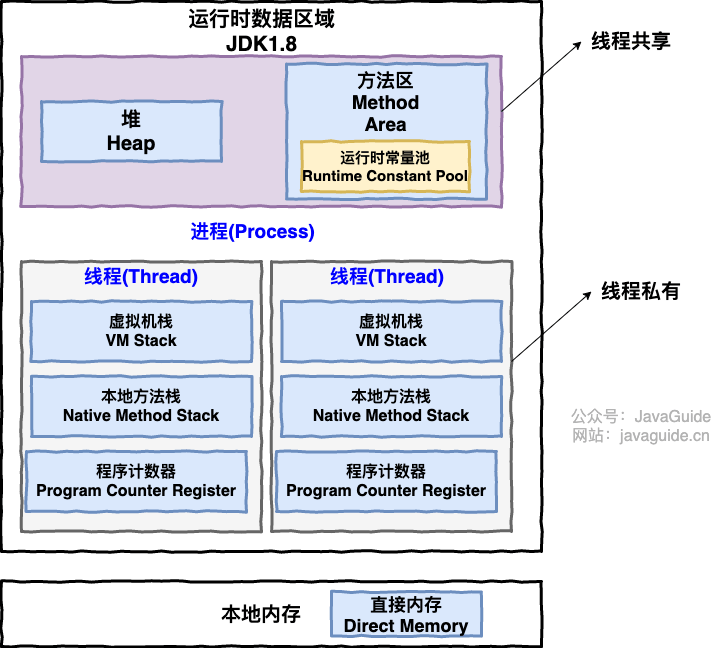
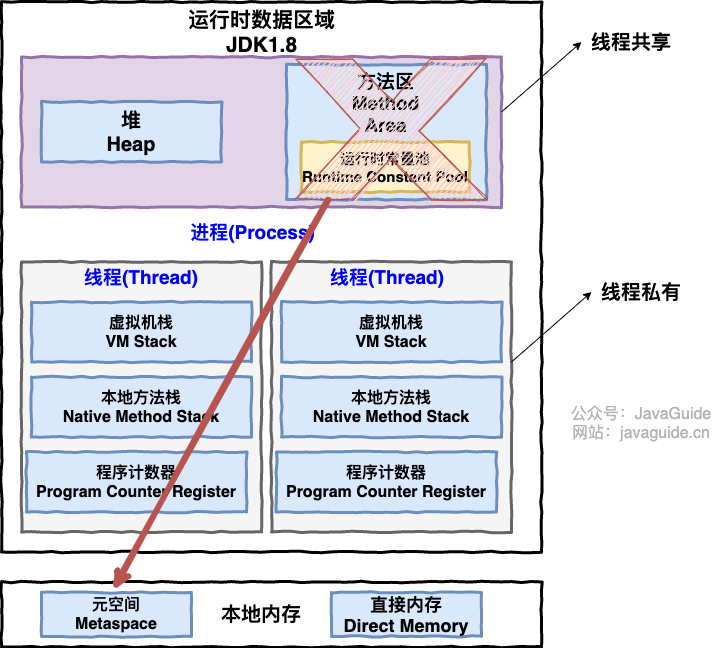
**学习总结**

**JVM**

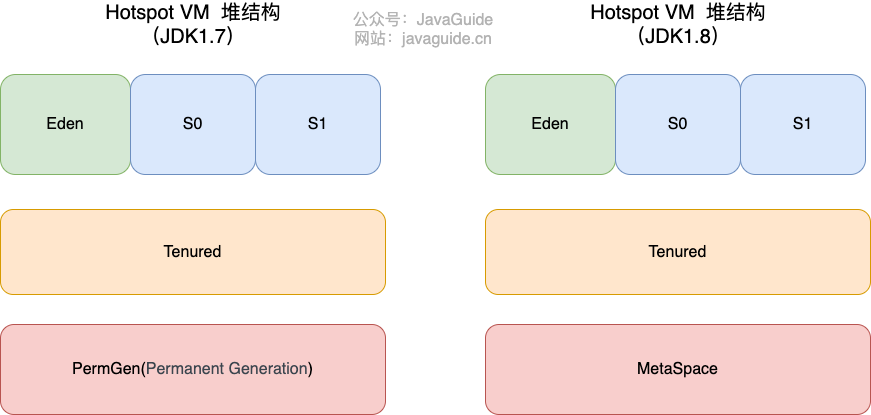
**1.8之前内存模型**



1.8之后的内存模型



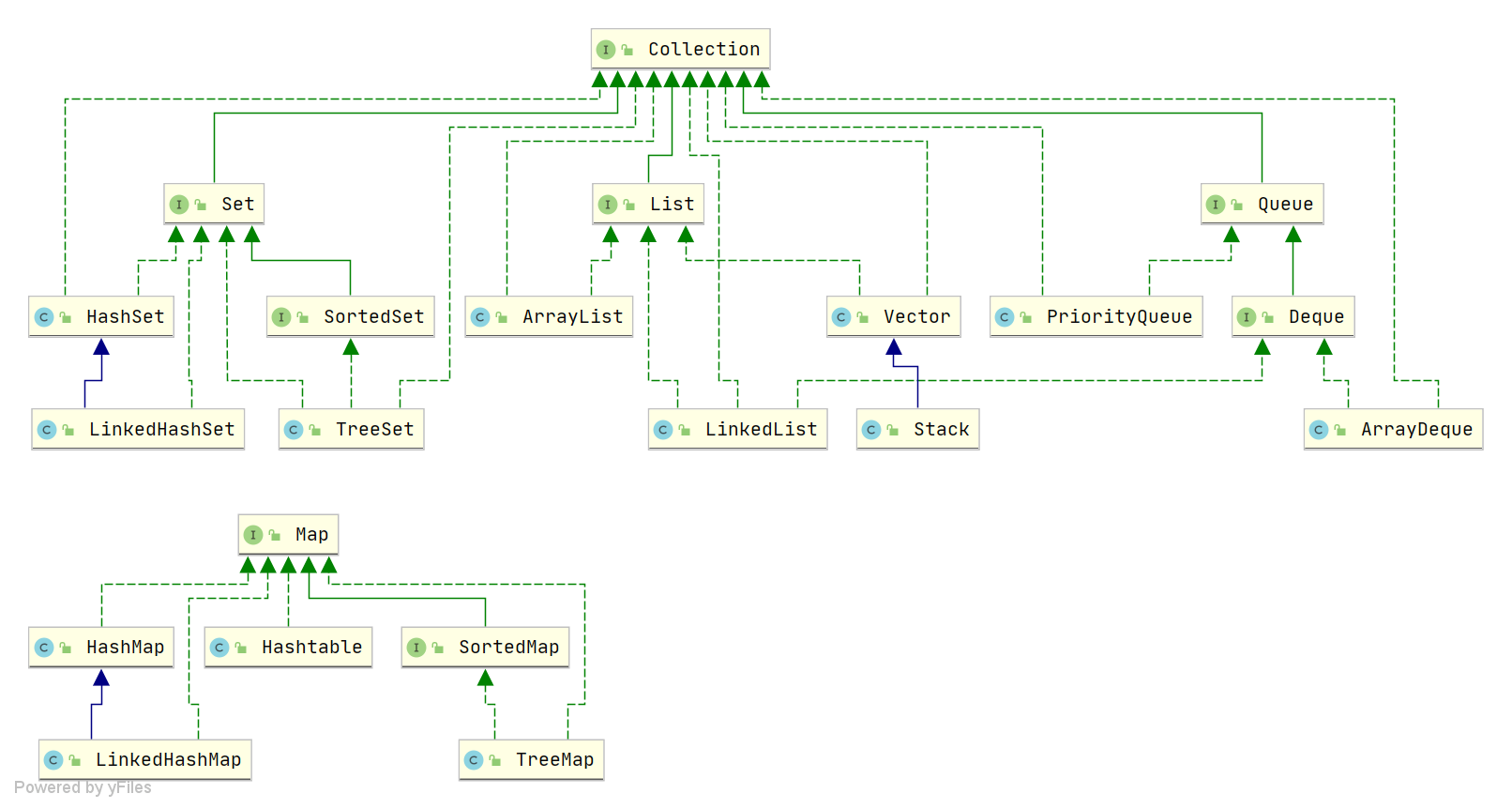
**为啥要用元空间替换永久代**



1. 整个永久代有一个 JVM 本身设置的固定大小上限，无法进行调整，而元空间使用的是直接内存，受本机可用内存的限制，虽然元空间仍旧可能溢出，但是比原来出现的几率会更小。
2. 元空间里面存放的是类的元数据，这样加载多少类的元数据就不由 MaxPermSize 控制了, 而由系统的实际可用空间来控制，这样能加载的类就更多了。

**集合**

Collection接口



**ArrayList 和 Vector 的区别?**

ArrayList 是 List 的主要实现类，底层使用 Object[]存储，适用于频繁的查找工作，线程不安全 ；

Vector 是 List 的古老实现类，底层使用Object[] 存储，线程安全的。

ArrayList 与 LinkedList 区别?

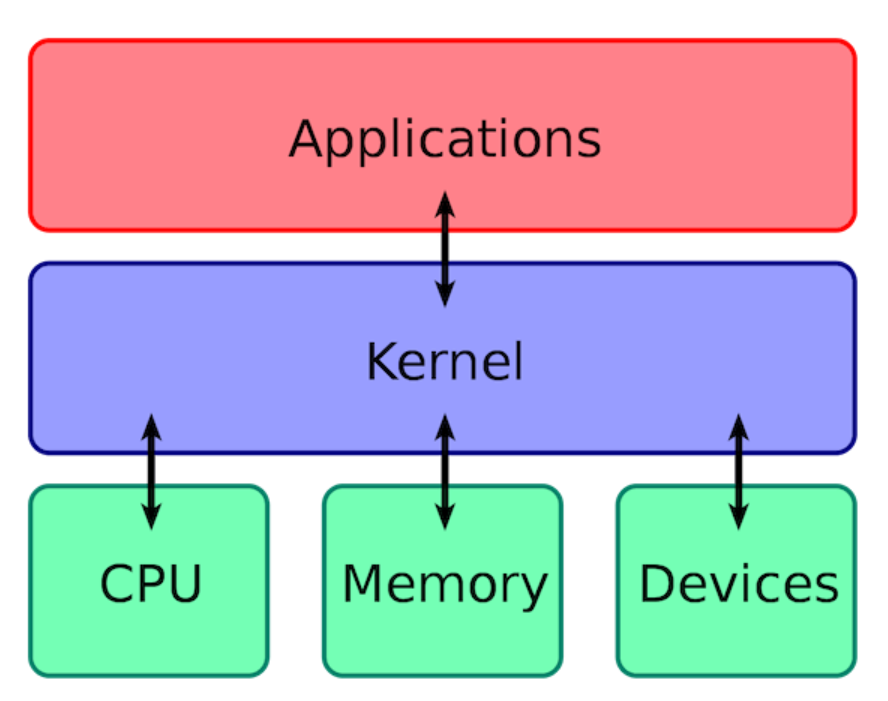
* **是否保证线程安全：** ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；
* **底层数据结构：** ArrayList 底层使用的是 **Object 数组**；LinkedList 底层使用的是 **双向链表** 数据结构（JDK1.6 之前为循环链表，JDK1.7 取消了循环。注意双向链表和双向循环链表的区别，下面有介绍到！）
* **插入和删除是否受元素位置的影响：**
* ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。 比如：执行add(E e)方法的时候， ArrayList 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是 O(1)。但是如果要在指定位置 i 插入和删除元素的话（add(int index, E element)）时间复杂度就为 O(n-i)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向前移一位的操作。
* LinkedList 采用链表存储，所以，如果是在头尾插入或者删除元素不受元素位置的影响（add(E e)、addFirst(E e)、addLast(E e)、removeFirst() 、 removeLast()），时间复杂度为 O(1)，如果是要在指定位置 i 插入和删除元素的话（add(int index, E element)，remove(Object o)）， 时间复杂度为 O(n) ，因为需要先移动到指定位置再插入。
* **是否支持快速随机访问：** LinkedList 不支持高效的随机元素访问，而 ArrayList（实现了RandomAccess接口） 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于get(int index)方法)。
* **内存空间占用：** ArrayList 的空 间浪费主要体现在在 list 列表的结尾会预留一定的容量空间，而 LinkedList 的空间花费则体现在它的每一个元素都需要消耗比 ArrayList 更多的空间（因为要存放直接后继和直接前驱以及数据）。

**LUA脚本语言**

**操作系统篇**

**什么是操作系统**

操作系统是计算机管理软件和硬件的程序，其中操作系统的内核负责系统的内存管理，硬件设备管理，文件系统管理以及应用程序的管理



**什么是系统调用**

系统调用是指调用操作系统提供的系统态级别的子功能，用户的应用程序凡是与系统态级别的资源有关的操作，都必须通过系统调用向操作系统发起请求，由操作系统进行完成。其中，系统态级别资源的操作主要分为设备管理、文件管理、进程控制、进程通信、内存管理

**进程和线程的区别**

**线程是进程划分的更小运行单元**，一个进程中可以有多个线程

线程共享进程中的**堆和方法区/元空间资源**，每个线程有自己的**程序计数器、虚拟机栈和本地方法栈**

进程间是互相独立的，同一进程的线程间会存在相互影响

线程开销小，但不利于资源的管理和保护，进程反之

**进程状态有哪几种**

**创建状态**

**就绪状态**

**运行状态**

**阻塞状态**

**结束状态**

**进程间的通信方式**

**管道/匿名管道**：用于具有亲缘关系的父子进程或兄弟进程之间的通信。存放于内存中

**有名管道**：匿名管道由于没有名字，只能用于亲缘关系的进程间通信。为了克服这个缺点，提出了有名管道。有名管道严格遵循**先进先出**。有名管道以磁盘文件的方式存在，可以实现本机任意两个进程通信。存放于磁盘系统中

**信号**：通知结束进程事件已经发生

**消息队列**：消息队列是消息的链表,具有特定的格式,存放在内存中并由消息队列标识符标识。管道和消息队列的通信数据都是先进先出的原则。存在在内核中。**消息队列克服了信号承载信息量少，管道只能承载无格式字 节流以及缓冲区大小受限等缺点。**

**信号量** ：信号量是一个计数器，用于多进程对共享数据的访问，信号量的意图在于进程间同步。这种通信方式主要用于解决与同步相关的问题并避免竞争条件。

**共享内存**：使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等。可以说这是最有用的进程间通信方式。

**套接字**: 此方法主要用于在客户端和服务器之间通过网络进行通信。套接字是支持 TCP/IP 的网络通信的基本操作单元，可以看做是不同主机之间的进程进行双向通信的端点，简单的说就是通信的两方的一种约定，用套接字中的相关函数来完成通信过程

**线程间的同步方式**

1. **互斥量**：采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以可以保证公共资源不会被多个线程同时访问。比如 Java 中的 synchronized 关键词和各种 Lock 都是这种机制。
2. **信号量**：它允许同一时刻多个线程访问同一资源，但是需要控制同一时刻访问此资源的最大线程数量。
3. **事件** Wait/Notify：通过通知操作的方式来保持多线程同步，还可以方便的实现多线程优先级的比较操作。

操作系统的调度算法

* **先到先服务调度算法** : 从就绪队列中选择一个最先进入该队列的进程为之分配资源，使它立即执行并一直执行到完成或发生某事件而被阻塞放弃占用 CPU 时再重新调度。
* **短作业优先的调度算法** : 从就绪队列中选出一个估计运行时间最短的进程为之分配资源，使它立即执行并一直执行到完成或发生某事件而被阻塞放弃占用 CPU 时再重新调度。
* **时间片轮转调度算法** : 时间片轮转调度是一种最古老，最简单，最公平且使用最广的算法，又称 RR(Round robin)调度。每个进程被分配一个时间段，称作它的时间片，即该进程允许运行的时间。
* **多级反馈队列调度算法** ：前面介绍的几种进程调度的算法都有一定的局限性。如**短进程优先的调度算法，仅照顾了短进程而忽略了长进程** 。多级反馈队列调度算法既能使高优先级的作业得到响应又能使短作业（进程）迅速完成。，因而它是目前**被公认的一种较好的进程调度算法**，UNIX 操作系统采取的便是这种调度算法。
* **优先级调度** ： 为每个流程分配优先级，首先执行具有最高优先级的进程，依此类推。具有相同优先级的进程以 FCFS 方式执行。可以根据内存要求，时间要求或任何其他资源要求来确定优先级。

什么是死锁

**互斥**

资源必须处于非共享模式

**占有并等待**

**非抢占**

**循环等待**

**Spring框架篇**

**什么是Spring**？

Spring是一款开源的轻量级的Java开源框架，旨在提高开发人员的开发效率以及系统的可维护性，Spring是多模块的集合

**Spring包含的模块有哪些？**

**核心模块（Core-Container）**：主要提供Ioc依赖注入功能的支持

* core:Spring框架的核心工具类
* beans:提供对beans创建、配置和管理功能的支持
* context：Context模块构建于核心模块之上，扩展了BeanFactory的功能，包括国际化，资源加载，邮件服务，任务调度等多项功能。ApplicationContext是Context模块的核心接口。
* expression：提供对表达式语言的SqEL的支持，支持设置和获取对象属性，调用对象方法，操作数组、集合等。使用它可以很方便的通过表达式和Spring IoC容器进行交互。

**AOP模块**

Spring AOP模块提供了满足AOP Alliance规范的实现，还整合了AspectJ这种AOP语言级的框架。通过AOP能降低耦合。

**数据访问集成模块（Data Access/Integration）**

该模块包括了JDBC、ORM、OXM、JMS和事务管理：

* 事务模块：该模块用于Spring管理事务，只要是Spring管理对象都能得到Spring管理事务的好处，无需在代码中进行事务控制了，而且支持编程和声明性的事务管理。
* JDBC模块：提供了一个JBDC的样例模板，使用这些模板能消除传统冗长的JDBC编码还有必须的事务控制，而且能享受到Spring管理事务的好处。
* ORM模块：提供与流行的“对象-关系”映射框架的无缝集成，包括hibernate、JPA、MyBatis等。而且可以使用Spring事务管理，无需额外控制事务。
* OXM模块：提供了一个对Object/XML映射实现，将Java对象映射成XML数据，或者将XML数据映射成java对象，Object/XML映射实现包括JAXB、Castor、XMLBeans和XStream。
* JMS模块：用于JMS(Java Messaging Service)，提供一套“消息生产者、消息消费者”模板用于更加简单的使用JMS，JMS用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。

**Web模块**

该模块建立在ApplicationContext模块之上，提供了Web应用的功能，如文件上传、FreeMarker等。Spring可以整合Struts2等MVC框架。此外，Spring自己提供了MVC框架Spring MVC。

**测试模块**

Spring可以用非容器依赖的编程方式进行几乎所有的测试工作，支持JUnit和TestNG等测试框架。

**谈谈对于Spring Ioc的了解**

IOC控制反转：是一种设计思想，将程序中手动创建对象的控制权，交由Spring框架来管理

在Spring当中，将对象的相互依赖关系交给Ioc容器来管理，并由Ioc容器完成对象的注入。这样可以很大程度简化应用的开发，把应用从复杂的依赖关系中解放出来，只需要配置好配置文件或注解 。 IoC 容器是 Spring 用来实现 IoC 的载体， IoC 容器实际上就是个 **Map（key，value）**，Map 中存放的是各种对象。

解决的主要问题：

1、对象之间的耦合或依赖程度降低

2、资源变得容易管理

**将一个类声明为 Bean 的注解有哪些?**

* @Component ：通用的注解，可标注任意类为 Spring 组件。如果一个 Bean 不知道属于哪个层，可以使用@Component 注解标注。
* @Repository : 对应持久层即 Dao 层，主要用于数据库相关操作。
* @Service : 对应服务层，主要涉及一些复杂的逻辑，需要用到 Dao 层。
* @Controller : 对应 Spring MVC 控制层，主要用户接受用户请求并调用 Service 层返回数据给前端页面。

**@Component 和 @Bean 的区别是什么？**

* @Component 注解作用于类，而@Bean注解作用于方法。
* @Component通常是通过类路径扫描来自动侦测以及自动装配到 Spring 容器中（我们可以使用 @ComponentScan 注解定义要扫描的路径从中找出标识了需要装配的类自动装配到 Spring 的 bean 容器中）。@Bean 注解通常是我们在标有该注解的方法中定义产生这个 bean,@Bean告诉了 Spring 这是某个类的实例，当我需要用它的时候还给我。
* @Bean 注解比 @Component 注解的自定义性更强，而且很多地方我们只能通过 @Bean 注解来注册 bean。比如当我们**引用第三方库中的类**需要装配到 Spring容器时，则只能通过 @Bean来实现。

|  |
| --- |
| Java  @Bean注解使用示例  @Configuration  public class AppConfig {  @Bean  public TransferService transferService() {  return new TransferServiceImpl();  }  }  @Bean  public OneService getService(status) {  case (status) {  when 1:  return new serviceImpl1();  when 2:  return new serviceImpl2();  when 3:  return new serviceImpl3();  }  } |

**注入bean的注解有哪些？**

|  |  |  |
| --- | --- | --- |
| Annotaion | Package | Source |
| @Autowired | org.springframework.bean.factory | Spring 2.5+ |
| @Resource | javax.annotation | Java JSR-250 |
| @Inject | javax.inject |  |

**@Autowired 和 @Resource 的区别是什么？**

@Autowired注解属于Spring内置的注解，默认的注入方式是byType(根据类型匹配)，优先根据类型匹配注入bean

|  |
| --- |
| Kotlin  实例  // 报错，byName 和 byType 都无法匹配到 bean  @Autowired  private SmsService smsService;  // 正确注入 SmsServiceImpl1 对象对应的 bean  @Autowired  private SmsService smsServiceImpl1;  // 正确注入 SmsServiceImpl1 对象对应的 bean  // smsServiceImpl1 就是我们上面所说的名称  @Autowired  @Qualifier(value = "smsServiceImpl1")private SmsService smsService; |

建议通过 @Qualifier 注解来显式指定名称而不是依赖变量的名称。

@Resource属于 JDK 提供的注解，默认注入方式为 byName。如果无法通过名称匹配到对应的 Bean 的话，注入方式会变为byType。

如果仅指定 name 属性则注入方式为byName，如果仅指定type属性则注入方式为byType，如果同时指定name 和type属性（不建议这么做）则注入方式为byType+byName。

|  |
| --- |
| Kotlin  // 报错，byName 和 byType 都无法匹配到 bean  @Resource  private SmsService smsService;  // 正确注入 SmsServiceImpl1 对象对应的 bean  @Resource  private SmsService smsServiceImpl1;  // 正确注入 SmsServiceImpl1 对象对应的 bean（比较推荐这种方式）  @Resource(name = "smsServiceImpl1")  private SmsService smsService; |

**Bean的作用域有哪些**

Spring 中 Bean 的作用域通常有下面几种：

* **singleton** : IoC 容器中只有唯一的 bean 实例。Spring 中的 bean 默认都是单例的，是对单例设计模式的应用。
* **prototype** : 每次获取都会创建一个新的 bean 实例。也就是说，连续 getBean() 两次，得到的是不同的 Bean 实例。
* **request** （仅 Web 应用可用）: 每一次 HTTP 请求都会产生一个新的 bean（请求 bean），该 bean 仅在当前 HTTP request 内有效。
* **session** （仅 Web 应用可用） : 每一次来自新 session 的 HTTP 请求都会产生一个新的 bean（会话 bean），该 bean 仅在当前 HTTP session 内有效。
* **application/global-session** （仅 Web 应用可用）： 每个 Web 应用在启动时创建一个 Bean（应用 Bean），该 bean 仅在当前应用启动时间内有效。
* **websocket** （仅 Web 应用可用）：每一次 WebSocket 会话产生一个新的 bean。

**单例 Bean 的线程安全问题了解吗？**

单例 Bean 存在线程问题，主要是因为当多个线程操作同一个对象的时候是存在资源竞争的。

常见的有两种解决办法：

1. 在 Bean 中尽量避免定义可变的成员变量。
2. 在类中定义一个 ThreadLocal 成员变量，将需要的可变成员变量保存在 ThreadLocal 中（推荐的一种方式）。

不过，大部分 Bean 实际都是无状态（没有实例变量）的（比如 Dao、Service），这种情况下， Bean 是线程安全的。

**Bean的生命周期**

**Spring Aop**

AOP：面向切面编程，将与业务无关，却为业务模块所共同调用的逻辑（事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。

**AOP实现**：

如果代理对象实现了某个接口，那么SpringAOP会使用JDK Proxy创建动态代理，对于没有实现接口的对象，会使用Cglib生成一个被代理对象的子类来作为代理

AOP 切面编程设计到的一些专业术语：

|  |  |
| --- | --- |
| 术语 | 含义 |
| 目标(Target) | 被通知的对象 |
| 代理(Proxy) | 向目标对象应用通知之后创建的代理对象 |
| 连接点(JoinPoint) | 目标对象的所属类中，定义的所有方法均为连接点 |
| 切入点(Pointcut) | 被切面拦截 / 增强的连接点（切入点一定是连接点，连接点不一定是切入点） |
| 通知(Advice) | 增强的逻辑 / 代码，也即拦截到目标对象的连接点之后要做的事情 |
| 切面(Aspect) | 切入点(Pointcut)+通知(Advice) |
| Weaving(织入) | 将通知应用到目标对象，进而生成代理对象的过程动作 |

**Spring AOP 和AspectJ AOP**

**Spring AOP属于运行时增强，而AspectJ是编译时增强。**Spring AOP 基于代理，而 AspectJ 基于字节码操作，如果我们的切面比较少，那么两者性能差异不大。但是，当切面太多的话，最好选择 AspectJ ，它比 Spring AOP 快很多。

**AspectJ 定义的通知类型有哪些？**

* **Before**（前置通知）：目标对象的方法调用之前触发
* **After** （后置通知）：目标对象的方法调用之后触发
* **AfterReturning**（返回通知）：目标对象的方法调用完成，在返回结果值之后触发
* **AfterThrowing**（异常通知） ：目标对象的方法运行中抛出 / 触发异常后触发。AfterReturning 和 AfterThrowing 两者互斥。如果方法调用成功无异常，则会有返回值；如果方法抛出了异常，则不会有返回值。
* **Around** （环绕通知）：编程式控制目标对象的方法调用。环绕通知是所有通知类型中可操作范围最大的一种，因为它可以直接拿到目标对象，以及要执行的方法，所以环绕通知可以任意的在目标对象的方法调用前后搞事，甚至不调用目标对象的方法

**多个切面的执行顺序如何控制？**

1、**通常使用@Order 注解直接定义切面顺序**

|  |
| --- |
| Java  // 值越小优先级越高  @Order(3)  @Component  @Aspect  public class LoggingAspect implements Ordered { |

2、**实现Ordered 接口重写 getOrder 方法。**

|  |
| --- |
| Java  @Component  @Aspect  public class LoggingAspect implements Ordered {  // ....  @Override  public int getOrder() {  // 返回值越小优先级越高  return 1;  }  } |

**Spring 框架中用到了哪些设计模式？**

**工厂设计模式**：BeanFactory或ApplicationContext创建Bean对象

**单例设计模式**：默认创建的Bean对象为单例对象

**代理设计模式**：AOP

**模板方法**：一种行为设计模式

**观察者模式**：一种对象行为型模式，它表示的是一种对象与对象之间具有依赖关系，当一个对象发生改变的时候，这个对象所依赖的对象也会做出反应。 Spring 事件驱动模型就是观察者模式很经典的一个应用。Spring 事件驱动模型非常有用，在很多场景都可以解耦我们的代码。比如我们每次添加商品的时候都需要重新更新商品索引，这个时候就可以利用观察者模式来解决这个问题。

**Spring 的事件流程总结**

1. 定义一个事件: 实现一个继承自 ApplicationEvent，并且写相应的构造函数；
2. 定义一个事件监听者：实现 ApplicationListener 接口，重写 onApplicationEvent() 方法；
3. 使用事件发布者发布消息: 可以通过 ApplicationEventPublisher 的 publishEvent() 方法发布消息。

|  |
| --- |
| Java  // 定义一个事件,继承自ApplicationEvent并且写相应的构造函数  public class DemoEvent extends ApplicationEvent{  private static final long serialVersionUID = 1L;  private String message;  public DemoEvent(Object source,String message){  super(source);  this.message = message;  }  public String getMessage() {  return message;  }  // 定义一个事件监听者,实现ApplicationListener接口，重写 onApplicationEvent() 方法；  @Component  public class DemoListener implements ApplicationListener<DemoEvent>{  //使用onApplicationEvent接收消息  @Override  public void onApplicationEvent(DemoEvent event) {  String msg = event.getMessage();  System.out.println("接收到的信息是："+msg);  }  }  // 发布事件，可以通过ApplicationEventPublisher 的 publishEvent() 方法发布消息。  @Component  public class DemoPublisher {  @Autowired  ApplicationContext applicationContext;  public void publish(String message){  //发布事件  applicationContext.publishEvent(new DemoEvent(this, message));  }  } |

**Spring管理事务的方式有哪几种**

**编程式事务**

在代码中硬编码(不推荐使用) : 通过 TransactionTemplate或者 TransactionManager手动管理事务，实际应用中很少使用，但是对于你理解 Spring 事务管理原理有帮助。

**声明式事务**

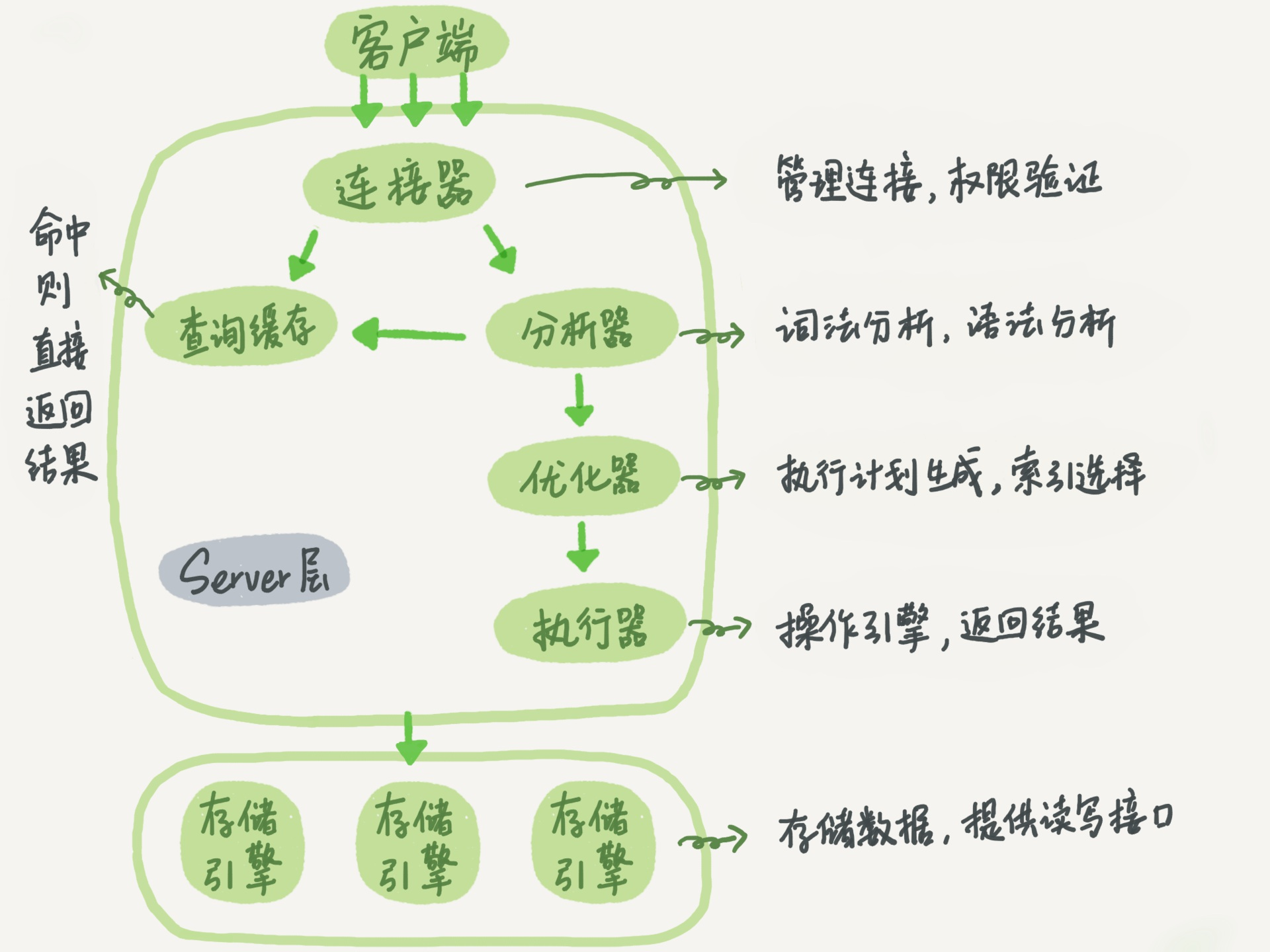
在 XML 配置文件中配置或者直接基于注解（推荐使用） : 实际是通过 AOP 实现（基于@Transactional 的全注解方式使用最多）

**Spring有哪几中事务传播行为**

事务传播行为是为了解决业务层方法之间存在互相调用的事务问题。

当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。

**MySQL**



MySQL可以分Server层和存储引擎层

**Redis**

**redis为什么执行快速**

1、redis基于内存的，内存的访问速度是磁盘的上千倍

2、采用了单线程事件循环和IO多路复用

3、内置了多种优化后的数据结构实现

**redis和memcached的区别和共同点**

**共同点**：

1、都是基于内存的数据库、一般都用来做缓存

2、都有过期策略

3、两者性能都很高

**区别**：

1、redis支持更丰富的数据类型。redis不仅仅支持简单的k/v类型的数据，同时提供了list、set、zset、hash等数据结构的存储；Memcached 只支持最简单的 k/v 数据类型。

2、redis 支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用,而 Memcached 把数据全部存在内存之中。

3、redis有灾难恢复机制。

4、redis 在服务器内存使用完之后，可以将不用的数据放到磁盘上。但是，Memcached 在服务器内存使用完之后，就会直接报异常。

5、memcached 没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据；但是 Redis 目前是原生支持 cluster 模式的。

6、Memcached 是**多线程，非阻塞 IO 复用的网络模型**；Redis 使用**单线程的多路 IO 复用模型**。 （Redis 6.0 引入了多线程 IO ）

7、Redis **支持发布订阅模型、Lua 脚本、事务等功能**，而 Memcached 不支持。并且，Redis 支持更多的编程语言。

8、Memcached 过期数据的删除策略只用了**惰性删除**，而 Redis 同时使用了**惰性删除与定期删除**。

**Redis如何保证高并发和高可用**

主从复制

读写分离，备份，一个Master可以有多个Slaves。

哨兵模式

监控，自动转移，哨兵发现主服务器挂了后，就会从slave中重新选举一个主服务器。

集群模式

为了解决单机Redis容量有限的问题，将数据按一定的规则分配到多台机器，内存/QPS不受限于单机，可受益于分布式集群高扩展性。

<https://blog.csdn.net/weixin_42595012/article/details/89500373>

**Redis其他用途**

* **分布式锁：**通常情况下，我们都是基于 Redisson 来实现分布式锁
* **限流：一般是通过 Redis + Lua 脚本的方式来实现限流。**
* 原子计数器
* 漏桶算法
* 令牌桶算法
* 消息队列
* 统计活跃用户、维护排行榜

**Redis为何6.0之前不用多线程，6.0之后引入多线程**

* 单线程编程容易并且更容易维护；
* Redis 的性能瓶颈不在 CPU ，主要在内存和网络；
* 多线程就会存在死锁、线程上下文切换等问题，甚至会影响性能。

**Redis6.0 引入多线程主要是为了提高网络 IO 读写性能**，因为这个算是 Redis 中的一个性能瓶颈（Redis 的瓶颈主要受限于内存和网络）。虽然，Redis6.0 引入了多线程，但是 Redis 的多线程只是在网络数据的读写这类耗时操作上使用了，执行命令仍然是单线程顺序执行。

**Redis过期策略**

1. **惰性删除** ：**只会在取出 key 的时候才对数据进行过期检查**。这样对 **CPU 最友好**，但是可能会造成太多过期 key 没有被删除。
2. **定期删除** ： **每隔一段时间抽取一批 key 执行删除过期 key 操作**。并且，Redis 底层会通过限制删除操作执行的时长和频率来减少删除操作对 CPU 时间的影响

**Redis内存淘汰机制**

Redis 提供 6 种数据淘汰策略：

1. **volatile-lru（least recently used）**：从已设置过期时间的数据集（server.db[i].expires）中挑选**最近最少使用的数据淘汰**
2. **volatile-ttl**：从已设置过期时间的数据集（server.db[i].expires）中**挑选将要过期的数据淘汰**
3. **volatile-random**：从已设置过期时间的数据集（server.db[i].expires）中**任意选择数据淘汰**
4. **allkeys-lru（least recently used）**：当内存不足以容纳新写入数据时，在键空间中，**移除最近最少使用的 key**（这个是最常用的）
5. **allkeys-random**：从数据集（server.db[i].dict）中**任意选择数据淘汰**
6. **no-eviction**：禁止驱逐数据，也就是说当内存不足以容纳新写入数据时，新写入操作会报错。这个应该没人使用吧！

4.0 版本后增加以下两种：

1. **volatile-lfu（least frequently used）**：从已设置过期时间的数据集（server.db[i].expires）中挑选最不经常使用的数据淘汰
2. **allkeys-lfu（least frequently used）**：当内存不足以容纳新写入数据时，在键空间中，移除最不经常使用的 key

**Redis 持久化机制**

**Redis 的一种持久化方式叫快照（snapshotting，RDB），另一种方式是只追加文件（append-only file, AOF）**

**RDB（默认）**

Redis 创建快照之后，可以对快照进行备份，可以将快照复制到其他服务器从而创建具有相同数据的服务器副本（Redis 主从结构，主要用来提高 Redis 性能），还可以将快照留在原地以便重启服务器的时候使用。

Redis 提供了两个命令来生成 RDB 快照文件：

* save : 主线程执行，会阻塞主线程；
* bgsave : 子线程执行，不会阻塞主线程，默认选项。

**AOF**

与快照持久化相比，AOF 持久化的实时性更好，因此已成为主流的持久化方案。默认情况下 Redis 没有开启 AOF方式的持久化，可以通过 appendonly 参数开启：

|  |
| --- |
| YAML  appendonly yes  appendfsync always #每次有数据修改发生时都会写入AOF文件,这样会严重降低Redis的速度  appendfsync everysec #每秒钟同步一次，显式地将多个写命令同步到硬盘  appendfsync no #让操作系统决定何时进行同步 |

开启 AOF 持久化后**每执行一条会更改 Redis 中的数据的命令**，Redis 就会将该命令**写入到内存缓存** server.aof\_buf 中，然后再根据 appendfsync 配置来决定何时将其**同步到**硬盘中的 AOF 文件。

**AOF重写**

当 AOF 变得太大时，Redis 能够在后台自动重写 AOF 产生一个新的 AOF 文件，这个新的 AOF 文件和原有的 AOF 文件所保存的数据库状态一样，但体积更小。BGREWRITEAOF 命令时，Redis 服务器会维护一个 AOF 重写缓冲区，该缓冲区会在子进程创建新 AOF 文件期间，记录服务器执行的所有写命令。当子进程完成创建新 AOF 文件的工作之后，服务器会将重写缓冲区中的所有内容追加到新 AOF 文件的末尾，使得新的 AOF 文件保存的数据库状态与现有的数据库状态一致。最后，服务器用新的 AOF 文件替换旧的 AOF 文件，以此来完成 AOF 文件重写操作。

**如何选择 RDB 和 AOF？**

关于 RDB 和 AOF 的优缺点，官网上面也给了比较详细的说明[Redis persistence](https://redis.io/docs/manual/persistence/)

**RDB 比 AOF 优秀的地方** ：

* RDB 文件存储的内容是**经过压缩的二进制数据**， 保存着**某个时间点**的数据集，**文件很小**，适合做数据的备份，灾难恢复。AOF 文件存储的是每一次写命令，类似于 MySQL 的 binlog 日志，通常会必 RDB 文件大很多。当 AOF 变得太大时，Redis 能够在后台自动重写 AOF。新的 AOF 文件和原有的 AOF 文件所保存的数据库状态一样，但体积更小。不过， Redis 7.0 版本之前，如果在重写期间有写入命令，AOF 可能会使用大量内存，重写期间到达的所有写入命令都会写入磁盘两次。
* 使用 RDB 文件**恢复数据**，**直接解析还原数据即可**，不需要一条一条地执行命令，速度非常快。而 **AOF 则需要依次执行每个写命令，速度非常慢**。也就是说，与 AOF 相比，恢复大数据集的时候，RDB 速度更快。

**AOF 比 RDB 优秀的地方** ：

* **RDB 的数据安全性不如 AOF，没有办法实时或者秒级持久化数据**。生成 RDB 文件的过程是比繁重的， 虽然 **BGSAVE 子进程写入 RDB 文件的工作不会阻塞主线程，但会对机器的 CPU 资源和内存资源产生影响**，严重的情况下甚至会直接把 Redis 服务干宕机。**AOF 支持秒级数据丢失（取决 fsync 策略，如果是 everysec，最多丢失 1 秒的数据），仅仅是追加命令到 AOF 文件**，操作轻量。
* RDB 文件是以特定的二进制格式保存的，并且在 Redis 版本演进中有多个版本的 RDB，所以存在老版本的 Redis 服务不**兼容新版本**的 RDB 格式的问题。
* AOF 以一种易于理解和解析的格式包含所有操作的日志。你可以轻松地导出 AOF 文件进行分析，你也可以直接操作 AOF 文件来解决一些问题。比如，如果执行FLUSHALL命令意外地刷新了所有内容后，只要 AOF 文件没有被重写，删除最新命令并重启即可恢复之前的状态。

**Redis常用数据结构和应用场景**

**String**

简单动态字符串

**需要存储常规数据的场景**

* 举例 ：缓存 session、token、图片地址、序列化后的对象(相比较于 Hash 存储更节省内存)。
* 相关命令 ： SET、GET。

**需要计数的场景**

* 举例 ：用户单位时间的请求数（简单限流可以用到）、页面单位时间的访问数。
* 相关命令 ：SET、GET、 INCR、DECR

**List**

链表数据结构

**信息流展示**

* 举例 ：最新文章、最新动态。
* 相关命令 ： LPUSH、LRANGE。

**Hash**

String 类型的 field-value（键值对） 的映射表，特别适合用于存储对象

**对象数据存储场景**

* 举例 ：用户信息、商品信息、文章信息、购物车信息。
* 相关命令 ：HSET （设置单个字段的值）、HMSET（设置多个字段的值）、HGET（获取单个字段的值）、HMGET（获取多个字段的值）

**Set**

**需要获取多个数据源交集、并集和差集的场景**

* 举例 ：共同好友(交集)、共同粉丝(交集)、共同关注(交集)、好友推荐（差集）、音乐推荐（差集） 、订阅号推荐（差集+交集） 等场景。
* 相关命令：SINTER（交集）、SINTERSTORE （交集）、SUNION （并集）、SUNIONSTORE（并集）、SDIFF（差集）、SDIFFSTORE （差集）

**Sorted Set**

**需要随机获取数据源中的元素根据某个权重进行排序的场景**

* 举例 ：各种排行榜比如直播间送礼物的排行榜、朋友圈的微信步数排行榜、王者荣耀中的段位排行榜、话题热度排行榜等等。
* 相关命令 ：ZRANGE (从小到大排序) 、 ZREVRANGE （从大到小排序）、ZREVRANK (指定元素排名)。

**Bitmap**

**需要保存状态信息（0/1 即可表示）的场景**

* 举例 ：用户签到情况、活跃用户情况、用户行为统计（比如是否点赞过某个视频）。
* 相关命令 ：SETBIT、GETBIT、BITCOUNT、BITOP。

**HyperLogLog**

**数量量巨大（百万、千万级别以上）的计数场景**

* 举例 ：热门网站每日/每周/每月访问 ip 数统计、热门帖子 uv 统计、
* 相关命令 ：PFADD、PFCOUNT

**Geospatial index**

**需要管理使用地理空间数据的场景**

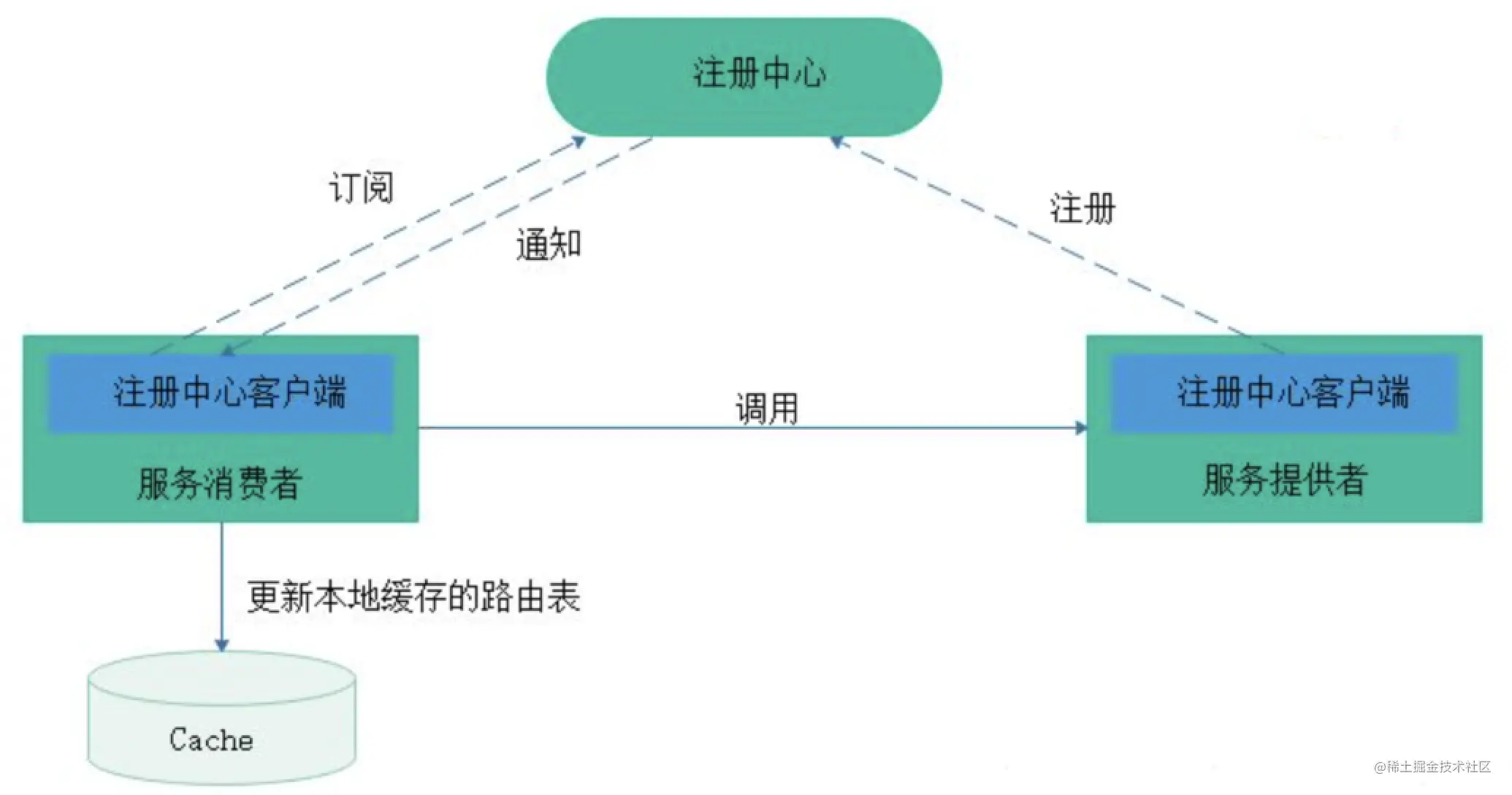
* 举例：附近的人。
* 相关命令: GEOADD、GEORADIUS、GEORADIUSBYMEMBER 。

**注册中心**

服务提供者（RPC Server）

服务消费者（RPC client）

服务注册中心（Registry）



CAP理论是分布式架构中重要理论：

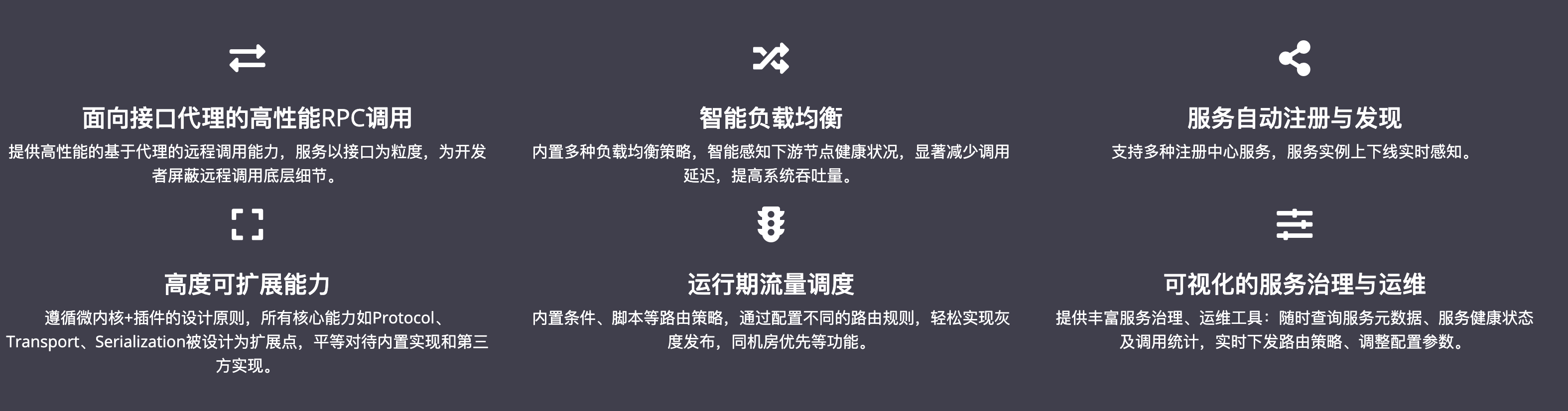
* 一致性(Consistency)：所有节点在同一时间具有相同的数据；
* 可用性(Availability) ：保证每个请求不管成功或者失败都有响应；
* 分隔容忍(Partition tolerance) ：系统中任意信息的丢失或失败不会影响系统的继续运作。

**为啥zookeeper不适合作为注册中心**

在 CAP 模型中，**Zookeeper整体遵循一致性（CP）原则**，即在任何时候对 Zookeeper 的访问请求能得到一致的数据结果，但是当机器下线或者宕机时，**不能保证服务可用性**

当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30 ~ 120s, 且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得zk集群失去master节点是较大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的

**Dubbo**



**Dubbo 的 SPI 机制了解么？ 如何扩展 Dubbo 中的默认实现？**

**注册中心选型**

**关于CP还是AP的选择**：选择 AP，因为可用性高于一致性，所以更倾向 Eureka 和 Nacos；关于Eureka、Nacos如何选择，哪个让我做的事少，我就选择哪个，显然 Nacos 帮我们做了更多的事。

**分布式锁**

**背景**

分布式系统下，不同的服务/客户端通常运行在独立的 JVM 进程上。如果多个 JVM 进程共享同一份资源的话，使用本地锁就没办法实现资源的互斥访问了。于是，**分布式锁** 就诞生了

**满足的条件**

一个最基本的分布式锁需要满足：

* **互斥** ：任意一个时刻，锁只能被一个线程持有；
* **高可用** ：锁服务是高可用的。并且，即使客户端的释放锁的代码逻辑出现问题，锁最终一定还是会被释放，不会影响其他线程对共享资源的访问。
* **可重入**：一个节点获取了锁之后，还可以再次获取锁

**本地缓存**

**Caffeine Cache-高性能Java本地缓存组件**

|  |
| --- |
| Makefile  **Caffeine常用配置说明：**  initialCapacity=[integer]: 初始的缓存空间大小  maximumSize=[long]: 缓存的最大条数  maximumWeight=[long]: 缓存的最大权重  expireAfterAccess=[duration]: 最后一次写入或访问后经过固定时间过期  expireAfterWrite=[duration]: 最后一次写入后经过固定时间过期  refreshAfterWrite=[duration]: 创建缓存或者最近一次更新缓存后经过固定的时间间隔，刷新缓存  weakKeys: 打开key的弱引用  weakValues：打开value的弱引用  softValues：打开value的软引用  recordStats：开发统计功能  注意：  expireAfterWrite和expireAfterAccess同时存在时，以expireAfterWrite为准  maximumSize和maximumWeight不可以同时使用  weakValues和softValues不可以同时使用 |

**缓存淘汰算法**

1. FIFO：先进先出，在这种淘汰算法中，先进入缓存的会先被淘汰，会导致命中率很低。
2. LRU：最近最少使用算法，每次访问数据都会将其放在我们的队尾，如果需要淘汰数据，就只需要淘汰队首即可。仍然有个问题，如果有个数据在 1 分钟访问了 1000次，再后 1 分钟没有访问这个数据，但是有其他的数据访问，就导致了我们这个热点数据被淘汰。
3. LFU：最近最少频率使用，利用额外的空间记录每个数据的使用频率，然后选出频率最低进行淘汰。这样就避免了 LRU 不能处理时间段的问题。

**LFU的局限性**：**在 LFU 中只要数据访问模式的概率分布随时间保持不变时，其命中率就能变得非常高。**比如有部新剧出来了，我们使用 LFU 给他缓存下来，这部新剧在这几天大概访问了几亿次，这个访问频率也在我们的 LFU 中记录了几亿次。但是新剧总会过气的，比如一个月之后这个新剧的前几集其实已经过气了，但是他的访问量的确是太高了，其他的电视剧根本无法淘汰这个新剧，所以在这种模式下是有局限性。

**LRU的优点和局限性**：LRU可以很好的应对突发流量的情况，因为他不需要累计数据频率。但LRU通过历史数据来预测未来是局限的，它会认为最后到来的数据是最可能被再次访问的，从而给与它最高的优先级。

**Caffine Cache 在算法上的优点-W-TinyLFU**

在处理频率记录的方案中，你可能会想到用hashMap去存储，每一个key对应一个频率值。那如果数据量特别大的时候，是不是这个hashMap也会特别大呢。由此可以联想到 Bloom Filter，对于每个key，用n个byte每个存储一个标志用来判断key是否在集合中。原理就是使用k个hash函数来将key散列成一个整数。

**手动加载**

|  |
| --- |
| Java  /\*\*  在每次get key的时候指定一个同步的函数，如果key不存在就调用这个函数生成一个值。  \* 手动加载  \* @param key  \* @return  \*/  public Object manulOperator(String key) {  Cache<String, Object> cache = Caffeine.newBuilder()  .expireAfterWrite(1, TimeUnit.SECONDS)  .expireAfterAccess(1, TimeUnit.SECONDS)  .maximumSize(10)  .build();  //如果一个key不存在，那么会进入指定的函数生成value  Object value = cache.get(key, t -> setValue(key).apply(key));  cache.put("hello",value);  //判断是否存在如果不存返回null  Object ifPresent = cache.getIfPresent(key);  //移除一个key  cache.invalidate(key);  return value;  }  public Function<String, Object> setValue(String key){  return t -> key + "value";  } |

**同步加载**

|  |
| --- |
| Java  /\*\*  \* 同步加载  \* @param key  \* @return  \*/  public Object syncOperator(String key){  LoadingCache<String, Object> cache = Caffeine.newBuilder()  .maximumSize(100)  .expireAfterWrite(1, TimeUnit.MINUTES)  .build(k -> setValue(key).apply(key));  return cache.get(key);  }  public Function<String, Object> setValue(String key){  return t -> key + "value";  } |

**异步加载**

|  |
| --- |
| TypeScript  /\*\*  \* 异步加载  \* @param key  \* @return  \*/  public Object asyncOperator(String key){  AsyncLoadingCache<String, Object> cache = Caffeine.newBuilder()  .maximumSize(100)  .expireAfterWrite(1, TimeUnit.MINUTES)  .buildAsync(k -> setAsyncValue(key).get());  return cache.get(key);  }  public CompletableFuture<Object> setAsyncValue(String key){  return CompletableFuture.supplyAsync(() -> {  return key + "value";  });  } |