# Knowledge Sharing

## OOP

### 基本概念

|  |  |
| --- | --- |
| **Abstraction** | 隐藏细节，暴露功能。(面向接口编程)  Hiding internal details and showing functionality |
| **Inheritance** | 重用基类的属性和方法。  create a class based on an existing one, reuse the fields and method of that class. |
| **Encapsulation** | 将类的属性设置成私有，通过类的公有方法访问这些属性。  Keep fields private in a class, then provide public methods to access those fields.  Java module 提供了一种封装方式。 |
| **Polymorphism** | 多种形态，包括overload 和 override. |

### 类&对象&接口

* **类(Class)**是构建对象的模板或蓝图。
* **对象**的三个主要特征：
* 对象的行为(behavior), 即方法, 对象可以完成哪些操作。
* 对象的状态(state, 即属性或字段)）。
* 对象的标识(identity), 区分对象。(不是所有对象都有标识)
* **对象(object) 和 对象的类(class) 和 对象的类型（type）**

Person jack = new Employee(“Jack”); // Person是Employee的超类/父类/基类

1. new 操作创建了一个Employee类的对象，返回一个引用并赋值给Person类型的变量jack。
2. 对象与对象变量之间的关系，对象变量引用这个对象，但对象变量本身不是对象。
3. 对象变量存储在JVM内存模型的栈（stack）中, 而对象本身存储在JVM内存模型的堆(Heap)。
4. 对象的类(class)定义了对象(object)的内部状态和操作的实现。
5. 对象的类型(type) 决定了此对象能响应的请求的集合，也就是访问这个对象的接口。
6. 一个对象可以有许多类型，对象的类型可以是这个对象的类以及它的任意超类或接口。

* **识别类 (即建模modeling),** 如何从真实世界的问题描述中抽象出类

1. **名词建模法**，通常名词是类，动词是类的方法，形容词等描述性词汇是类的字段， 这种方式比较直观，和真实世界的概念直接反应。

比如：护士给病人打标准剂量的流感疫苗，可以识别出 Nurse, Patient, Vaccine 三个类，Nurse可以完成打疫苗的操作，也就是Nurse承担打疫苗的责任(Responsibility)。

aNurse.administrateFluVaccine(aPatient, aVaccine);

1. **动词建模法**， 找出动词是否产生有价值的过程数据。

比如：

1. 通过银行账号取钱或存钱，取钱和存钱都产生了交易记录(transaction), 所以要为交易建模。
2. 请假申请需要审批，审批产生记录，需要对 审批记录建模。
3. **也可以对描述性的词汇进行抽象建模。**

比如，策略模式中可以对类型码进行抽象，员工(Employee)的薪资策略和他们的岗位相关，可以对岗位类型进行抽象。

* **类与类之间的关系(从语义semantics上讲)**
* **依赖(“use-a”),** 订单与账户(Order & Account), 订单需要通过账户查看客户信用。
* **聚合(“has-a”),** 订单与订单项(Order & OrderItem)。
* **继承(“Is-a”) ,** 子类和父类的关系是 Is-a relationship, a safe door is a door

类与类之间的四种关系的定义：<https://blog.csdn.net/u012207345/article/details/72933691>

* 依赖(Dependency)关系是类与类之间的联接。依赖关系表示一个类依赖于另一个类的定义。例如，一个人(Person)可以买车(car)和房子(House)，Person类依赖于Car类和House类的定义，因为Person类引用了Car和House。与关联不同的是，Person类里并没有Car和House类型的属性，Car和House的实例是以参量的方式传入到buy()方法中去的。**一般而言，依赖关系在Java语言中体现为局域变量、方法的形参，或者对静态方法的调用。**
* 关联(Association）关系是类与类之间的联接，它使一个类知道另一个类的属性和方法。**关联可以是双向的，也可以是单向的。**在Java语言中，关联关系一般使用成员变量来实现。
* 聚合(Aggregation) 关系是关联关系的一种，是**强的关联关系。聚合是整体和个体之间的关系。**例如，汽车类与引擎类、轮胎类，以及其它的零件类之间的关系便整体和个体的关系。与关联关系一样，聚合关系也是通过实例变量实现的。但是关联关系所涉及的两个类是处在同一层次上的，而在聚合关系中，两个类是处在不平等层次上的，一个代表整体，另一个代表部分。
* 组合(Composition) 关系是关联关系的一种，是**比聚合关系强的关系。它要求普通的聚合关系中代表整体的对象负责代表部分对象的生命周期，组合关系是不能共享的。**代表整体的对象需要负责保持部分对象和存活，在一些情况下将负责代表部分的对象湮灭掉。代表整体的对象可以将代表部分的对象传递给另一个对象，由后者负责此对象的生命周期。换言之，代表部分的对象在每一个时刻只能与一个对象发生组合关系，由后者排他地负责生命周期。部分和整体的生命周期一样。
* **接口**
* 接口是行为/功能集合的抽象，定义了行为规范。
* 接口是一种契约(Contract) 或定义了通信的协议。
* 接口可以用于定义类型。
* 无任何抽象方法的接口可以看做标识。比如Spring Framework中Aware 接口。
* **抽象类 vs 接口**
* 语法(Syntax)的不同
* 语义(Semantics)的不同（或设计的不同）， 子类和超类是 **Is-a** 的关系；而接口是定义某种功能，实现某个接口的类，具有接口中定义的功能。
* 面向接口编程， 这里的接口指的是超类类型(父类或接口)。

### 对象建模

将业务领域中的概念抽象成对象，避免设计出只有数据的贫血对象，对象设计的核心为：

* **角色与职责**

1. 行为的协作者
2. 信息专家模式
3. 单一职责原则

* **抽象**

1. 提炼行为特征

2. 封装变化

还要从重用(Reusable)、解耦(Decoupling)提高灵活性(Feasible)来应对变化等角度考虑设计。

### 面向对象的设计原则

* 封装变化
* 多用组合，少用继承
* 针对接口编程，不针对实现编程
* 为交互对象之间的松耦合设计而努力
* 类应该对扩展开放，对修改闭合 – 开闭原则
* 依赖抽象，不要依赖具体类 – 依赖倒置原则
* 只和朋友交谈 – 迪米特法则
* 别找我，我会找你
* 类只有一个改变的理由 – 单一职责原则

**SOLID 原则 + 其他原则**

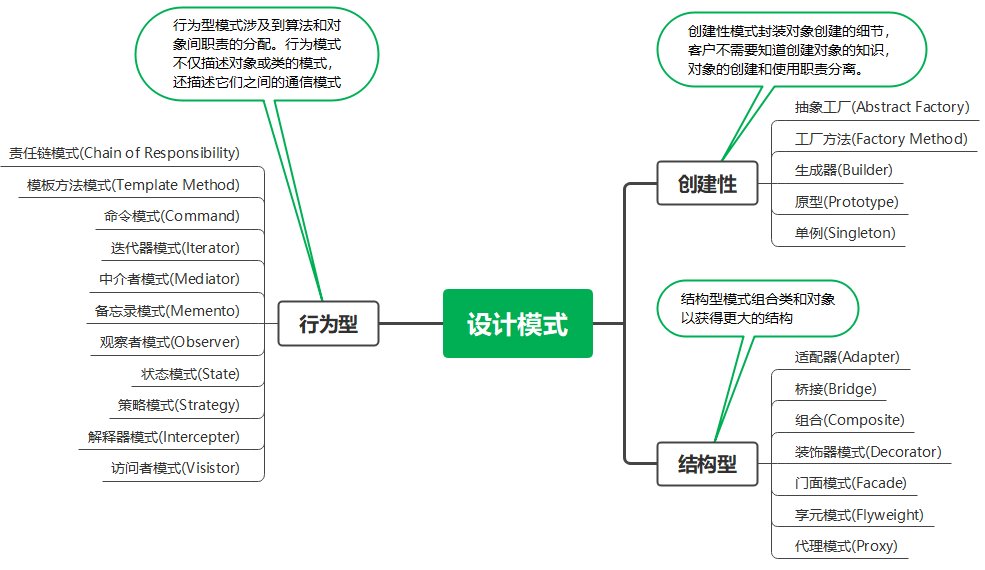
|  |  |
| --- | --- |
| **单一职责原则**  **(Single Responsibility)** | 一个类只有一个引起它变化的原因。 |
| **开发闭合原则(Open-Closed)** | 对扩展是开放的，对更改是封闭的。 |
| **Liskov替换原则**  **(Liskov Substitution)** | 子类型必须能够替换它的父类型，这是类继承设计原则，类继承的is-a关系是就行为而言的。违背此原则的例子：  1. 功能退化的子类违背里氏替换原则。2.子类共有的行为，没有被提炼到父类。3.长方形和正方形 |
| **接口隔离原则(Interface Segregation)** | 接口的设计原则，类的接口要是高内聚的。不要强迫客户依赖它们不用的方法。不要接口污染。 |
| **依赖倒置原则**  **(Dependency Inversion)** | 高层模块不依赖于底层模块，二者都依赖于抽象，面向接口编程  每个高层次都为它所需要的服务声明一个抽象接口，较低的层次实现这个接口，每个高层次通过接口依赖低层次。 |
| **迪米特法则 /最少知识原则** | 一个类对于其他类知道的越少越好，就是说一个对象应当对其他对象有尽可能少的了解。Only talk to your immediate friends，即只与“直接的朋友”通信。应用到面向对象的程序设计中时，可描述为 "类应该与其协作类进行交互但无需了解它们的内部结构"。一个对象或方法只能调用对象自身，传入的对象或内部创建的对象。 |
| **Tell, Don't Ask** | 最好是向对象发出执行某些操作或逻辑的命令，而不是查询其状态，然后采取某些行动。 |
| **KISS Keep it simple, stupid.** | 不要过度设计 |
| **YAGNI**  **You aren't gonna need it** | 不要过度设计 |
| **DRY Don't repeat yourself** | 避免重复。 |
| **信息专家模式** | 如果某个类拥有完成某个职责所需要的所有信息，那么这个职责就应该分配给这个类来实现。信息的持有者即为操作该信息的专家，优先考虑将信息相关的行为分配给这些信息的持有者。DDD聚合的设计就符合信息专门模式。 |
| **奥卡姆剃刀原理** | 原理称为“如无必要，勿增实体”，即“简单有效原理”。 |

### 设计模式 – 可复用的面向对象设计

复用机制包括：

* 类继承
* 对象组合
* 泛型

面向对象系统中功能复用的两种技术是类继承和对象组合，优先使用对象组合，委托(delegation) 是对象组合的一种方式。



注：除了上面介绍的23种设计模式之外，还有其他模式，如规格模式(Specification)

**Java & Spring中的模式：**

|  |  |
| --- | --- |
| 装饰器模式 | FilterInputStream  Collections.SynchronizedXXX  Collections.UnmodifiableXXX |
| 观察者模式 | Spring的Event-Listener |
| 迭代器模式 | Iterator |
| 原型模式 | Object#clone() 浅克隆，深克隆 |
| 责任链模式 | Spring 的拦截器，Web的Filter |
| 代理模式 | JDK的动态代理 Proxy和CGLIB代理, CGLIB用于Spring AOP，JDK中的proxy必须基于接口，CGLIB却没有这个限制。 |
| 工厂模式 |  |
| 模板方法模式 | AbstractApplicationContext.onFresh() |
| 策略模式 | Thread 和 Runnable |
| 单例模式 | 数据库连接对象 |

## Java Foundation

### Modifier修饰符

|  |  |
| --- | --- |
| Access modifier | public, protected, private, package-private (no access modifier) |
| final | Final class cannot be inherited;  Final method cannot be overridden.  Final field, its value cannot be changed once it has been initialized at the declaration or by a constructor. |
| abstract | Abstract class can’t be instantiated.  The combination of **abstract** and **final** is illegal. |
| static | Static fields and methods belong to the class, rather than one object.  Static field:   * When the value of the field is independent of objects. * When the value of the field is supposed to be shared across the objects.   static methods:   * to access/manipulate static variables and other static methods that don't depend upon objects. * static methods are widely used in utility and helper classes. |
| synchronized | Java provides synchronized keyworks to implement synchronization.  Synchronization in Java is a capability to control the access of multiple threads to shared resource.  When a method or a block of code is marked with synchronized keyword, it can only be accessed by one thread at a time. |
| volatile | The value of an attribute is not cached thread-locally, and is always read from the "main memory" |
| transient | Attributes and methods are skipped when serializing the object containing them |

### Constructor构造器

* Default constructor (no argument)
* Subclass constructor 必须在第一行调用superclass的构造器，如果调用superclass有默认构造器，可以省略。

### Generic泛型

Generic allows you to **abstract over type** through type parameter （类型参数）.

* Generic class
* Generic method:

|  |
| --- |
| List<String> ls = new ArrayList<String>(); //legal  List<Super> ls1 = new ArrayList<Sub>(); //illegal, Super is superclass of Sub |

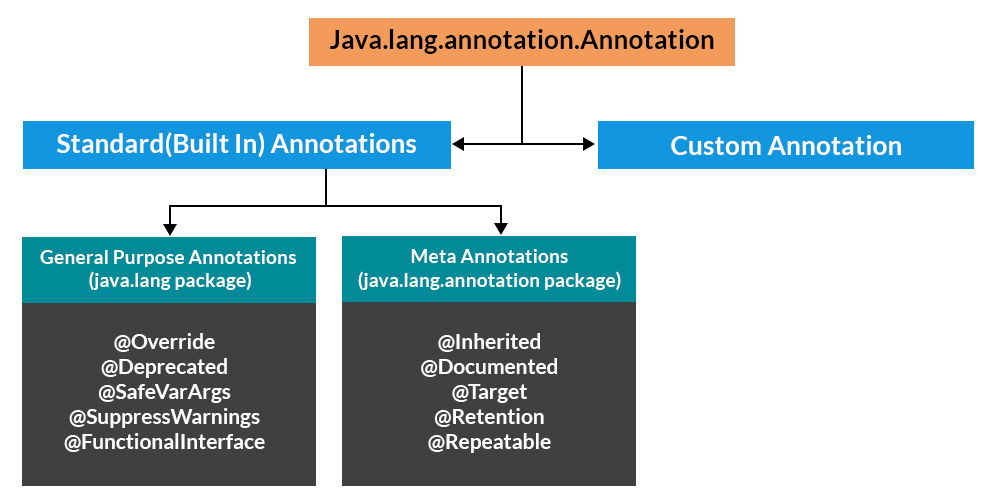
**Restrictions on Generics:**

* [Cannot Instantiate Generic Types with Primitive Types](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#instantiate)
* [Cannot Create Instances of Type Parameters](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createObjects)
* [Cannot Declare Static Fields Whose Types are Type Parameters](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createStatic)
* [Cannot Use Casts or instanceof With Parameterized Types](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#cannotCast)
* [Cannot Create Arrays of Parameterized Types](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createArrays)
* [Cannot Create, Catch, or Throw Objects of Parameterized Types](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#cannotCatch)
* [Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#cannotOverload)

### Annotation

Java Annotation is a form of **metadata**, provides data about a program that is not part of the program itself, annotation information can be processed at compile-time or runtime.

Java Annotation is a tag that represents metadata, it provides additional information to a program, it can be used to describe a class, method, field, etc.



@Retention: source, class, runtime

@Target: type, method, field, parameter, constructor, package, module, etc.

### Abstract Super Class & Interface

Java语言类只能但继承，接口可以多继承。

* 子类和抽象父类之间的关系：Is-a
* 实现类和接口直接的关系：实现类具有接口中声明的功能

案例：

|  |
| --- |
| // An Integer is a Number, and has the functions of Comparable  public final class Integer extends Number implements Comparable<Integer>  public SafeDoor extends Door implements Alarm {  public void open() {//…};  public void close() {//…};  public void alarm() {//…};  } |

### Object Class

Object是属于所有类的基类，它有以下方法：

* clone
* equals
* hashCode
* notify
* notifyAll
* toString
* wait

### Equal operator & equals() & hashCode()

**== 和 equals()方法的区别:**

判断下面的代码中 s1, s2, s3 and s4 的相等性

|  |
| --- |
| String s1 = “abc”;  String s2 = “ab” + “c”;  String tmp = “ab”;  String s3 = tmp + “c”;  String s4 = new String(“abc); |

**Java规范，override equals()方法, 同时override hashCode()方法**

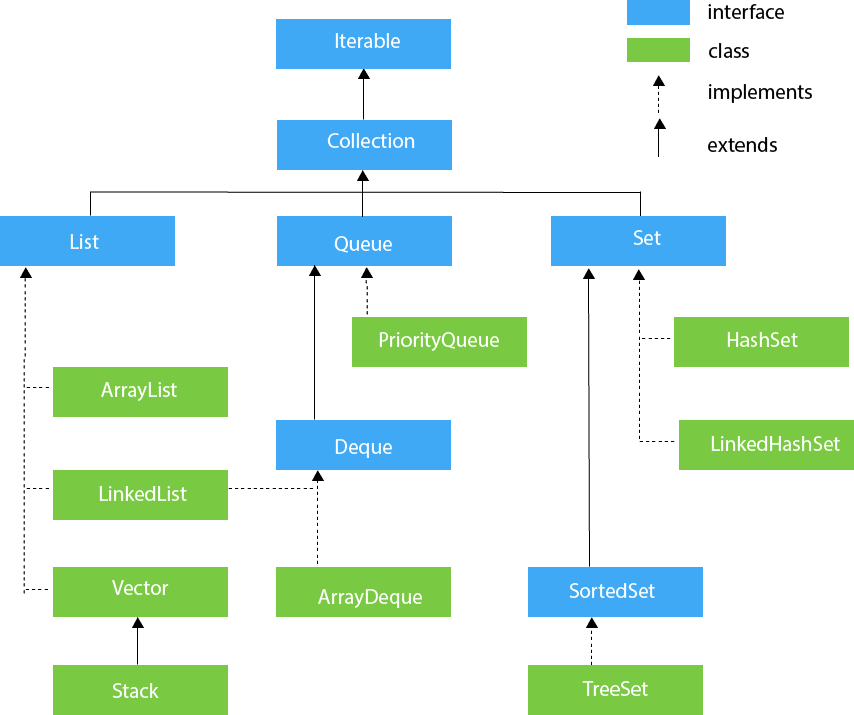
* 两个对象equals()方法相等，则hashCode()相等
* 两个对象的hashCode()相等，equals()方法不一定相等

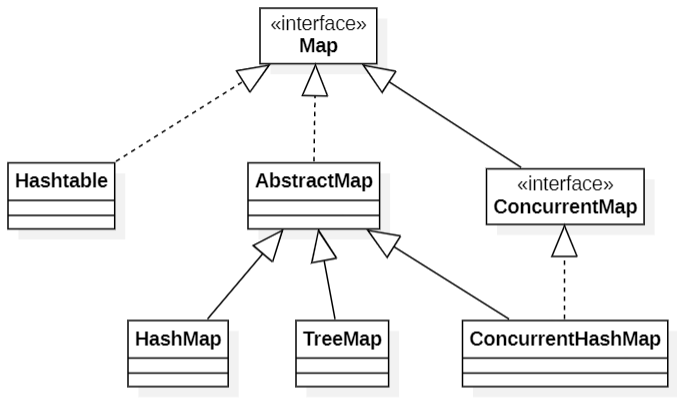
In java, both the **HashMap** and **HashSet** classes use **hashing algorithms** to store and retrieve data.

HashMap的put(k, v)方法, 先比较hashCode, 再用equal operator, 然后再用equals()方法比较key是否已经存在，如果存在，key不变，替换原来的value, 返回原来的value.

|  |
| --- |
| if (p.hash == hash &&  ((k = p.key) == key || (key != null && key.equals(k)))) |

### Collections & Map





|  |  |
| --- | --- |
| Collection | Collection is an interface, a collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not, some collections are ordered and others are unordered. Collection has sub-interfaces such as List, Set and Queue. |
| List | An ordered collection, it allows duplicate elements |
| ArrayList | Resizable array implementation, initial capacity and maximum capacity.  Each ArrayList instance has a capacity. As elements are added to an ArrayList, its capacity grows automatically.  it contains elements that can be accessed using an integer index.  It is faster to access an element but slower to add or remove an element compared with LinkedList |
| LinkedList | Doubly-linked list implementation. |
| Vector | Thread-safe |
| Set | A collection that contains no duplicate elements, it contains at most one null element. |
| HashSet | It is backed by HashMap |
| TreeSet | It is based on a TreeMap. The elements are ordered using their natural ordering, or by a Comparator |
| Queue |  |
| Map | An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value. |
| HashMap | HashMap has initial capacity and maximum capacity.  Allow at most one null key, and multiple null values.  Using hash code first, and then equals operator and equals() method to check for duplicate key.  If key is already present, replaces its value and returns old value.  if (p.hash == hash &&  ((k = p.key) == key || (key != null && key.equals(k)))) |
| Hashtable | Thread-safe, not allow null key.  Using hash code and equals() method to check for duplicate key.  If key is already present, replaces its value and returns old value.  if ((entry.hash == hash) && entry.key.equals(key)) |
| ConcurrentHashMap | It is thread-safe.  One of the key features of the ConcurrentHashMap is that it provides **fine-grained locking**, **meaning that it locks only the portion of the map being modified, rather than the entire map**. This makes it highly scalable and efficient for concurrent operations. Additionally, the ConcurrentHashMap provides various methods for atomic operations such as putIfAbsent(), replace(), and remove().  In Java 8, 采用**synchronized+CAS+红黑树**来实现的。锁的粒度也从段锁缩小为结点锁. |
| Collections | 提供了工厂方法生成synchronized的集合 和 unmodifiable 的集合 |

**Questions:**

1. Why doesn’t HashSet allow duplicate element?
2. How to Iterator a Map?
   * 1. entrySet()
     2. keySet()
     3. forEach()
3. Why do we need to override equals() and hashCode() method when creating a class?

### IO APIs

1. Byte Stream
2. Char Stream

### Exception

* Checked exception
* Unchecked exception (RuntimeException)

### Thread

线程的实现方式

线程的状态，一个启动的线程可否再启动？

线程池，如何创建线程池？

创建多线程的数量的基准？

Java多线程的内存模型

Servlet的工作模式是单实例多线程

### Thread Pool

**Executors工具类提供工厂方法创建线程池**, 按类型分为：

* Cached: Creates a thread pool that creates new threads as needed.
* Fixed
* Single
* Scheduled
* Work-Stealing

**自定义线程池及参数**

|  |
| --- |
| **public** ThreadPoolExecutor(**int** corePoolSize,  **int** maximumPoolSize,  **long** keepAliveTime,  TimeUnit unit,  BlockingQueue<Runnable> workQueue,  ThreadFactory threadFactory,  RejectedExecutionHandler handler) |

### Synchronized & Lock & ReadWriteLock

|  |  |
| --- | --- |
| **Synchronized** | * Java关键字，内置锁。 * 获取锁的线程执行完同步代码后自动释放锁，或线程执行发生异常自动释放锁。 * 获取锁时候需要一直等待其他线程释放锁。 * 可重入、不可中断、非公平 * 不可以判断锁的状态 |
| **Lock/**  **ReentrantLock**  **Condition**  (refre to BlockingQueue) | * 获取锁的线程执行完同步代码后不能自动释放，应该在finally子句中手工调用unlock()方法释放锁。 * 获取锁的时候可以指定时间tryLock(long timeout, TimeUnit unit)，不需要一直等待。 * 可重入，可中断，可公平 * 可以判断锁的状态. * Condition用于线程间通信，await()方法让当前的线程等待直到它被唤醒或中断；signal()方法唤醒任一等待的线程。类似于Object的wait()和notify()线程。ReentrantLock has newCondition() method for creating a Condition object which is bound to the lock; invoking await() method of Condition object causes the current thread to wait until it is signaled or interrupted; invoking signal() method of Condition object to wait up one of waiting threads. |
| ReadWriteLock/  ReentrantReadWriteLock | A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing. The read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive. All ReadWriteLock implementations must guarantee that the memory synchronization effects of writeLock operations (as specified in the Lock interface) also hold with respect to the associated readLock. That is, a thread successfully acquiring the read lock will see all updates made upon previous release of the write lock.  ReadWriteLock维护了一对锁，一个读锁和一个写锁，对于只读方法加上ReadLock, 对于写方法加上WriteLock, 在同一时刻允许多个线程进行读操作，只要没有写操作；写锁是互斥的。 通过分离读锁和写锁，是的并发性相比一般的排它锁有了很大的提升. 在读多写少的情况下，使用ReadWriteLock比排它锁更好的并发性和吞吐量. |
| **synchronized的优化** | synchronized关键字在 JDK1.6 版本之前，是通过操作系统的 Mutex Lock 来实现同步的，属于重量锁。  在 JDK1.6 版本中，HotSpot 虚拟机开发团队花了很大的精力去实现各种锁优化技术，如：适应性自旋、锁消除、锁粗话、偏向锁、轻量级锁等。其中最重要的是：**自旋锁、轻量级锁、偏向锁**   |  |  |  |  | | --- | --- | --- | --- | | **现状** | **锁名称** | **收益** | **使用场景** | | 大多数情况下，等待锁的时间比操作系统 mutex 短得多 | 自旋锁 | 减少内核态与用户态切换的开销 | 线程获取锁时间较短的情况 | | 大多数情况下，锁同步期间没有线程竞争 | 轻量级锁 | 与自旋锁相比，减少了自旋时间 | 没有线程竞争锁 | | 大多数情况下，锁同步期间没有线程竞争 | 偏向锁 | 与轻量级锁相比，减少了多余的对象复制操作 | 没有线程竞争 | |

### Future & FutureTask & Callable

|  |  |
| --- | --- |
| **Runnable** | The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run. |
| **Callable** | A task that returns a result and may throw an exception |
| **Future** | Represents the result of an asynchronous computation.  获取结果的get() 是阻塞的。 |
| **FutureTask** | A cancellable asynchronous computation, 实现了Runnable and Future接口  A FutureTask can be used to wrap a **Callable** or **Runnable** object. Because FutureTask implements Runnable, a FutureTask can be submitted to an Executor for execution.  获取结果的get() 是阻塞的.  FutureTask is an implementation of Future. |
| **Executor** | An object that executes submitted Runnable tasks. This interface provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc  执行提交的可运行任务的对象。这个接口提供了一种将**任务提交**与**每个任务将如何运行**的机制解耦的方法，包括线程使用、调度等细节。 |
| **ExecutorService** | Sub interface of Executor.  An Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.  提供管理终止的方法的Executor，以及可以生成用于跟踪一个或多个异步任务进度的Future的方法。 |
| **CompletableFuture** |  |

### CountDownLatch & CyclicBarrier & Semaphore

|  |  |
| --- | --- |
| **CountDownLatch** | A thread waits while other threads count down on the latch until it reaches zero. |
| **CyclicBarrier** | A group of threads waits together until all of the threads arrive. At that point, the barrier is broken and an action can optionally be taken. |
| **Semaphore** | a semaphore maintains a set of permits. Each acquire blocks if necessary until a permit is available, and then takes it. Each release adds a permit, potentially releasing a blocking acquirer.  Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource. |

### Thread-Safe & 线程内存模型

* 什么是线程安全？
* Java中线程安全的, 如类String, Integer, Vector, Hashtable, ConcurrentHashMap, StringBuffer, Java 8新增加的时间和日期的类等。
* 什么样的类是线程安全的？
* Stateless class 无状态的类
* Immutable class 不可变的类
* Synchronized 访问类的属性/字段的方法实现同步
* Lock 对访问类的属性/字段加锁
* 使用Java中线程安全的类作为类的字段
* ThreadLocal
* 原则性、可见性、排序性
* 线程**本地内存**和**主内存**

### Lambda Expression

1. Lambda expression 代表一个函数，由arguments（或者no arguments），body, 或 Arrow Token组成。（函数式编程中函数式一等公民）

2. Lambda expression 提供了functional interface的一种实现方法，lambda expression 返回一个functional interface的一个实例。e.g.:

|  |
| --- |
| //变量类型必须是 functional interface  Runnable runnable = () -> System.out.println("hello, world"); |

3. 只有一个抽象方法的接口叫函数式接口（functional interface），lambda expression 只能为一个方法提供body, 所以functional interface 只能有一个抽象方法。

4. 因为functional interface中的抽象方法没有声明异常，所以lambda expression不能直接处理checked exception, 需要将checked exception转换成unchecked expression。网上有例子如何避免在lambda expression中使用try-catch block

5. Lambda expression可以使用本地变量，本地变量必须是final的，或事实不可改变的， 也就是初始化后不能再一次在lambda expression外赋值。

6. Lambda expression的好处，

1）减少代码量。

2）可以方便的处理大数据集。

3）可以方便的利用多核CPUs, 比如Stream API提供的并行操作。

### Stream API

1. Collection用于存取数据，Stream用于并行或串行操作数据。
2. Stream operations 分为中间操作(intermediate operations)和终结操作(terminal operations); intermediate operations 如filer, map, flatMap, sort, distinct 等，用于转换一个stream到另外一个stream; 而终结操作如reduce, collect, sum, average, max, min, forEach, anyMatch, findAny 等计算出一个结果。
3. Stream operations 可以组成Stream 流水线， stream pipeline由 一个 source, 零个、1个或多个 中间操作 和一个终结操作组成，中间操作将不执行，直到终结操作被调用。
4. 一个Stream对象被调用后，不能再次被调用, 也就是stream 对象只能被消费一次。
5. Stream的数据源source可以来自 Collection, Array, IO channel 或者Stream 自身的创建者。

E.g.: Arrays.stream(new String[]{"a","b","c"});

IntStream.rangeClosed(1, 10);

1. 并行stream, 例如: employees.parallelStream()。

### 类加载机制

类加载机制 （javap -c \*.class 生产可读的jvm指令)

**步骤： 加载(load), 链接(link), 初始化(Initialize)**

* Load: 将二进制字节码装载到JVM中， 类的全限定名+ClassLoader完成加载
* Link: 校验， 初始化静态变量赋予默认值，解析类中的调用的接口、类 （NoSuchMethodError, NoSuchFileError), 包括： 校验，准备， 解析(可选）。
* Initialize: 执行类中的静态代码，构造器代码，静态属性的初始化。

触发初始化有: new , 反射调用类的方法， 子类调用初始化，JVM启动指定的初始化类。

加载和链接 将二进制字节码转换为 Class 对象；

初始化 是 在初次主动使用对象前执行， 给静态变量赋值和调用<clinit>()等

### Class Loader 类加载顺序

* Bootstrap class loader
* Extension class loader
* System class loader
* Custom class loader

### Memory Leak & OOM

**内存泄漏**：程序未能释放不再使用的对象（因为强引用存在）, 内存泄漏可能导致内存溢出Out of Memory, 但是Out of Memory不一定是内存泄漏。

JVM在为新的对象分配空间时，内存不足报OutOfMemoryError异常

## JVM调优

**系统调优： CPU、IO (Network, Disk) and JVM内存**

**分布式系统调优：可以使用Zipkin, Skywalking or AppDaymics 识别出有性能问题的微服务。**

* **什么是调优？**
  1. 慢、卡顿， 通过压测定位一个系统的瓶颈
  2. JVM运行过程中产生的各种问题Memory Leak OOM

**JVM tuning mainly involves optimizing the garbage collector for better collection performance** so that applications running on VMs can have a larger throughput while using less memory and experiencing lower latency.

### The primary measures of garbage collection

垃圾收集的主要度量是吞吐量(throughput)和延迟(latency)。

* **Throughput**特定时间内完成的工作量。
* **Latency** is the **responsiveness** of an application. Garbage collection pauses affect the responsiveness of applications. **延迟**是应用程序的响应性。垃圾收集暂停会影响应用程序的响应性。（后代）

在 GC 调优中，GC 导致的应用暂停时间影响系统响应速度，GC 处理线程的 CPU 使用率影响系统吞吐量。

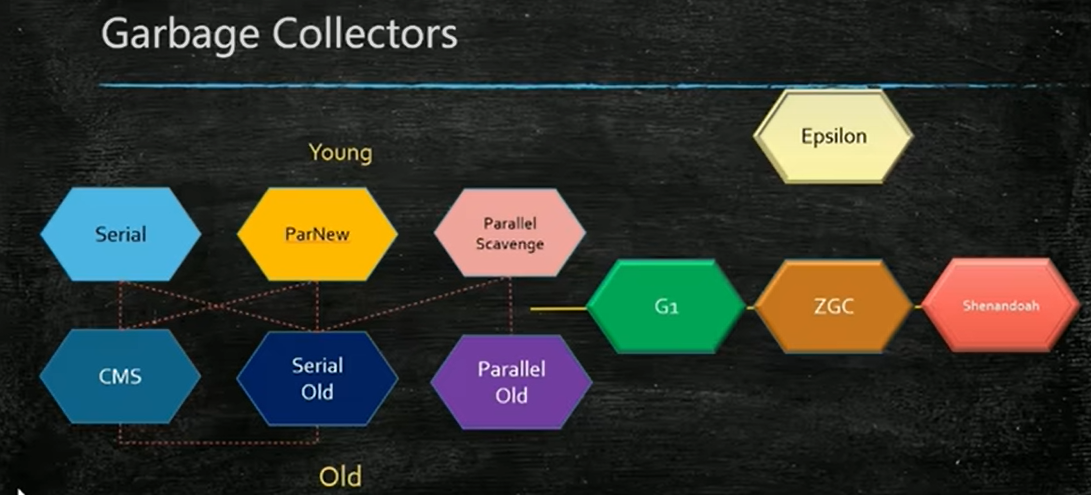
### Garbage Algorithms

|  |  |  |  |
| --- | --- | --- | --- |
| **GC Algorithm** | **Description** | **Advantage** | **Disadvantage** |
| **Mark-Sweep** | 采用从根集合进行扫描，对存活的对象对象标记，标记完毕后，再扫描整个空间中未被标记的对象，进行回收。 | 不需要进行对象的移动，并且仅对不存活的对象进行处理，在存活对象比较多的情况下极为高效 | 直接回收不存活的对象，因此会造成内存碎片  (fragmentation)。 |
| **Copying** |  |  | It consumes more memory for copying |
| **Mark-Compact** |  |  | It is slower |

### Garbage Collectors

随着内存的增大不断的演进。

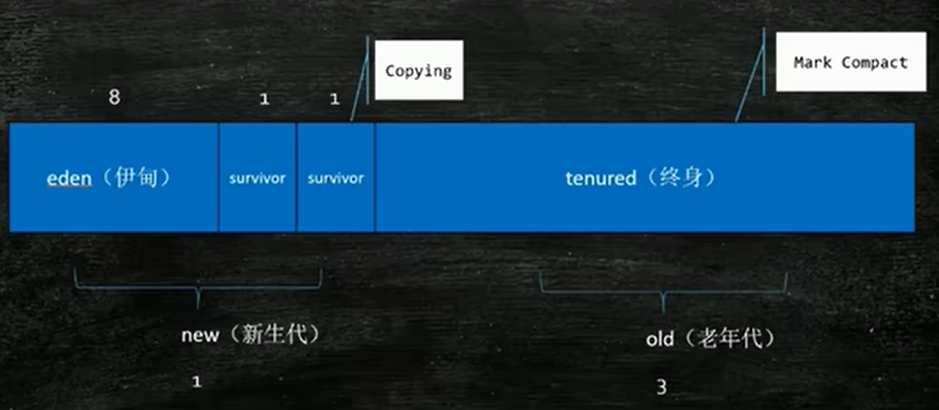
通过命令查看当前使用的垃圾收集器：java -XX:+PrintCommandLineFlags -version



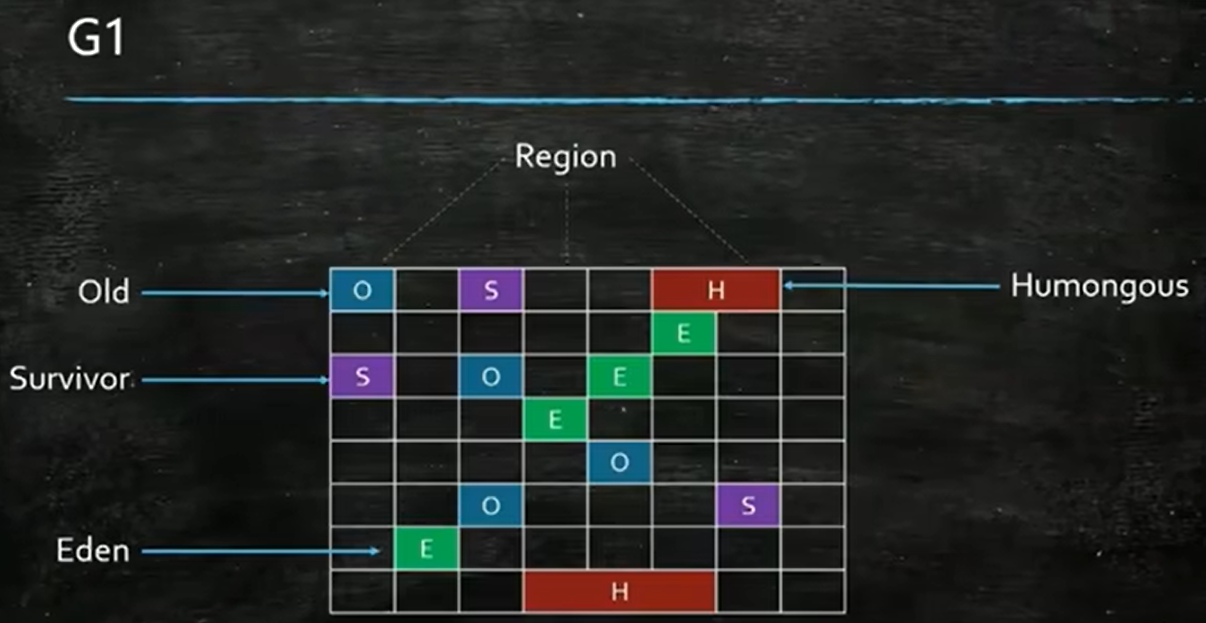
|  |  |
| --- | --- |
| **Serial Collector**  **(分代内存模型)** | **串行收集器**使用单个线程执行所有垃圾收集工作，这使得它相对高效，因为线程之间没有通信开销。  它最适合于单处理器机器，因为它不能利用多处理器硬件，尽管它对于具有小数据集(最多大约100 MB)的多处理器应用程序很有用。在某些硬件和操作系统配置中默认选择串行收集器，或者可以使用选项-XX:+UseSerialGC显式启用串行收集器。 |
| **Parallel Collector**  **(分代内存模型)** | **并行收集器**也称为吞吐量收集器，它是一种类似于串行收集器的分代收集器。串行收集器和并行收集器之间的主要区别是并行收集器有多个线程，用于加速垃圾收集。  并行收集器适用于在多处理器或多线程硬件上运行的具有中型到大型数据集的应用程序。您可以使用-XX:+UseParallelGC选项来启用它。  并行压缩是一个允许并行收集器并行执行主要收集的特性。如果没有并行压缩，主集合将使用单个线程执行，这可能会严重限制可伸缩性。如果指定了选项-XX:+UseParallelGC，则默认启用并行压缩。您可以使用-XX:-UseParallelOldGC选项禁用它。  **Java 8 默认垃圾收集器：Parallel Scavenge + Parallel Old** |
| **Garbage-First (G1) Garbage Collector** | G1是一个**大多数并发的收集器**。大多数并发收集器并发地对应用程序执行一些昂贵的工作。该收集器设计用于从小型机器扩展到具有大量内存的大型多处理器机器。它提供了在实现**高吞吐量**的同时以高概率满足暂停时间目标的能力。  在大多数硬件和操作系统配置中默认选择G1，或者可以使用-XX:+UseG1GC显式启用G1。  将整个Java堆划分成多个大小相等的Region，每个Region都可以扮演Eden、Survivor和老年代空间。 物理不分代，逻辑分代,在G1中除了Region，还有一类特殊的区域被称为Humongous Region，专门存储大对象。在G1中只要超过了一个Region容量的一半的对象就会被称为大对象，而超过整个Region容量的大对象，将会由多个连续的Humongous Region存储。在大多数时Humongous Region会被当做老年代看待。  **Java 9+默认垃圾收集器 G1** |
| **The Z Garbage Collector**  **分页回收，不分代** | Z垃圾收集器(ZGC)是一种可伸缩的**低延迟垃圾收集器**。ZGC并发地执行所有昂贵的工作，而不停止应用程序线程的执行。  ZGC提供了几毫秒的最大暂停时间，但以牺牲一些吞吐量为代价。它适用于需要**低延迟**的应用程序。暂停时间与正在使用的堆大小无关。ZGC支持从8MB到16TB的堆大小。要启用此功能，使用-XX:+UseZGC选项。 |

### GC的内存模型

* **分代模型(Young Generation, Tenured Generation)**



* **G1 的内存模型 （分区region）**



### GC的执行

JVM自动完成垃圾回收 或 手工完成垃圾回收。

* Minor GC: Collecting garbage from Young space
* Major GC： Collecting garbage from Tenured space
* Full GC: Collecting garbage from the entire heap – both Young and Tenured spaces in case of
  + - 年老代（Tenured）被写满
    - 持久代（Perm）被写满
    - 方法区空间不够
    - System.gc()被显示调用 （手工回收），只是通知JVM进行垃圾回收，但是回收时间不确定。（不建议使用）

### GC Tuning Tools (实战)

* **JVM的命令行参数：**<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html>

JVM命令参数分类：

标准参数： -开头，所有的Hotspot都支持

非标准参数： -X开头，特定版本的Hotspot支持特定命令

不稳定参数： -XX开头 如

java -XX:+PrintFlagsFinal

java -XX:+PrintFlagsInitial

java -XX:+PrintCommandLineFlags

* **常用JVM运行参数设置**

java -Xms 200M -Xmx200M -XX:+PrintGC 打印出GC信息，可以看出是否有频繁Full GC

* **垃圾回收器选择**

<https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/collectors.html#sthref28>

* 优先调整**堆**的大小-Xms -Xmx，让JVM自己选择
* 如果内存小于100M, 使用串行收集器
* 如果是单核，并且没有停顿时间要求，使用串行收集器或JVM自己选
* 如果允许停顿时间超过1秒，选择并行收集器或JVM自己选
* 如果响应时间最重要，且不能超过1秒，选择并发收集器(CMS or G1)
* **频繁Full GC的原因**

1. 系统承载高并发请求，或者处理数据量过大，导致Young GC很贫乏，而且每次Young GC过后存活对象太多，内存分配不合理，Survivor区过小，导致对象频繁进入老年代，频繁触发Full GC。
2. 系统一次性加载过多数据进内存，搞出来很多大对象，导致频繁有大对象进入老年带，必然频繁触发Full GC。
3. 系统发生了内存泄漏，莫名其妙创建大量的对象，始终无法回收，一直占用在老年代里，必然频繁触发Full GC。
4. Metaspace（永久代）因为加载类过多触发Full GC。
5. 误调用System.gc()触发Full GC。

**解决方法：**

* 原因1的解决方法是合理分配内存，调大Survivor区；
* 原因2、3，dump出内存快照，用MAT工具进行分析；
* 上述原因都不是，必然是第4、5种原因。
* **常用分析工具**

|  |  |
| --- | --- |
| **top** | top 看哪个进程使用的CPU 和 Memory  top -Hp 看某个进程的所有线程使用的CPU 和 Memory |
| **jps** | 列出java进程 |
| **jinfo** | Java进程的相关的信息 |
| **jstat** | jstat -gc <sid> 500 跟踪内存的变化 |
| **jstack** | 跟踪线程及调用栈, 结合top -Hp使用， VM线程和业务线程 |
| **jmap** | jmap -histo <pid>   1. 查看某种类型对象占用的堆内存 2. 产生堆转储文件，进行分析   jmap -dump:format=b,file=20230504.hprof 生成堆存储文件  生产环境不会直接用jmap， 会卡死。  测试环境压测执行jmap  生产环境JVM参数设置:  java -Xms20M -Xmx20M -XX:+UseParallelGC -XX:+HeapDumpOnOutOfMemoryError OOM时生成堆存储文件  java -Xms100M -Xmx100M -Xlog:gc -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=D:\log com.example.gc.JVMTest1 |
| **jconsole** |  |
| **VisualVM** | 图形界面，可以远程连接   1. 压测环境通过图形界面工具远程连接分析 2. 负载均衡的环境中，将heap dump文件取出，再通过图形界面工具分析 3. 使用tcpcopy将heap dump文件取出，然后用图形界面工具分析 |
| **jprofiler** |  |
| **arthas** |  |

When a standalone application is slow or gets out of memory error, we will consider tuning JVM performance.

JVM performance tuning mainly involves Garbage Collection tuning, known as GC tuning.

What is Garbage Collection in Java?

Garbage collection is the process by which Java programs perform automatic memory management, the garbage collector reclaims the memory of unreferenced objects by deleting them.

What is Garbage Collector?

The garbage collector (GC) automatically manages the application's dynamic memory allocation requests.

The primary measures of garbage collection are throughput and latency.

Throughput is the percentage of total time not spent in garbage collection considered over long periods of time.

Latency is the responsiveness of an application. Garbage collection pauses affect the responsiveness of applications.

The Java HotSpot VM includes three different types of collectors, each with different performance characteristics.

Serial Collector: The serial collector uses a single thread to perform all garbage collection work, It's best-suited to single processor machines because it can't take advantage of multiprocessor hardware, although it can be useful on multiprocessors for applications with small data sets (up to approximately 100 MB). it can be explicitly enabled with the option -XX:+UseSerialGC.

Parallel Collector: The parallel collector is also known as throughput collector, it is a default GC for Java 8, the parallel collector has multiple threads that are used to speed up garbage collection. The parallel collector is intended for applications with medium-sized to large-sized data sets that are run on multiprocessor.

Garbage-First (G1) Garbage Collector: G1 is a mostly concurrent collector, Mostly concurrent collectors perform some expensive work concurrently to the application. This collector is designed to scale from small machines to large multiprocessor machines with a large amount of memory. It provides the capability to meet a pause-time goal with high probability, while achieving high throughput.

The Z Garbage Collector: The Z Garbage Collector (ZGC) is a scalable low latency and fully concurrent collector. ZGC performs all expensive work concurrently, without stopping the execution of application threads.

ZGC provides max pause times of a few milliseconds, but at the cost of some throughput. It is intended for applications, which require low latency. Pause times are independent of heap size that is being used. Java 11 supports Z Garbage collector

Serial Collector and Parallel Collector are generational garbage collector, the heap memory is managed in generations - Young Generation and Old Generation, and Young Generation is divided into Eden spance and two equal Survior spaces.

G1 Collector is a regionalized and generational garbage collector, which means that the Java Object Heap is divided into a number of equally sized regions. the eden, survivor and old generations are logic sets of these regions and are not contiguous.

There are some commands and tools to monitor JVM and Garbage collections: like JConsole, JStat, JStack, VisualVM, JProfiler, and Arthas.

How to do GC turning?

If the JVM recommended collector doesn't achieve the desired performance, then first attempt to adjust the heap and generation sizes to meet the desired goals. If performance is still inadequate, then try a different collector: Use the concurrent collector to reduce pause-time, and use the parallel collector to increase overall throughput on multiprocessor hardware.

OutOfMemoryError

The Java Virtual Machine cannot allocate an object because it is out of memory, it will throw OutOfMemoryError error, the possible reason is memory leak or large number of objects, JVM provides vm parameter to generate Heap Dump file, and we can use tools to anaylize the file and figure out the reason.

## Spring Framework

Spring framework is an open-source lightweight solution for building enterprise applications.

Spring framework has features of

* Core technologies – IoC, AOP, etc.
* Spring MVC
* Testing
* Data Access
* Integration – caching
* Languages

Spring features are **loosely coupled**, so you can use any of these features as necessary to build an application

### Core Technologies

**Core technologies**includes IoC container, AOP, Internationalization and so on.

#### IoC容器和Spring Bean

**IoC**is short for Inversion of Control, also known as dependency injection, it is a process whereby objects define dependencies only through constructor arguments, setter method arguments or properties, the container then inject those dependencies when it creates the bean.

**BeanFactory and ApplicationContext interfaces represent Spring IoC container.**

* BeanFactory provides configuration framework and basic functionality.
* ApplicationContext is a sub-interface of BeanFactory, it adds more functionality such as AOP, internationalization, and event publication and so on.

**Benefits of IoC**

Developers only need to define beans and handle business logic, and Spring IoC container is responsible for instantiating, configuring, assembling, and managing the beans.

**Spring Bean**

A Spring Bean is an object that is instantiated, assembled, and managed by Spring IoC container.

#### Container Overview

Spring IoC container 负责Spring Bean的创建、装配、初始化和管理Bean的整个生命周期。

好处：**解耦**，开发者只需要关心Bean使用。



#### Bean Overview

Spring IoC容器管理一个或多个bean。这些bean是用您提供给容器的配置元数据创建的。

在容器本身内，这些bean定义表示为**BeanDefinition**对象，其中包含以下元数据(以及其他信息):

| Property | Explained in…​ |
| --- | --- |
| Class | [Instantiating Beans](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-class) |
| Name | [Naming Beans](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-beanname) |
| Scope | [Bean Scopes](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-scopes) |
| Constructor arguments | [Dependency Injection](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-collaborators) |
| Properties | [Dependency Injection](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-collaborators) |
| Autowiring mode | [Autowiring Collaborators](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-autowire) |
| Lazy initialization mode | [Lazy-initialized Beans](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-lazy-init) |
| Initialization method | [Initialization Callbacks](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-lifecycle-initializingbean) |
| Destruction method | [Destruction Callbacks](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-lifecycle-disposablebean) |

#### Spring IoC Container Configuration

There are three ways of Spring container configuration.

* **Xml-based configuration**
* **Annotation-based configuration:** @Autowired @Value @Primary @Qualifier (@PostConstruct @Predestory @Resource, @Inject @Named)
* **Java-based configuration:** @Configuration @Bean @Import @Profile @Conditional

From Spring 5, we usually use both annotation-based configuration and java-based configuration.

#### Dependencies

An application has many objects to work together.

There are two major ways of dependency injection.

* Constructor-based dependency injection
* Setter-based dependency injection.

As a rule, we use constructors for mandatory dependencies，setters for optional dependencies. If you want to define an immutable class, then use constructor-based dependency injection to inject its dependencies.

#### Autowiring

The Spring container can autowire relationships between collaborating beans.

You can use @Autowired annotation on

* Fields: 通过Bean后置处理器和反射机制实现, 不需要setter方法。
* Setters
* Constructors

通过类型或名称查找bean，缺省是通过类型byType。

@Autowired实际注入是通过BeanPostProcessor执行的, AutowiredAnnotationBeanPostProcessor, @Vaue注解也是如此。

@Qualifier和@Autowired一起工作，如果同一个类型有多个Bean实例，可以通过@Qualifier指定注入的Bean。

@Primary注解。

#### Bean Scopes

In Spring, beans can be defined with bean scope, there are 6 types of bean scopes:

* Singleton – default scope
* prototype
* request
* session
* application
* websocket

**what are differences between Singleton bean and Prototype bean?**

* If A bean is scoped as a singleton, only one instance is created per Spring IoC Container, this single instance is created when Spring IoC container is initialized, and then cached and reused for all requests.
* Unlike singleton bean, a brand new instance is created on each request for bean with prototype bean per Spring IoC Container.
* Singleton scope bean在Spring IoC Container 启动的时候创建对象。默认Scope是 singleton, 如果加上@lazy注解，则在调用时创建对象。一个Spring IoC容器只有一个实例。
* Prototype scope bean在调用的时候创建对象。一个Spring IoC容器可以有多个实例。
* As a rule, you should use the prototype scope for all stateful beans and the singleton scope for stateless beans. 通常，你应该对有状态的bean（不是线程安全的有状态的Bean）使用prototype scope, 而对无状态的Bean（线程安全的Bean）使用Singlton scope.

**Spring Singleton Bean 与Singleton设计模式**

* Spring Singleton Bean 是Spring IoC容器只为一个Bean创建一个实例，缓存起来，当客户端发起多个请求(多线程) 并发访问同一个Bean，Spring IoC容器不保证这个bean在多线程条件下是线程安全的。换句话说，Bean的类定义时需要考虑线程安全。Spring Singleton Bean 不一定是Singleton class。
* Singleton Design Pattern 保证一个类仅有一个实例，并提供一个访问它的全局访问点。

#### Customizing Bean

Spring IoC container is flexible, it allows you to customize beans through:

* Lifecycle callbacks and annotation like InitializingBean and DisposableBean @PostConstruct, @PreDestory
* ApplicationContextAware, BeanFactoryAware
* Other Aware interfaces
* BeanPostProcessor.

#### Environment Abstraction

Environment接口是集成在容器中的抽象，它对应用程序环境的两个关键方面建模: Profiles和Properties.

* **profile**是一个命名的、逻辑的bean定义组，只有在给定的profile处于活动状态时才向容器注册。@Profile注释允许您指出，当一个或多个指定的概要文件处于活动状态时，组件有资格注册。
* **properties**各种来源: 属性文件、JVM系统属性、系统环境变量等。

可以通过@Value, @ConfigurationProperties 和Environment读取外部配置文件中的属性(properties)

#### Standard and Custom Events

ApplicationContext中的事件处理是通过**ApplicationEvent**类和**ApplicationListener**接口提供的。如果将实现ApplicationListener接口的bean部署到上下文中，则每次将ApplicationEvent发布到ApplicationContext时，都会通知该bean。本质上，这是标准的**Observer设计模式**。

**Spring 内置事件(Spring built-in events):**

* ContextRefreshedEvent
* ContextStartedEvent
* ContextStoppedEvent
* ContextClosedEvent
* RequestHandledEvent
* ServletRequestHandledEvent

**自定义事件和事件监听器以及如何发布事件**

要发布自定义ApplicationEvent，调用ApplicationEventPublisher上的publishEvent()方法。通常，这是通过创建一个实现**ApplicationEventPublisherAware**的类并将其注册为Spring bean来完成的。

**基于注解的事件监听(Annotation-based Event Listeners)**

方法注解**@EventListener**, 可以和@Async和@Order注解一起工作。

#### Spring Bean Life Cycle

Spring IoC container 负责Spring Bean的创建、装配、初始化和管理Bean的整个生命周期。

#### Spring Framework的常用回调(callback)接口

Spring framework 中定义了很多回调（callback） 接口，你可以定义一个类实现这些接口, 注册成bean后，Spring IoC会 自动侦测到这些bean，自动调用它们的方法， 这样大家可以做一些业务逻辑处理。

**常见的需要掌握的回调接口有：**

* BeanPostProcessor
* Aware类型的接口，如 ApplicationContextAware, BeanFractoryAware
* ApplicationListener
* InitializingBean
* DisposableBean
* WebMvcConfigurer
* HandlerInterceptor

#### IoC容器创建过程

参考： AbstractApplicationContext.refresh()

1-4. 【prepareRefresh()】【obtainFreshBeanFactory()】【prepareBeanFactory(beanFactory)】【postProcessBeanFactory(beanFactory)】， 创建和初始化bean factory （DefaultListableBeanFactory）

5. 【invokeBeanFactoryPostProcessors方法】，调用bean factory的后置处理器，根据配置装载注册的bean的定义信息（BeanDefintion)到BeanDefinitionRegistry。

6. 【registerBeanPostProcessors方法】，注册并排序bean后置处理器 （BeanPostProcessor/ BeanDefinitionRegistryPostProcessor）

7. 【initMessageSource方法】，初始化message source

8. 【initApplicationEventMulticaster方法】，初始化事件多播器

ApplicationEventMulticaster是个Helper，代理publish event，

Listener注册到ApplicationEventMulticaster。

9. 【onfresh方法】如果是web 应用， 调用子类ServletWebServerApplicationContext的onfresh()方法, 做两件事：1）创建TomcatWebServer对象, 并启动Tomcat. 2) 注册Servlet (DispatcherServlet) 到ServletContext.

10. 【registerListeners方法】，注册事件监听器到ApplicationEventMulticaster

11. 【finishBeanFactoryInitialization方法】，完成剩余的单实例非lazy加载的bean的实例化

12. 【finishRefresh方法】，完成refresh并发布相关事件

#### Spring AOP

Spring AOP (Aspect Oriented Programming)常用于Transaction Management, Logging, Security.

### Spring MVC

Spring Web MVC是基于Servlet API的Web框架。

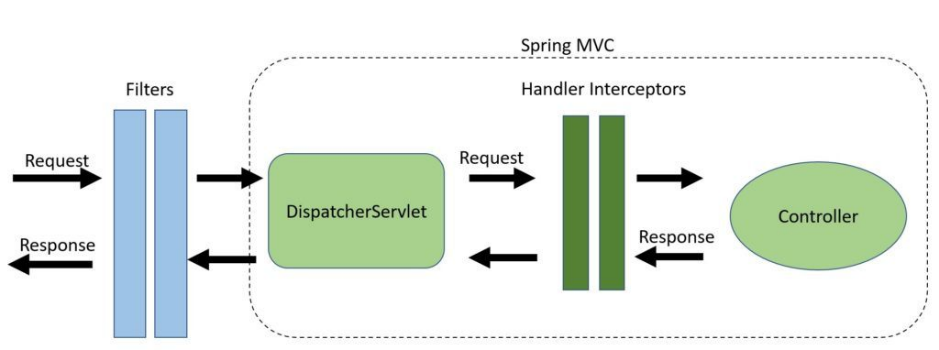
**DispatcherServlet**处理传入的HttpRequest，根据已配置的Handler映射，将请求委托给指定的Handler的方法处理。

**DispatcherServlet执行流程**

**HandlerMapping**接口，由定义requests和处理handler对象之间映射的对象实现。

* BeanNameUrlHandlerMapping
* RequestMappingHandlerMapping: Creates RequestMappingInfo instances from type and method-level @RequestMapping annotations in @Controller classes.
* SimpleUrlHandlerMapping

**如何自定义拦截器？**



**Spring MVC相关的注解**

@Controller

@RestController – 使用Jackson库中的ObjectMapper将Java对象转换为JSON字符串。

@RequestMapping

@GetMapping

@PostMapping

@PutMapping

@DeleteMapping

@PatchMapping

@RequestParam – 方法参数绑定到Web请求参数，query parameter用于过滤、排序和分页

@PathVariable – 方法参数绑定到URI模板变量， path parameter定义资源的位置

@RequestBody

@ReponseBody

**统一异常处理**

@ControllerAdvice和@ ExceptionHandler

### Caching

Spring Framework provides caching abstraction with a set of caching **annotation** and **CacheManager** interface.

Spring Data Redis provides **RedisCacheManager** as an implementation of CacheManager. RedisCacheManager is the CacheManager backed by Redis.

Spring提供了Caching抽象，可以支持Redis or EhCache等cache存储。

* Caching注解
* Caching 管理器接口CacheManager
* Caching 配置

Spring Data Redis项目 提供了CacheManager的实现RedisCacheManager。

## Spring Boot

Spring Boot让你开发基于Spring Framework的应用更容易，因为：

* Auto-configuration
* Starter dependencies
* Embedded application server like Tomcat, Jetty
* External configuration, e.g.: application.properties/ application.yml

### Spring boot自动配置

Spring Boot自动配置尝试根据您添加的jar依赖项自动配置Spring应用程序。例如，如果MySQL在您的类路径上，并且您没有手动配置任何数据库连接bean，那么Spring Boot将自动配置一个内存中的数据库。

你需要通过在你的@Configuration类中添加@EnableAutoConfiguration或@SpringBootApplication注解来选择自动配置。

**禁用特定的自动配置类**

|  |
| --- |
| @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })  public class MyApplication {  } |

### ApplicationRunner or CommandLineRunner

如果需要在SpringApplication启动后运行一些特定的代码，可以实现ApplicationRunner或commandlinerrunner接口。

【SpringApplication启动过程中运行一些特定的代码如何做？】

### 外部配置Externalized Configuration

Spring Boot允许您外部化您的配置，以便您可以在不同的环境中使用相同的应用程序代码。您可以使用各种外部配置源，包括Java属性文件、YAML文件、环境变量和命令行参数。

属性值可以通过使用@Value注释直接注入到bean中，通过Spring的环境抽象进行访问，或者通过@ConfigurationProperties绑定到结构化对象。

**配置文件**

application.properties/ application.yml

application-{profile}.properties/ application-{profile}.yml

**配置文件的优先级**

1. From the classpath
   1. The classpath root
   2. The classpath /config package
2. From the current directory
   1. The current directory
   2. The config/ subdirectory in the current directory
   3. Immediate child directories of the config/ subdirectory

### 日志Logging

Spring Boot使用Commons Logging进行所有内部日志记录，但保留底层日志实现开放。为Java Util Logging、Log4J2和Logback提供了默认配置。在每种情况下，记录器都预先配置为使用控制台输出和可选的文件输出。

默认情况下，如果您使用“starter”，则使用Logback进行日志记录。还包括适当的Logback路由，以确保使用Java Util Logging、Commons Logging、Log4J或SLF4J的依赖库都能正确工作。

### 创建自定义的Starter

<https://docs.spring.io/spring-boot/docs/2.7.11/reference/html/features.html#features.developing-auto-configuration>

* SpringBoot 2.7以前的版本，配置方式为：META-INF/spring.factories，加上@Configuration注解。
* 从SpringBoot 2.7 开始，配置方式为：

META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports, 以及@AutoConfiguration注解。

自动配置的bean 通常是基于条件的，加上@Conditional或@ConditionalOnClass等注解。

【为什么用户自定义的Bean能替代引入的Starter中通过自动配置同类型的Bean？】

|  |
| --- |
| @Configuration  public class RedisConfig {  @Bean  public RedisTemplate<Object, Object> redisTemplate(  RedisConnectionFactory redisConnectionFactory) throws UnknownHostException {  RedisTemplate<Object, Object> template = new RedisTemplate();  template.setConnectionFactory(redisConnectionFactory);  //设置key的序列化方式  template.setKeySerializer(RedisSerializer.string());  template.setHashKeySerializer(RedisSerializer.string());  //设置值的序列化方式  template.setValueSerializer(jsonSerializer());  template.setHashValueSerializer(jsonSerializer());  //更新一下RedisTemplate对象的默认配置  template.afterPropertiesSet();  return template;  }  } |

### 调用REST Services

RestTemplate

### Spring Boot jar直接运行

pom.xml中加人了Spring Boot 插件

|  |
| --- |
| <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin> |

在jar的 MANIFEST.MF中可以相关信息，例如：

|  |
| --- |
| Manifest-Version: 1.0  Archiver-Version: Plexus Archiver  Created-By: Apache Maven 3.6.1  Built-By: lenovo  Build-Jdk: 11.0.18  Main-Class: org.springframework.boot.loader.JarLauncher  Start-Class: com.example.springbootdemo.SpringbootdemoApplication  Spring-Boot-Version: 2.3.7.RELEASE  Spring-Boot-Classes: BOOT-INF/classes/  Spring-Boot-Lib: BOOT-INF/lib/  Spring-Boot-Classpath-Index: BOOT-INF/classpath.idx |

## Spring & Spring Boot Annotation

@ComponentScan

@Component

@Controller @Service

@Configuration

@Bean

@Import

@Autowired

@Value

@Qualifer: if there are more than one bean of the same type, you want to wire one of them, you can use this annotation along with @Autowired annotation.

@Primary

@Profile

@Transactional

@Lazy

@Async

@Order

@DependOn: specify one bean to be created by IoC container before this bean.

@Conditional: indicate that a component is only eligible for registration when all specified conditions match.

Spring MVC相关的注解, 如@RequestMpping

**Spring Boot 注解：**

@SpringBootAppliction

@EnableAutoConfiguraton

@AutoConfiguration

@ConditionalOnXXX

@ConfigurationProperties

## Spring Security

## Spring Cloud

Spring Cloud tools are used to build distributed systems.

## 分布式事务

## 高并发系统设计

## RESTful API Design

OAS 3.0 <https://swagger.io/specification/v3/>

<https://swagger.io/specification/>

example:

<https://editor.swagger.io/?_ga=2.182741158.1248925185.1671853933-1005930888.1670039681>



<https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design>

* HTTP Method：GET, POST, PUT, PATCH, DELETE
* HTTP status： 2XX, 4XX, 5XX
* URI pattern: e.g: /customers/1/orders
* JSON Response design
* RESTful Framework:
* Spring RESTful framework
* Jersey

## Microservice

1. Advantages and disadvantages
2. How to break a Monolith into Microservices 如何划分微服务？

<https://martinfowler.com/articles/break-monolith-into-microservices.html>

1. How to integrate microservices? RESTful API, MQ, RPC

## Code Quality (Test)

Ensure the code quality by code review and testing tools:

* Code review
* Junit：Junit is a testing platform on the JVM
* JMeter: performance test
* Selenium: Java web test
* Postman: Postman is an API platform for building and using APIs
* SolarQube: the leading tool for continuously inspecting the Code Quality and Security of your codebases and guiding development teams during Code Reviews.
* Mock: 测试工具有 Mockito、JMock、EasyMock等等，SpringBoot 目前默认的测试框架是 Mockito 框架。

## Data Security

### JWT

JWT – java web token, 由Header, playload 和signature 组成

### Spring Cloud Gateway + JWT + Role-based authorization (RBAC)

Authentication: 认证

Authorization: 授权

### OAuth2

### SSL

### Encrypt sensitive data

### Prevent code Injection

### SSO

## MySQL

**SQL:** DDL, DML, DCL statements

**Index**

**Query SQL Tuning**

## DevOps

### CICD – Jenkins Pipeline

### Zero downtime

* 蓝绿部署
* 灰度发布(A/B测试)
* 滚动发布

### Managing and monitoring Spring Boot applications

APM – Application Performance Monitoring

* Spring boot admin
* Sleuth + Zipkin
* Skywalking
* AppDynamics

## Cloud

### Cloud providers:

AWS, GCP, Azure, Alibaba, etc.

### Types of cloud services

IaaS, PaaS, SaaS

## 学习书籍推荐

**《重构》**

**《设计模式》**

**《Head First设计模式》**

**《重构与模式》**

《敏捷软件开发原则、模式与实践》

**《Effective Java中文版》**

《Java并发编程实战》

《阿里巴巴Java开发手册 第二册》

《Java 8实战》

《实现领域驱动设计》

**《解构领域驱动设计》**

《中台架构与实现 – 基于DDD和微服务》

《互联网时代的软件革命SaaS架构设计》

《数据仓库工具箱 第3版 维度建模权威指南》

《大数据之路大数据实践 阿里巴巴大数据实践》

**B站的免费视频， 例如 尚硅谷Spring/ Spring Boot/ Spring Cloud/ 大数据**

## 学习方法

* 费曼学习法
* 思维导图学习法