1. Classloader

三类类加载器：

Bootstrap ClassLoader ：最顶层的加载类，主要加载核心类库，%JRE\_HOME%\lib下的rt.jar、resources.jar、charsets.jar和class等。另外需要注意的是可以通过启动jvm时指定-Xbootclasspath和路径来改变Bootstrap ClassLoader的加载目录。比如java -Xbootclasspath/a:path被指定的文件追加到默认的bootstrap路径中。我们可以打开我的电脑，在上面的目录下查看，看看这些jar包是不是存在于这个目录。

Extention ClassLoader 扩展的类加载器，加载目录%JRE\_HOME%\lib\ext目录下的jar包和class文件。还可以加载-D java.ext.dirs选项指定的目录。

Appclass Loader也称为SystemAppClass 加载当前应用的classpath的所有类。

加载顺序：

Bootstrap CLassloder

Extention ClassLoader

AppClassLoader

BootstrapClassLoader、ExtClassLoader、AppClassLoader实际是查阅相应的环境属性sun.boot.class.path、java.ext.dirs和java.class.path来加载资源文件

AppClassLoader的parent为ExtClassLoader

Bootstrap ClassLoader是由C/C++编写的，它本身是虚拟机的一部分，所以它并不是一个JAVA类。

一个类加载器查找class和resource时，是通过“委托模式”进行的，它首先判断这个class是不是已经加载成功，如果没有的话它并不是自己进行查找，而是先通过父加载器，然后递归下去，直到Bootstrap ClassLoader，如果Bootstrap classloader找到了，直接返回，如果没有找到，则一级一级返回，最后到达自身去查找这些对象。这种机制就叫做双亲委托

ExtClassLoader的parent为null

自定义ClassLoader步骤

编写一个类继承自ClassLoader抽象类。

复写它的findClass()方法。

在findClass()方法中调用defineClass()。

1. Jvm内存模型

这几个存储区最主要的就是栈区和堆区，那么什么是栈什么是堆呢？说的简单点，栈里面存放的是基本的数据类型和引用，而堆里面则是存放各种对象实例

| **名称** | **特征** | **作用** | **配置** | **异常** |
| --- | --- | --- | --- | --- |
| 栈区 | 线程私有，使用一段连续的内存空间 | 存放局部变量表、操作栈、动态链接、方法出口 | -XSs | StackOverflowError OutOfMemoryError |
| 堆 | 线程共享，生命周期与虚拟机相同 | 保存对象实例 | -Xms -Xmx -Xmn | OutOfMemoryError |
| 程序计数器 | 线程私有、占用内存小 | 字节码行号 | 无 | 无 |
| 方法区 | 线程共享 | 存储类加载信息、常量、静态变量等 | -XX:PermSize -XX:MaxPermSize | OutOfMemoryError |

1. 线程池

ThreadPoolExecutor：

A.ThreadPoolExecutor的重要参数  
    a、corePoolSize：核心线程数  
        \* 核心线程会一直存活，及时没有任务需要执行  
        \* 当线程数小于核心线程数时，即使有线程空闲，线程池也会优先创建新线程处理  
        \* 设置allowCoreThreadTimeout=true（默认false）时，核心线程会超时关闭

    b、queueCapacity：任务队列容量（阻塞队列）  
        \* 当核心线程数达到最大时，新任务会放在队列中排队等待执行

    c、maxPoolSize：最大线程数  
        \* 当线程数>=corePoolSize，且任务队列已满时。线程池会创建新线程来处理任务  
        \* 当线程数=maxPoolSize，且任务队列已满时，线程池会拒绝处理任务而抛出异常

    d、 keepAliveTime：线程空闲时间  
        \* 当线程空闲时间达到keepAliveTime时，线程会退出，直到线程数量=corePoolSize  
        \* 如果allowCoreThreadTimeout=true，则会直到线程数量=0

    e、allowCoreThreadTimeout：允许核心线程超时  
    f、rejectedExecutionHandler：任务拒绝处理器  
        \* 两种情况会拒绝处理任务：  
            - 当线程数已经达到maxPoolSize，切队列已满，会拒绝新任务  
            - 当线程池被调用shutdown()后，会等待线程池里的任务执行完毕，再shutdown。如果在调用shutdown()和线程池真正shutdown之间提交任务，会拒绝新任务  
        \* 线程池会调用rejectedExecutionHandler来处理这个任务。如果没有设置默认是AbortPolicy，会抛出异常  
        \* ThreadPoolExecutor类有几个内部实现类来处理这类情况：  
            - AbortPolicy 丢弃任务，抛运行时异常  
            - CallerRunsPolicy 执行任务  
            - DiscardPolicy 忽视，什么都不会发生  
            - DiscardOldestPolicy 从队列中踢出最先进入队列（最后一个执行）的任务  
        \* 实现RejectedExecutionHandler接口，可自定义处理器

B、ThreadPoolExecutor执行顺序  
        线程池按以下行为执行任务  
    a. 当线程数小于核心线程数时，创建线程。  
    b. 当线程数大于等于核心线程数，且任务队列未满时，将任务放入任务队列。  
    c. 当线程数大于等于核心线程数，且任务队列已满  
        - 若线程数小于最大线程数，创建线程  
        - 若线程数等于最大线程数，抛出异常，拒绝任务

1. JAVA 垃圾回事机制
2. redis 持久化策略

---------rdb:快照形式是直接把内存中的数据保存到一个dump文件中，定时保存，保存策略，配置：

save 900 1

save 300 10

save 60 10000

默认是如上配置：900秒之内，如果超过1个key被修改，则发起快照保存；

300秒内，如果超过10个key被修改，则发起快照保存

1分钟之内，如果1万个key被修改，则发起快照保存

---------aof：每一个写命令都通过write函数追加到appendonly.aof中.Java中的原子性类,配置：appendonly yes

1. lock锁

Lock和ReadWriteLock是两大锁根接口，Lock代表实现类是ReentrantLock（可重入锁），ReadWriteLock（读写锁）的代表实现类是ReentrantReadWriteLock。

1. Eureka集群
2. Eureka元数据

Eureka的元数据有两种：标准元数据和自定义元数据。

标准元数据：主机名、IP地址、端口号、状态页和健康检查等信息，这些信息都会被发布在服务注册表中，用于服务之间的调用。

自定义元数据：可以使用eureka.instance.metadata-map配置

1. 设计模式
2. Java中的流
3. Java中的socket
4. http协议
5. dubbo

调用流程

a.服务容器负责启动，加载，运行服务提供者。

b.服务提供者在启动时，向注册中心注册自己提供的服务。

c.服务消费者在启动时，向注册中心订阅自己所需的服务。

d.注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。

e.服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

f.服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心

1. ActiveMQ
2. Spring mvc流程

具体步骤：

第一步：发起请求到前端控制器(DispatcherServlet)

第二步：前端控制器请求HandlerMapping查找 Handler （可以根据xml配置、注解进行查找）

第三步：处理器映射器HandlerMapping向前端控制器返回Handler，HandlerMapping会把请求映射为HandlerExecutionChain对象（包含一个Handler处理器（页面控制器）对象，多个HandlerInterceptor拦截器对象），通过这种策略模式，很容易添加新的映射策略

第四步：前端控制器调用处理器适配器去执行Handler

第五步：处理器适配器HandlerAdapter将会根据适配的结果去执行Handler

第六步：Handler执行完成给适配器返回ModelAndView

第七步：处理器适配器向前端控制器返回ModelAndView （ModelAndView是springmvc框架的一个底层对象，包括 Model和view）

第八步：前端控制器请求视图解析器去进行视图解析 （根据逻辑视图名解析成真正的视图(jsp)），通过这种策略很容易更换其他视图技术，只需要更改视图解析器即可

第九步：视图解析器向前端控制器返回View

第十步：前端控制器进行视图渲染 （视图渲染将模型数据(在ModelAndView对象中)填充到request域）

第十一步：前端控制器向用户响应结果

1. Jsp 四大作用域

ServletContext context域

HttpServletRequet request域

HttpSession session域 --前三种在学习Servlet时就能接触到

PageContext page域

1. Jsp 九大内置对象

内置对象名 类型

request HttpServletRequest

response HttpServletResponse

config ServletConfig

application ServletContext

session HttpSession

exception Throwable

page Object(this)

out JspWriter

pageContext PageContext

1. Struts2流程
2. Mybatis分页
3. Jdbc实现
4. Spring注入
5. Spring控制反转
6. Spring AOP应用场景

AOP用来封装横切关注点，具体可以在下面的场景中使用:

Authentication 权限

Caching 缓存

Context passing 内容传递

Error handling 错误处理

Lazy loading　懒加载

Debugging　　调试

logging, tracing, profiling and monitoring　记录跟踪　优化　校准

Performance optimization　性能优化

Persistence　　持久化

Resource pooling　资源池

Synchronization　同步

Transactions 事务

1. Spring AOP XML配置

<aop:config>

<aop:pointcut id="loggerCutpoint"

expression=

"execution(\* com.how2java.service.ProductService.\*(..)) "/>

<aop:aspect id="logAspect" ref="loggerAspect">

<aop:after pointcut-ref="loggerCutpoint" method="log"/>

</aop:aspect>

</aop:config>

1. Spring AOP 注解的实现
2. Redis的session共享
3. 分布式锁的实现
4. Threadlocal
5. @Resource、@Autowired、@Qualifier的注解注入及区别

@Autowired和@Qualifier是Spring项目中比较常用的两个注解。一般在同一个实现Bean可以在不同的平台使用，Controller调用Service,Service调用DAO。其中很多实例都是使用@Autowired自动实现，但是如果同一份Bean实例有不同的构造器来创建的时候，使用Autowired自动创建Bean实例IOC容器就分不清了，这时候需要使用@Qualifier来指定特定的bean实例构造方法。

@Autowired可以标注类中引用属性，构造方法，和普通方法。用Autowired标记的属性和方法，都会自动创建实体，当然对于方法中的参数意思是IOC容器自动创建bean实体，并负责给参数列表中的引用了。

@Autowired标记的方法都会在实体创建的时候，自动调用。

@Resource（这个注解属于J2EE的），默认按照名称进行装配，名称可以通过name属性进行指定，如果没有指定name属性，当注解写在字段上时，默认取字段名进行安装名称查找，如果注解写在setter方法上默认取属性名进行装配。当找不到与名称匹配的bean时才按照类型进行装配。但是需要注意的是，如果name属性一旦指定，就只会按照名称进行装配

1. kafka
2. java 原子类

AtomicBoolean,AtomicInteger,AtomicLong,AtomicReference<V>对boolean,Integer,long,reference定义的变量进行原子性的操作，

简单点说就是它提供的方法都是原子性的，不会出现线程之间存在的问题

1. git常用命令

Git commit/pull/push的操作步骤

a.操作步骤需要严格执行如下顺序：commit->pull->push

b.commit：将代码提交到本地仓库。

c.pull：将远程仓库代码同步到本地仓库。如遇冲突，解决冲突，重复commit->pull，直到没有冲突。

d.push:将本地仓库代码提交到远程仓库。

1. 事务的四大特性以及隔离级别

原子性、一致性、分离性、持久性

未提交读（read uncommited） :脏读，不可重复读，虚读都有可能发生

已提交读 （read commited）:避免脏读。但是不可重复读和虚读有可能发生

可重复读 （repeatable read） :避免脏读和不可重复读.但是虚读有可能发生.

串行化的 （serializable） :避免以上所有读问题.

Mysql 默认:可重复读

Oracle 默认:读已提交

---------------------

1. JDK1.5新特性

a：自动装箱与拆箱：

类型包装器有：Double,Float,Long,Integer,Short,Character和Boolean

b：枚举

c：静态导入

import xxxx 和 import static xxxx的区别是前者一般导入的是类文件如import java.util.Scanner;后者一般是导入静态的方法，import static java.lang.System.out。

d：可变参数（Varargs）

e：内省（Introspector）

f：泛型(Generic)

g：For-Each循环

1. JDK1.8新特性

Lambda表达式

函数式接口

\*方法引用和构造器调用

Stream API

接口中的默认方法和静态方法

新时间日期API

Optional容器

1. zokeeper 原理

ZooKeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能

在 ZooKeeper 中，有三种角色（一个 ZooKeeper 集群同一时刻只会有一个 Leader，其他都是 Follower 或 Observer。）：

Leader

Follower

Observer

1. Redis的session共享
2. token的实现（JWT）
3. 缓存框架
4. Jwt的使用

参考：<http://www.ibloger.net/article/3075.html>，https://github.com/X-rapido/jjwt-demo

jwt的组成

Header: 标题包含了令牌的元数据，并且在最小包含签名和/或加密算法的类型

Claims: Claims包含您想要签署的任何信息

JSON Web Signature (JWS): 在header中指定的使用该算法的数字签名和声明

示例

Header:

{

"alg": "HS256",

"typ": "JWT"

}

Claims:

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

Signature:

base64UrlEncode(Header) + "." + base64UrlEncode(Claims)

1. JDK动态代理

// target需要代理的对象，然后通过Proxy.newProxyInstance获取代理对象，

实现InvocationHandler接口进行代理

public class BeanFactory {

public static Object getAgencyObj(final Object target){

Object agencyObj = null;

agencyObj = Proxy.newProxyInstance(

//获取目标对象类的加载器

target.getClass().getClassLoader(),

//获取对象接口的Class对象数组

target.getClass().getInterfaces(),

//一个内部类，用于创建监听对象

new InvocationHandler() {

/\*\*

\* 通过反射机制获得实现类中方法的实例 method 及方法的参数 args 这取决于代理对象 agencyObj调用了那个方法。

\*/

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

int a = (int)args[0];

int b = (int)args[1];

//用于处理次要业务逻辑

if(b == 0){

System.out.println("除数不能为0");

return -1;

}

//用于处理次要业务逻辑

if(a == 0){

return 0;

}

//执行主业务逻辑

Object result = method.invoke(target, args);

return result;

}

});

return agencyObj;

}

}

1. Shiro

Subject

Subject即主体，外部应用与subject进行交互，subject记录了当前操作用户，将用户的概念理解为当前操作的主体，可能是一个通过浏览器请求的用户，也可能是一个运行的程序。 Subject在shiro中是一个接口，接口中定义了很多认证授相关的方法，外部程序通过subject进行认证授，而subject是通过SecurityManager安全管理器进行认证授权

SecurityManager

SecurityManager即安全管理器，对全部的subject进行安全管理，它是shiro的核心，负责对所有的subject进行安全管理。通过SecurityManager可以完成subject的认证、授权等，实质上SecurityManager是通过Authenticator进行认证，通过Authorizer进行授权，通过SessionManager进行会话管理等。

SecurityManager是一个接口，继承了Authenticator, Authorizer, SessionManager这三个接口。

Authenticator

Authenticator即认证器，对用户身份进行认证，Authenticator是一个接口，shiro提供ModularRealmAuthenticator实现类，通过ModularRealmAuthenticator基本上可以满足大多数需求，也可以自定义认证器。

Authorizer

Authorizer即授权器，用户通过认证器认证通过，在访问功能时需要通过授权器判断用户是否有此功能的操作权限。

realm

Realm即领域，相当于datasource数据源，securityManager进行安全认证需要通过Realm获取用户权限数据，比如：如果用户身份数据在数据库那么realm就需要从数据库获取用户身份信息。

注意：不要把realm理解成只是从数据源取数据，在realm中还有认证授权校验的相关的代码。

sessionManager

sessionManager即会话管理，shiro框架定义了一套会话管理，它不依赖web容器的session，所以shiro可以使用在非web应用上，也可以将分布式应用的会话集中在一点管理，此特性可使它实现单点登录。

SessionDAO

SessionDAO即会话dao，是对session会话操作的一套接口，比如要将session存储到数据库，可以通过jdbc将会话存储到数据库。

CacheManager

CacheManager即缓存管理，将用户权限数据存储在缓存，这样可以提高性能。

Cryptography

Cryptography即密码管理，shiro提供了一套加密/解密的组件，方便开发。比如提供常用的散列、加/解密等功能。

1. Dubbo介绍

Consumer服务消费者，Provider服务提供者。Container服务容器。消费当然是invoke提供者了，invoke这条实线按照图上的说明当然同步的意思了，多说一句，在实际调用过程中，Provider的位置对于Consumer来说是透明的，上一次调用服务的位置（IP地址）和下一次调用服务的位置，是不确定的。这个地方就是实现了软负载。

服务提供者先启动start，然后注册register服务。

消费订阅subscribe服务，如果没有订阅到自己想获得的服务，它会不断的尝试订阅。新的服务注册到注册中心以后，注册中心会将这些服务通过notify到消费者。

Monitor这是一个监控，图中虚线表明Consumer 和Provider通过异步的方式发送消息至Monitor，Consumer和Provider会将信息存放在本地磁盘，平均1min会发送一次信息。Monitor在整个架构中是可选的（图中的虚线并不是可选的意思），Monitor功能需要单独配置，不配置或者配置以后，Monitor挂掉并不会影响服务的调用。

1. Dubbo协议

Dubbo协议：

a.传入传出参数数据包较小

b.消费者 比提供者多

c.常规远程服务方法调用

d不适合传送大数据量的服务，比如文件、传视频

Hessian协议

a.提供者比消费者多

b.可传文件

c.跨语言传输

HTTP协议

RMI协议

a.常规RPC调用

b.与原RMI客户端互操作

c.可传文件

d.不支持防火墙穿透

WebService协议

Thrift协议

Memcached协议

Redis协议

1. Dubbo的容错机制

Failover Cluster

　　　　失败自动切换，当出现失败，重试其它服务器 。通常用于读操作，但重试会带来更长延迟。可通过 retries="2" 来设置重试次数(不含第一次)。

　　　　<dubbo:service retries="2" />

　　　　<dubbo:reference retries="2" />

　　　　<dubbo:reference> <dubbo:method name="findFoo" retries="2" /> </dubbo:reference>

　　Failfast Cluster

　　　　快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。

　　Failsafe Cluster

　　　　失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

　　Failback Cluster

　　　　失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。

　　Forking Cluster

　　　　并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2" 来设置最大并行数

　　Broadcast Cluster

　　　　广播调用所有提供者，逐个调用，任意一台报错则报错 。通常用于通知所有提供者更新缓存或日志等本地资源信息。

1. Spring cloud组件介绍

Spring Cloud Config：配置管理开发工具包，可以让你把配置放到远程服务器，目前支持本地存储、Git以及Subversion。

Spring Cloud Bus：事件、消息总线，用于在集群（例如，配置变化事件）中传播状态变化，可与Spring Cloud Config联合实现热部署。

Spring Cloud Netflix：针对多种Netflix组件提供的开发工具包，其中包括Eureka、Hystrix、Zuul、Archaius等。

Netflix Eureka：云端负载均衡，一个基于 REST 的服务，用于定位服务，以实现云端的负载均衡和中间层服务器的故障转移。

Netflix Hystrix：容错管理工具，旨在通过控制服务和第三方库的节点,从而对延迟和故障提供更强大的容错能力。

Netflix Zuul：边缘服务工具，是提供动态路由，监控，弹性，安全等的边缘服务。

Netflix Archaius：配置管理API，包含一系列配置管理API，提供动态类型化属性、线程安全配置操作、轮询框架、回调机制等功能。

Spring Cloud for Cloud Foundry：通过Oauth2协议绑定服务到CloudFoundry，CloudFoundry是VMware推出的开源PaaS云平台。

Spring Cloud Sleuth：日志收集工具包，封装了Dapper,Zipkin和HTrace操作。

Spring Cloud Data Flow：大数据操作工具，通过命令行方式操作数据流。

Spring Cloud Security：安全工具包，为你的应用程序添加安全控制，主要是指OAuth2。

Spring Cloud Consul：封装了Consul操作，consul是一个服务发现与配置工具，与Docker容器可以无缝集成。

Spring Cloud Zookeeper：操作Zookeeper的工具包，用于使用zookeeper方式的服务注册和发现。

Spring Cloud Stream：数据流操作开发包，封装了与Redis,Rabbit、Kafka等发送接收消息。

Spring Cloud CLI：基于 Spring Boot CLI，可以让你以命令行方式快速建立云组件。

1. Fds
2. d