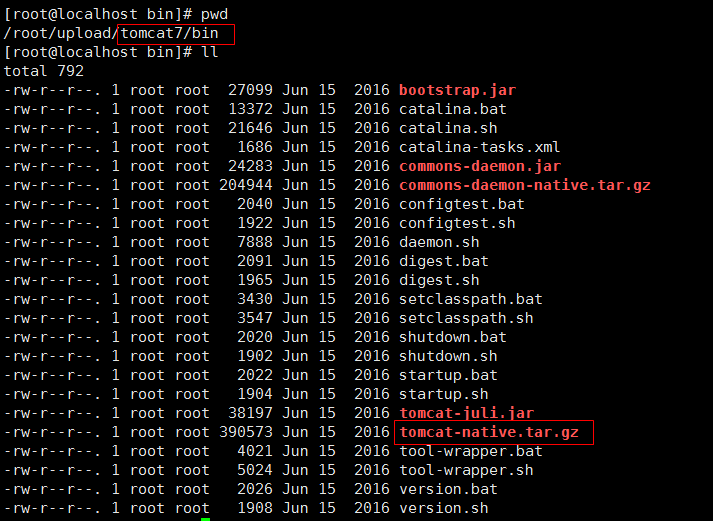
./configure --prefix=/usr/local/apr-util --with-apr=/usr/local/apr

make

make install

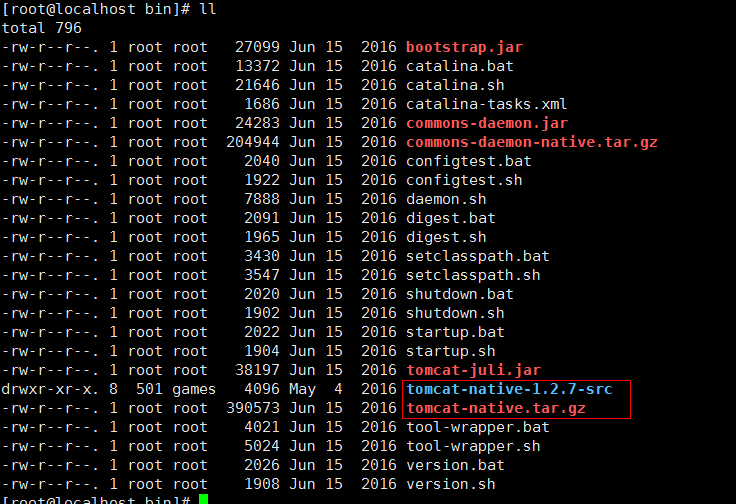
##### 安装Tomcat的native组件

native可以看成是tomcat和apr交互的中间环节，tomcat的bin目录中存在native压缩目录，具体如下：



tar -zxf tomcat-native.tar.gz

解压后如下：



进入tomcat-native-1.2.7-src/native目录，并进行配置安装。

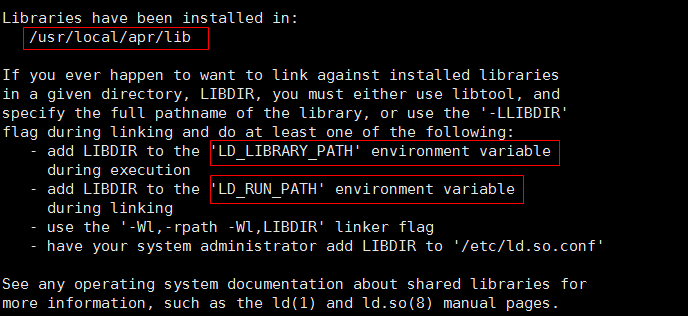
cd tomcat-native-1.2.7-src/native

./configure --with-apr=/usr/local/apr --with-java-home=/usr/local/java --with-ssl=/usr/local/openssl

make

make install

安装成功后，会有环境变量修改的提示信息，具体如下：



##### 修改环境变量

vi /etc/profile

export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/usr/local/apr/lib

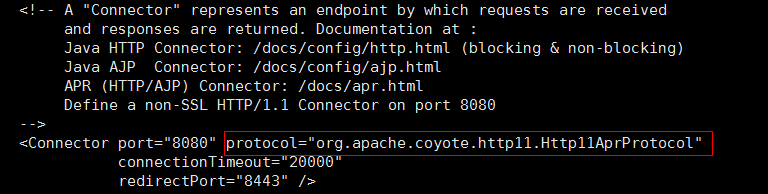
export LD\_RUN\_PATH=$LD\_RUN\_PATH:/usr/local/apr/lib

刷新环境变量

. /etc/profile

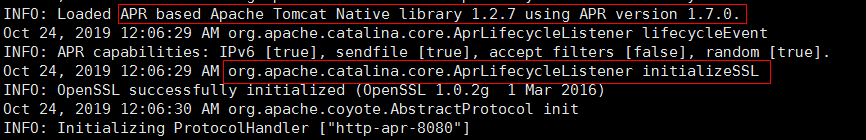
##### 修改Tomcat配置

修改tomcat/conf/server.xml配置文件，将Connector标签中的protocol属性值修改为：org.apache.coyote.http11.Http11AprProtocol。具体如下：



##### 启动Tomcat测试APR模式

bin/startup.sh & tailf logs/catalina.out



### autoDeploy配置

当设置为true时，tomcat会在启动时，定时的去检查是否有新的项目更新。默认为true。因此我们需要将这个参数设置为false。这样可以避免Tomcat将无谓的性能消耗在检查项目更新。



修改后：



### 线程池优化

Tomcat中默认配置中，启动的后会初始化一个最大容量为200的线程池，这个容量的线程池在商业项目中是明显不够的，所以对线程池的优化就显得极其重要。

设置线程池的方式为，修改conf/server.xml配置文件，具体配置方式如下：

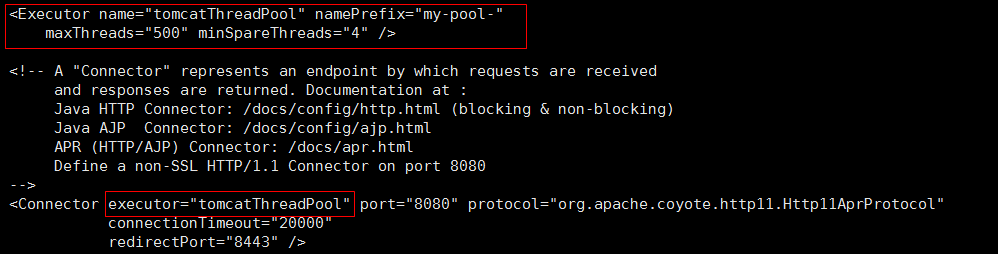
新增线程池标签：

<Executor name="tomcatThreadPool" namePrefix="my-pool-"

maxThreads="500" minSpareThreads="100" />

新增Connector标签的executor属性：

<Connector executor="tomcatThreadPool" ... />



线程池Executor标签的常用属性简介：

maxConnections 服务器所能接受最大的请求和处理的连接数，NIO和NIO2默认为10000，APR默认是8192。

maxThreads 同一时间点上处理线程的最大数量。 默认为200。

acceptCount 当所有可能的请求处理线程都在使用时，传入连接请求的最大队列长度。 队列已满时收到的任何请求都将被拒绝。 默认值为100。

minSpareThreads 最小空闲线程数，默认为10，不易过小。

### 禁用DNS

当web应用程序想要记录客户端的信息时，它也会记录客户端的IP地址或者通过域名服务器查找机器名再转换为IP地址。DNS查询需要占用网络，并且包括可能从很多很远的服务器或者不起作用的服务器上去获取对应的IP的过程，这样会消耗一定的时间。

修改server.xml文件中的Connector元素，修改属性enableLookups参数值: enableLookups="false"。

如果为true，则可以通过调用request.getRemoteHost()进行DNS查询来得到远程客户端的实际主机名，若为false则不进行DNS查询，而是返回其ip地址。

# MyBatis回顾

建议自学内容：Interceptor

是MyBatis提供的一个拦截器接口。为mapper配置文件中的sql做额外定义。如：pageHelper就是用Interceptor做的。

Interceptor像做好，需要配合plugin实现。就是将拦截器插件化。

Interceptor接口中的核心方法是intercept方法，参数是Invocation。方法就是拦截。方法的返回值，就是拦截的SqlSession数据库访问操作的返回值。如：拦截的是查询，返回的Object就是查询结果；拦截的是更新操作，返回的Object就是更新影响的行数。参数Invocation就是执行器，其中封装了SqlSession本次访问数据库要执行的SQL语句。

## 配置

### XML文件特性

良构：结构良好的XML文件。文件必须有唯一的根标签，且内部子标签结构正确。良构是XML编写的第一要素。不符合，无法正常解析。

<?xml version="1.0" encoding="UTF-8" ?>

<root><a></a><b><c></c></b><d/></root>

合法：在良构基础上，符合某一个约束限定的XML编写方式。限定方式可以是DTD或Schema（XSD）。

如果是DTD约束，在XML中编写方式是：

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE 根标签名 SYSTEM|PUBLIC "name" "uri">

SYSTEM - dtd文件没有经过w3c验证的，没有在广域网发布的dtd约束。

PUBLIC - dtd文件经过w3c验证，在广域网发布过。

name - 是给dtd约束定义一个名字。名字就是一个唯一的字符串。

uri - 就是在广域网发布的url地址。

只要XML编写良构，且符合dtd约束，就称为合法。

如果是Schema约束，在XML中的编写方式是：

<?xml version="1.0" encoding="UTF-8"?>

<根标签 xxx:ns="" xxx:ns="" >

</根标签>

xxx:ns - 命名空间（name space），代表在一个命名空间下的标签。默认命名空间可以直接定义标签并编写，如：spring中的bean标签。其他命名空间，需要通过命名空间名称:标签名称定义并编写， 如：spring中的context:placeholder，aop:advisor， tx:advice。

只要XML编写良构，且符合Schema约束，就称为合法。

### DTD解读

<!ELEMENT 标签名 (当前标签内可定义的内容 - xxx)> - 标签， dtd文档中的第一个，代表xml中的根标签。

xxx可以定义的内容有：

1. 子标签 - (标签名?,标签名\*,标签名+,标签名,(标签名|标签名)+)?|+|\*。 其中逗号","代表子标签的定义顺序；问号"?"代表子标签可以定义0或1个；星号"\*"代表子标签可以定义0或n个；加号"+"代表子标签可以定义1或n个；标签名后不加数量符号，代表只能定义唯一一个，且必须定义，括号代表优先级；竖线"|"代表选择。
2. 当前空标签 - EMPTY，代表当前标签是一个空标签，开始标签和结束标签之间不能有任何信息。如：<a></a>, <b/>
3. 标签体 - PCDATA，代表当前标签没有子标签，但是可以定义字符类型的标签体，如：spring中的value标签<value>abc</value>,<documentation></documentation><description>

<!ATTLIST 标签名 属性名 属性特征 属性名 属性特征> - 标签的属性列表

属性特征：都是属性的可写内容格式 属性的必要特征。

1. CDATA - 字符类型，属性值不能包含空格、回车、换行、制表符。
2. a|b|c - 枚举，代表当前属性的可选值有哪些
3. #IMPLIED - 可选属性
4. #REQUIRED - 必要属性
5. default "数据" - 代表当前属性默认值是什么。
6. NMTOKEN - 字符类型，可以包含空格。代表多个字符串。

### config

environments：配置环境，就是连接什么数据库，用的用户名和密码以及驱动是什么。必要属性default，代表默认使用配置的多个环境中的哪一个。default属性的值，必须是子标签environment标签的id属性值。

environment：就是配置具体的环境。必要属性id，定义这个具体环境的唯一标识。

transactionManager：环境中使用的事务管理器是什么类型的。必要属性type，赋值内容为JDBC，代表使用JDBC的Connection.commit|rollback实现事务管理。

dataSource：必要属性type，代表数据源的类型。可选值为POOLED|UNPOOLED。POOLED代表池化，提前初始化连接池。UNPOOLED代表非池化，每次通过SqlSessionFactory.openSession获取SqlSession的时候，创建连接。

### mapper

mapper - 可选属性namespace，必须填写。否则抛出异常。

insert - 定义insert语法的。可以定义所有的写操作SQL。常用的属性：id,parameterType

update - 定义update语法的。可以定义所有的写操作SQL。所有的写操作都使用update标签。常用的属性：id,parameterType

delete - 定义delete语法的。可以定义所有的写操作SQL。常用的属性：id,parameterType

select - 查询语法定义select。 常用的属性：parameterType,resultType,resultMap,id。id属性是唯一标识；parameterType属性是参数类型，不写由MyBatis自动识别；resultType对应返回结果的具体类型；resultMap对应返回结果的映射类型<resultMap>

resultMap - 定义映射的标签，就是定义查询语法中的字段名column和类型中的属性名property的映射关系。在MyBatis3.2+版本后，MyBatis可以跳过property直接访问field。有子标签id和result。id和result并没有严格的区分，可以使用id标签描述非PK字段和property的映射，可以使用result标签描述PK字段和property的映射。resultMap标签中的子标签有严格的顺序，先id后result。

sql - 定义SQL语法片段的。 <sql id="a">column1, column2</sql>

include - 在定义SQL语法的过程中，可以引用sql标签定义的语法片段。<include refid="a"/>

where - 用于动态定义where条件子句的标签，MyBatis可以根据标签内容，动态决定是否在SQL中增加where条件子句。判断依据是：where标签内没有有效数据，不提供where子句；有有效数据，提供where子句。<where>id = 1</where> <where></where>

if - 逻辑判断标签。属性test代表判断规则。属性的值为布尔值或布尔表达式。代表标签体内容是否生效。如：<where><if test="1=1">id = 1</if></where>

foreach - 循环标签。属性collection,item,begin,end,separator。collection属性是要循环的容器（包含数组，Collection集合，Map集合）。item是每次循环的数据。begin循环开始前输出什么。end循环结束后输出什么。separator是间隔符，代表每次循环输出的间隔符是什么。如：insert into xxx(id) <foreach collection="a" item="item" begin="values(" end=")" separator="),(">#{item.id}</foreach> 输出结果 -> values(id),(id),(id)

choose - 选择判断，相当于if else if else if else。<choose><when test="布尔值|布尔表达式"> </when><otherwise> </otherwise></choose>

set - 提供set子句动态逻辑判断的标签。可以动态根据标签内容决定是否输出set子句。用于更新SQL定义。如：update xxx <set><if test="">id=#{id},</if></set><where>xxx</where>。set标签可以自动剔除末尾逗号。

trim - 可以为标签体数据增加前后缀或删除前后缀。属性：prefix去除前缀、prefixOverrides增加前缀、suffix去除后缀、suffixOverrides增加后缀。

如：

<set>

<trim sufix=",">

<if test="">id=#{id},</if>

</trim>

</set>

<where>

<trim prefix="and">

<if test="">and id = #{id}</if>

</trim>

</where>

association - 多对一或一对一关联属性描述标签。一个标签代表实体中的一个引用类型对象。常用属性：column执行select查询语法初始化对象的时候，需要传入的参数是什么，对应当前select语法中的字段名、property代表当前描述的是实体中的哪一个属性、javaType代表当前属性的具体java类型、resultMap代表当前属性对应的映射标签是什么，对应resultMap标签的id、select代表当前对象使用哪一个查询语法初始化，对应另外一个select标签的id。

class A{B b;} class B{ String m;}

<resultMap id="a" type="A">

<association property="b" javaType="B">

<result column="m" property="m"/>

</association>

<association property="b" select="s2" column="{fk=id,m=m}"/>

</resultMap>

<select id="s1" resultMap="a">select id, m from xxx</select>

<select id="s2" resultType="B">select m from yyy where fk=#{fk}</select>

collection - 是描述一对多或多对多关系属性的标签，一个标签对应一个集合类型的属性。常用属性和association一样，代表含义相同。额外属性为ofType，代表集合的泛型。

## 工厂创建

InputStream inputStream = Xxx.class.getClassLoader().getResourceAsStream("xxx.xml");

SqlSessionFactory factory = SqlSessionFactory.build(inputStream);

## SqlSession常用方法

insert

update

delete

selectOne

selectList

selectMap

commit

rollback

## Mapper代理