

Spring bean 不被 GC 的真正原因

概述

自从开始接触 `Spring` 之后，一直以来都在思考一个问题，在 `Spring` 应用的运行过程中，为什么这些 `bean` 不会被回收？今天深入探究了这个问题之后，才有了答案。

思考点

大家都知道，一个 `bean` 会不会被回收，取决于**对象存活判定算法**。在 `JVM` 底层中使用的是**可达性分析算法**，抛开 `HotSpot` 的实现细节不谈，那么一个对象被判定为死亡，应该与 `GC Root` 不存在可达的引用路径。

- 1
- 所以，`Spring` 的 `bean` 肯定是与 `GC Root` 存在可达的引用路径，才不会被回收掉

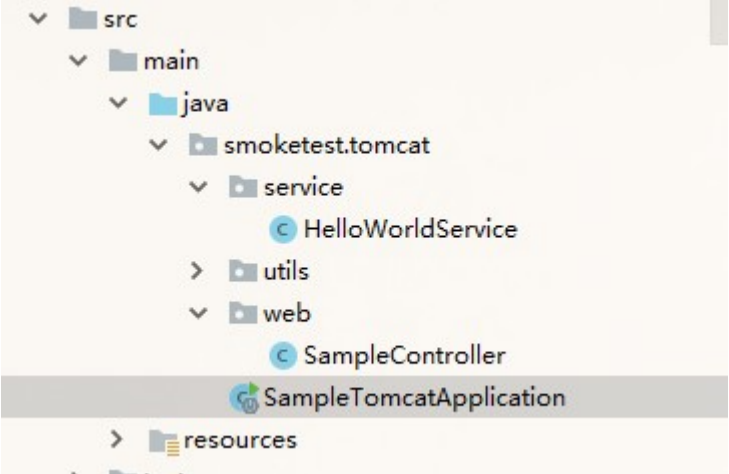
在 `Java` 语言对于 `GC Root` 的定义中，以下几种对象可以作为 `GC Root`：

- **虚拟机栈**的栈帧中的本地变量表中，引用类型对象所指向的堆中的对象
- 处于运行中状态（`RUNNABLE`，`BLOCKED`，`WAITING`，`TIMED_WAITING`）的线程对象
- `JDK` 自带的类加载器对象
- 本地方法所引用的对象
- `JVM` 持有的对象，例如基本类型的 `Class` 对象，`NullPointerException` 等常用异常对象
- 被 `synchronized` 关键字修饰的对象

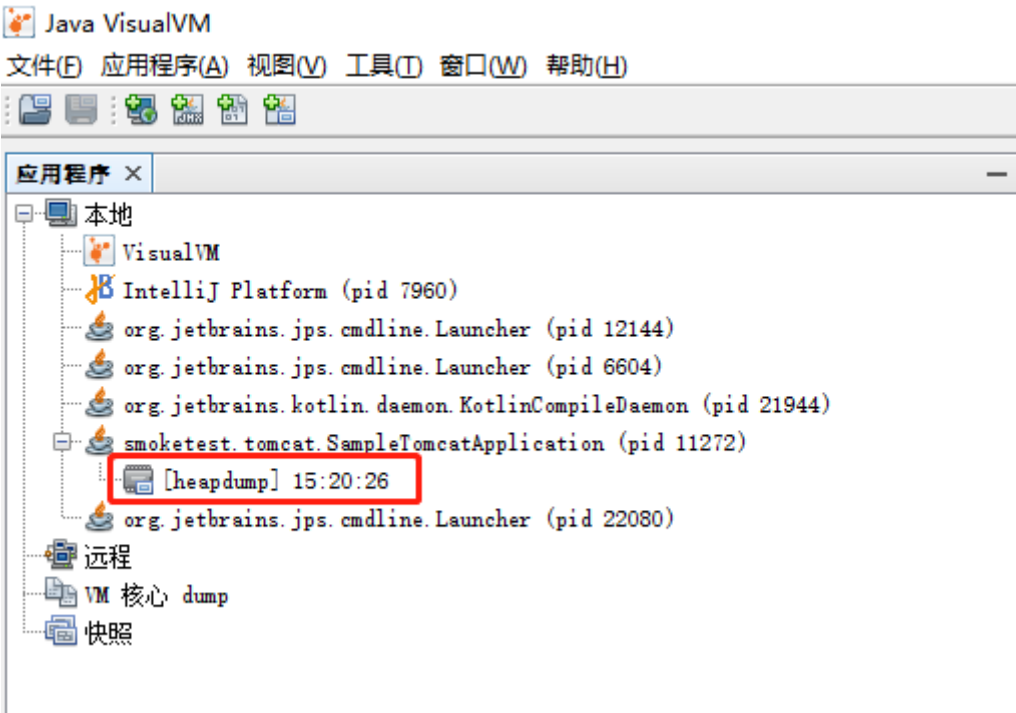
一般来说，只要是符合上面这几种规则的对象，或者能由上面的规则推导出存在引用的对象，都可以作为 `GC Root`。那么 `Spring` 的 `bean` 的 `GC Root` 是哪一种呢？或者说，找到了 `Spring` 的 `bean` 的 `GC Root`，就找到了问题的答案。

动手寻找答案

首先新建一个 `SpringBoot` 应用，里面定义了两个 `bean` 以及一个启动类，包结构如下：



然后点击运行启动类，启动完成之后，打开 `jvisualVM`，找到对应的应用，然后点击生成当前堆 `dump`：



然后打开后选择类，输入 `Hello` 过滤类名，找到 `HelloWorldService`，点击在实例视图中显示，发现只有一个实例存在，这符合我们的预期。最后右键点击这个实例，选择显示最近的垃圾回收根节点，可以观察到如下的引用路径：

引用	字段	类型	值
	<code>this</code>	<code>HelloWorldService</code>	#1
	<code>helloWorldService</code>	<code>SampleController</code>	#1
	<code>val</code>	<code>ConcurrentHashMap\$Node</code>	#8756
	<code>[187]</code>	<code>ConcurrentHashMap\$Node[]</code>	#34 512 个项
	<code>table</code>	<code>ConcurrentHashMap</code>	#3
	<code>singletonObjects</code>	<code>DefaultListableBeanFactory</code>	#1
	<code>beanFactory</code>	<code>AnnotationConfigServletWebServerApplicationContext</code>	#1
	<code>this\$0</code>	<code>AbstractApplicationContext\$1</code>	#1 SpringContextShutdownHook
	<code>[0]</code>	<code>Object[]</code>	#4131 64 个项
	<code>table</code>	<code>IdentityHashMap</code>	#4
	<code>hooks (sticky class)</code>	<code>ApplicationShutdownHooks</code>	class ApplicationShutdownHooks
	<code>[1]</code>	<code>Object[]</code>	#4131 64 个项
	<code>shutdownHook (循环至this.val.[187].table.singletonObjects.beanFactory)</code>	<code>AnnotationConfigServletWebServerApplicationContext</code>	#1
	<code>value</code>	<code>HashMap\$Node</code>	#4588
	<code>resourceLoader</code>	<code>WebMvcAutoConfiguration\$EnableWebMvcConfiguration</code>	#1
	<code>applicationContext</code>	<code>WebMvcAutoConfiguration\$EnableWebMvcConfiguration</code>	#1
	<code>applicationContext</code>	<code>ContentNegotiatingViewResolver</code>	#1
	<code>messageSource</code>	<code>MessageSourceAccessor</code>	#10
	<code>applicationContext</code>	<code>BeanNameViewResolver</code>	#1
	<code>messageSource</code>	<code>MessageSourceAccessor</code>	#8
	<code>applicationContext</code>	<code>InternalResourceViewResolver</code>	#1
	<code>messageSource</code>	<code>MessageSourceAccessor</code>	#9

可以看到，`DefaultListableBeanFactory` 和 `AnnotationConfigServletWebServerApplicationContext` 都是我们比较熟悉的 `bean` 容器，对应的往下找发现有 `ConcurrentHashMap$Node` 引用。我们都知道在 `Spring` 中，正是这两个容器（准确地说是 `DefaultListableBeanFactory`）中使用 `ConcurrentHashMap` 存放了实例化好的 `bean`。这都是非常符合我们预期的。但是在 `AbstractApplicationContext` 再往上找后，发现有个叫 `ApplicationShutdownHooks` 的东西。意思就是说，我们的容器，最终与这个 `ApplicationShutdownHooks` 的东西扯上了引用关系。接着我们翻阅 `Spring` 源码进行求证：

```
@Override
public void registerShutdownHook() {
    if (this.shutdownHook == null) {
        // No shutdown hook registered yet.
        this.shutdownHook = (Thread) run() -> {
            synchronized (startupShutdownMonitor) {
                doClose();
            }
        };
        Runtime.getRuntime().addShutdownHook(this.shutdownHook);
    }
}
```

发现在 `AbstractApplicationContext` 的 `registerShutdownHook` 方法中调用了这一行代码，而 `registerShutdownHook` 方法正是在 `Spring` 容器初始化时要调用的方法：

src

main

java

smoketest.tomcat

service

utils

web

SampleTomcatApplication

resources

test

target

flattened-pom.xml

pom.xml

spring-boot-smoke-test-tomcat.iml

514

515

516

517

518

519

520

521

522

523

524

525

526

527

528

529

530

531

532

533

534

535

536

537

538

539

540

541

542

543

544

545

546

547

548

549

550

551

552

553

554

555

556

557

558

559

560

561

562

563

564

565

566

567

568

569

570

571

572

573

574

575

576

577

578

579

580

581

582

583

584

585

586

587

588

589

590

591

592

593

594

595

596

597

598

599

600

601

602

603

604

605

606

607

608

609

610

611

612

613

614

615

616

617

618

619

620

621

622

623

624

625

626

627

628

629

630

631

632

633

634

635

636

637

638

639

640

641

642

643

644

645

646

647

648

649

650

651

652

653

654

655

656

657

658

659

660

661

662

663

664

665

666

667

668

669

670

671

672

673

674

675

676

677

678

679

680

681

682

683

684

685

686

687

688

689

690

691

692

693

694

695

696

697

698

699

700

701

702

703

704

705

706

707

708

709

710

711

712

713

714

715

716

717

718

719

720

721

722

723

724

725

726

727

728

729

730

731

732

733

734

735

736

737

738

739

740

741

742

743

744

745

746

747

748

749

750

751

752

753

754

755

756

757

758

759

760

761

762

763

764

765

766

767

768

769

770

771

772

773

774

775

776

777

778

779

780

781

782

783

784

785

786

787

788

789

790

791

792

793

794

795

796

797

798

799

800

801

802

803

804

805

806

807

808

809

810

811

812

813

814

815

816

817

818

819

820

821

822

823

824

825

826

827

828

829

830

831

832

833

834

835

836

837

838

839

840

841

842

843

844

845

846

847

848

849

850

851

852

853

854

855

856

857

858

859

860

861

862

863

864

865

866

867

868

869

870

871

872

873

874

875

876

877

878

879

880

881

882

883

884

885

886

887

888

889

890

891

892

893

894

895

896

897

898

899

900

901

902

903

904

905

906

907

908

909

910

911

912

913

914

915

916

917

918

919

920

921

922

923

924

925

926

927

928

929

930

931

932

933

934

935

936

937

938

939

940

941

942

943

944

945

946

947

948

949

950

951

952

953

954

955

956

957

958

959

960

961

962

963

964

965

966

967

968

969

970

971

972

973

974

975

976

977

978

979

980

981

982

983

984

985

986

987

988

989

990

991

992

993

994

995

996

997

998

999

1000

public void registerShutdownHook() {

if (this.shutdownHook == null) {

this.shutdownHook = new Thread(name: "SpringContextShutdownHook") {

public void run() {

synchronized (AbstractApplicationContext.this.startupShutdownMonitor) { startupShutdownMonitor: Object@2954

AbstractApplicationContext.this.doClose();

}

}

}

Runtime.getRuntime().addShutdownHook(this.shutdownHook); shutdownHook: "Thread[SpringContextShutdownHook, 5, main]"

AbstractApplicationContext

registerShutdownHook()

Debugger

Endpoints

Frames

Threads

Variables

Console

"main" @1 in group "main": RUNNING

registerShutdownHook:953, AbstractApplicationContext (org.springframework.context.support)

refreshContext:399, SpringApplication (org.springframework.boot)

run:315, SpringApplication (org.springframework.boot)

run:1226, SpringApplication (org.springframework.boot)

run:1215, SpringApplication (org.springframework.boot)

main:82, SampleTomcatApplication (smoketest.tomcat)

D:\Java\jdk_1.8.0_291\bin\java.exe ...

Connected to the target VM, address: '127.0.0.1:55488', transport: 'socket'

._._._._._

(O\ _ _ _ _ _

W _ _ _ _ _

这说明在 Spring 容器初始化时，调用的这个方法，然后在继续往里跟踪这个方法：

```
201 * @see #removeShutdownHook
202 * @see #halt(int)
203 * @see #exit(int)
204 * @since 1.3
205 */
206 @ public void addShutdownHook(Thread hook) {
207     SecurityManager sm = System.getSecurityManager();
208     if (sm != null) {
209         sm.checkPermission(new RuntimePermission("shutdownHooks"));
210     }
211     ApplicationShutdownHooks.add(hook);
212 }
213
```

```
59 @ private ApplicationShutdownHooks() {}
60
61 /* Add a new shutdown hook. Checks the shutdown state and the hook itself,
62  * but does not do any security checks.
63  */
64 @ static synchronized void add(Thread hook) {
65     if(hooks == null)
66         throw new IllegalStateException("Shutdown in progress");
67
68     if (hook.isAlive())
69         throw new IllegalArgumentException("Hook already running");
70
71     if (hooks.containsKey(hook))
72         throw new IllegalArgumentException("Hook previously registered");
73
74     hooks.put(hook, hook);
75 }

```

```
class ApplicationShutdownHooks {
    /* The set of registered hooks */
    private static IdentityHashMap<Thread, Thread> hooks;
    static {
        try {
            Shutdown.add( slot: 1 /* shutdown hook invocation order */,
                registerShutdownInProgress: false /* not registered if shutdown in p
            new Runnable() {
                public void run() { runHooks(); }
            }
        );
        hooks = new IdentityHashMap<>();
    } catch (IllegalStateException e) {
        // application shutdown hooks cannot be added if
        // shutdown is in progress.
        hooks = null;
    }
}
```

最后我们可以发现，AbstractApplicationContext 中的 Thread shutdownHook 变量，最终被放在了 ApplicationShutdownHooks 的这个 map 里面，而这个 map 恰好就是一个静态变量。

结论

所以，Spring 的 bean 没有被回收，正是在 AbstractApplicatuonContext 的 registerShutdownHook 方法中，与 ApplicationShutdownHooks 中的一个静态变量建立了可达的引用路径。

题外话

那么为什么类的静态变量可以作为 GC Root 呢？抱着严谨的心态，我们继续往下求证：

- 1
- 类的静态变量属于类对象，类对象由类加载器进行加载，而类加载器是 GC Root，那么类加载器是不是与被加载的类对象存在引用关系呢？

翻阅 ClassLoader 类，赫然看到这一段代码：

```
1 public abstract class ClassLoader {
2     // The classes loaded by this class loader. The only purpose of this table
3     // is to keep the classes from being GC'ed until the loader is GC'ed.
4     private final Vector<Class<?>> classes = new Vector<>();
5 }
```

注释一目了然，好家伙！原来类加载器把所有的已加载的类对象都保存在这个容器里面，怪不得类对象和类静态变量也属于 GC Root

相关文章：[Spring管理单例对象的时候,如何实现不被JVM回收的？](#)

singletonObjects从属于ApplicationContext，只要ApplicationContext不被回收，singletonObjects就不会被回收。而ApplicationContext，就有多种情况了 你手动创建，例如在main方法中，那么生存周期根据你的代码而定。 整合到Servlet中，那么应用服务器持有ApplicationContext 引用，服务器不关闭则引用不失效。 SpringBoot类似于 或

这里是第二种解释，即web服务器，我们在启动ioc容器时，会在ServletContext里把它设置为根容器（springboot），传统项目还有子容器，子容器跟父容器有关联，父容器被设置在ServletContext里，服务器在，它们都在，至于底层是如何保证ServletContext不被gc的，这个需要追源码了，我看了下visualvm的实例引用信息，实在太复杂，很难找到ServletContext的gc root在哪。还有一个问题是内嵌的tomcat是ioc容器的一个属性（也就是tomcat的生命周期与ioc容器一致），那么这第二种解释就不适用了，得依赖本文的解释，如果是传统的项目，那么第二种解释是可行的。

Springboot是在SpringApplication#refreshContext里注册本文的钩子函数的，传统项目是没有这这个步骤的。

另外多例bean是会被gc回收的，因为它在使用的时候创建，IOC没有设置一个容器来保存实例对象。