## Linux

Linux系统的启动过程可以分为5个阶段：

1. 内核的引导

当计算机打开电源后，首先是BIOS开机自检，按照BIOS中设置的启动设备（通常是硬盘）来启动。

操作系统接管硬件以后，首先读入 /boot 目录下的内核文件。

1. 运行 init

init 进程是系统所有进程的起点，你可以把它比拟成系统所有进程的老祖宗，没有这个进程，系统中任何进程都不会启动。

init程序的类型：

SysV: init, CentOS 5之前, 配置文件：/etc/inittab。

Upstart: init,CentOS 6, 配置文件：/etc/inittab, /etc/init/\*.conf。

Systemd：systemd, CentOS 7,配置文件：/usr/lib/systemd/system、 /etc/systemd/system。

运行级别：

许多程序需要开机启动。它们在Windows叫做"服务"（service），在Linux就叫做"守护进程"（daemon）。

init进程的一大任务，就是去运行这些开机启动的程序。但是，不同的场合需要启动不同的程序，比如用作服务器时，需要启动Apache，用作桌面就不需要。

Linux允许为不同的场合，分配不同的开机启动程序，这就叫做"运行级别"（runlevel）。也就是说，启动时根据"运行级别"，确定要运行哪些程序。

Linux系统有7个运行级别(runlevel)：

运行级别0：系统停机状态，系统默认运行级别不能设为0，否则不能正常启动

运行级别1：单用户工作状态，root权限，用于系统维护，禁止远程登陆

运行级别2：多用户状态(没有NFS)

运行级别3：完全的多用户状态(有NFS)，登陆后进入控制台命令行模式

运行级别4：系统未使用，保留

运行级别5：X11控制台，登陆后进入图形GUI模式

运行级别6：系统正常关闭并重启，默认运行级别不能设为6，否则不能正常启动

1. 系统初始化

在init的配置文件中有这么一行： si::sysinit:/etc/rc.d/rc.sysinit　它调用执行了/etc/rc.d/rc.sysinit，而rc.sysinit是一个bash shell的脚本，它主要是完成一些系统初始化的工作，rc.sysinit是每一个运行级别都要首先运行的重要脚本。

它主要完成的工作有：激活交换分区，检查磁盘，加载硬件模块以及其它一些需要优先执行任务。

1. 建立终端

rc执行完毕后，返回init。这时基本系统环境已经设置好了，各种守护进程也已经启动了。

init接下来会打开6个终端，以便用户登录系统。在inittab中的以下6行就是定义了6个终端：

1:2345:respawn:/sbin/mingetty tty1

2:2345:respawn:/sbin/mingetty tty2

3:2345:respawn:/sbin/mingetty tty3

4:2345:respawn:/sbin/mingetty tty4

5:2345:respawn:/sbin/mingetty tty5

6:2345:respawn:/sbin/mingetty tty6

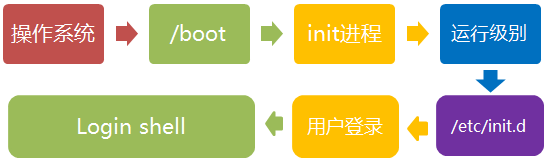
1. 用户登录系统

一般来说，用户的登录方式有三种：

（1）命令行登录

（2）ssh登录

（3）图形界面登录



## 轻量级与重量级框架

SSH轻量级框架与EJB重量级框架解决问题的侧重点

轻量级框架  
1、侧重于减小开发的复杂度，相应的处理能力有所减弱（事务功能较弱，不具备分布式处理能力），比较适合开发中小型企业应用。

2、尽可能采取基于POJO的方法进行开发，是应用不依赖于任何容器。这样可以提高开发效率。

3、轻量级框架多为开源项目，提供了良好的涉及和许多快速构建工具以及大量现成可供参考的开源代码。

重量级框架

1. 强调高可伸缩性，是和大型企业应用。
2. 在EJB体系结构中，一切与基础结构服务相关的问题和底层分配问题都由应用程序容器或服务器来处理。
3. EJB容器铜鼓哦减少数据库访问次数以及分布式处理等方式提供了专门的系统性能解决方案。

EJB最初的[设计思想](https://www.baidu.com/s?wd=%E8%AE%BE%E8%AE%A1%E6%80%9D%E6%83%B3&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)考虑的是为分布式的应用服务的，分布式是针对大型应用构造的跨平台的协作计算，EJB最初的目的就是为这种计算服务的。但是软件发展到目前为止，大多数应用不需要采用分布式的解决方案，因此用EJB显得太臃肿了。Spring的出现恰恰为了解决这个问题。

轻量级容器"越来越成为Java世界的讨论关键词之一。那么到底什么是"轻量级"容器?

在你浏览最近的有关J2EE的文章或者blog，都有意无意的提到了"lightweigh containter",

伴随着个词出现最多的就是"IoC","dependencies inject","AOP"等等。

何谓"容器"?

说穿了没什么特别的地方，而且在我们的世界里面司空见惯。**容器就是一组提供一系列服务的管理器**，只要你符合容器的服务要求(规范)

容器就可以让你使用范围内的管理服务. 早在Web流行的时候，http container就为我们提供解析Html的能力，

让我们的html代码可以通过http协议来发布到internet.随着web应用的推广，动态语言的发展，http容器逐步

可以用指定的接口来解释特殊文件中的特殊片断，如:php,asp等等。那么让我们揭开容器的盖子，看看里面都有什么(服务).

1,生命周期管理

- 既然要管理运行的组件(或程序片断)，至少可以通过某种方法在需要它的时候创建，没用的时候销毁。

2,查询定位

- 通过一定的策略(通常用文件名称)确定需要调用哪个组件或者资源，这可是作为管理器的主要工作。

3,配置器

- 提供静态配置策略。让用户可以控制服务的细节部分。如，资源在哪里等等

4,依赖解析

- 如果组件之间存在关系，或者需要相互通讯。那么至少要保证他们都是"活"的。这就需要分析出之间的依赖关系，动态管理。

5,不同协议层的通讯支持

- 对于程序片断不是自己就可以完成所有的计算的，还需要和不同的系统(平台)进行通讯，如用JDBC和数据存储数据库，用RMI分布组件

6,扩展支持

-扩展服务项不仅是容器需要的，也是任何软体需要的。比如说，增加安全策略。

何谓"轻量级"?

既然是"轻"，那就是比较"重"的而言."重"的典型代表就是EJB,EJB提供了一系列"重量级"企业级服务，并可以让你开发的组件可以很好的

集成EJB容器所提供的企业级服务，如JTA等。那么既然有了"要嘛有嘛"的服务容器，为什么还需要"轻量级"呢?这是一个好问题。

对于全面的EJB容器，虽然给了我们看起来完整的服务策略，但是，EJB不是雷锋，它也给我们带来了许多负面效果。有过EJB经验的人们

深有感触：

1，部署复杂，运行缓慢

2，内在服务多，启动慢

3, 规则特多，空间很小

4，难预测试(调试)

...

EJB容器的服务往往是"买一送三"，不要都不行。我只需要JTA，而真的不需要JMS.

所以，在这个"对我没用的都是垃圾"的世界里，我们就需要"可选择型"的服务容器，让那些对我们没用的服务不要站在这里碍事。

可选择性就是轻量级容器的目标之一。

"轻量级"在哪里?

容器的服务既然可以DIY了，那么确实看上去"轻"了许多。但是，事情并非如此简单。

在开始之前，先来回顾一下几个关键词

1，封装

Java本身在照顾到非java程序员接受程度和配合市场宣传的"running anywhere"，不得不将一些

非面向对象的特性语法加入其中。"是否破坏了封装?"，称为Java十年来争议最多的问题。

2，组件式开发

这个词在软件的圈子里显得技术含量很高，但是，在硬件范围里，哪个硬件不是组件式的呢？你的计算机里就是各式各样的组件，

他们靠各个接口(插槽)集成工作。甚至你正在看得数字电视机顶盒也是一个标准的组件。

3，Running Anywhere

这个词伴随着一杯黑咖啡进入我们的世界，虽然现在已经变成了"Debug Anywhere",但是，这一直是我们的目标。

轻量级容器就是以这几个为目标的解决方案。

面对封装，组件粒度一直是SA的最大问题。根据EJB模型，EJB组件让我们把业务组件封装成粗粒度的业务组件。

轻量级容器这可以定义更为细粒度的组件，甚至这个组件只有一个对象.以依赖注入(Dependency Injection,DI)为代表的解耦模式，

可以让组件不去依赖容器(运行环境)的API。DI作为容器的管道，承担中间人的角色，让使用者(component)和提供者毫无关联。

对于以何种粒度去设计业务组件，这就是业务的具体需求所决定的，任何框架都帮不了你。

而轻量级容器提供技术方面的支持也绝对让你有的放矢。

面对组件，真正的组件就像你手里的U-disk，插到USB上就可以享受其功能。

轻量级容器通过反向控制(Inversion of Control,IoC)让容器具有主动权，去管理插进来的组件。只要组件是符合标准的，就可以被

轻量级容器管理。

面对Running Anywhere,轻量级容器让组件以POJO的形式存在，只要你有java.exe就可以运行它。

这样，组件根本就是一个独立的功能集成类，根本不需要容器就可以实现测试行为。

以上三点，给我们带来的好处不仅如此，我们伴随着还将重用，生产力，测试(质量)，移植性等各个方面受益，这里就不多啰嗦了。

轻量级容器的现状

目前，轻量级容器在open source范围已经得到了广泛的认可，而且，作为下一代J2EE构架的基础有着无法比拟的优势。

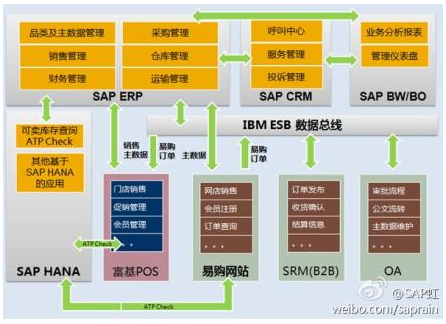
当前，已存在的轻量级容器框架中，有绝大部分来自open source社群，这给我们对这项技术的了解降低了门槛。

## SOA

资源调度和治理中心(SOA)

首先Martin Fowler提出SOA歧义Service Oriented Ambiguity，认为"什么是SOA"是不可能回答，因为不同的人意味着不同的事情，SOA意味服务接口，意味流程整合，意味资源再利用，意味着管制

在实际应用中，很多单位使用SOA主要看中其能够无缝整合新旧系统,称为EAI企业应用整合，下图是苏宁的一种SOA图，使用ESB企业服务总线这样的消息系统整合了新旧各种系统。



SOA的提出是在企业计算领域，就是要将紧耦合的系统，划分为面向业务的，粗粒度，松耦合，无状态的服务。服务发布出来供其他服务调用，一组互相依赖的服务就构成了SOA架构下的系统。

微服务与SOA相比，更强调分布式系统的特性，比如横向伸缩性，服务发现，负载均衡，故障转移，高可用。互联网开发对服务治理提出了更多的要求，比如多版本，比如灰度升级，比如服务降级，比如分布式跟踪，这些都是在SOA实践中重视不够的。

## 正向代理与反向代理

正向代理（forward proxy） ，一个位于客户端和原始服务器之间的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并制定目标（原始服务器），然后代理向原始服务器转发请求并将获得的内容返回给客户端，客户端才能使用正向代理。我们平时说的代理就是指正向代理。

反向代理（Reverse Proxy），以代理服务器来接受internet上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给internet上请求的客户端，此时代理服务器对外表现为一个反向代理服务器。

    理解起来有些抽象，可以这么说：A向B借钱，B没有拿自己的钱，而是悄悄地向C借钱，拿到钱之后再交给A,A以为是B的钱，他并不知道C的存在。

正向代理和反向代理的区别：

位置不同

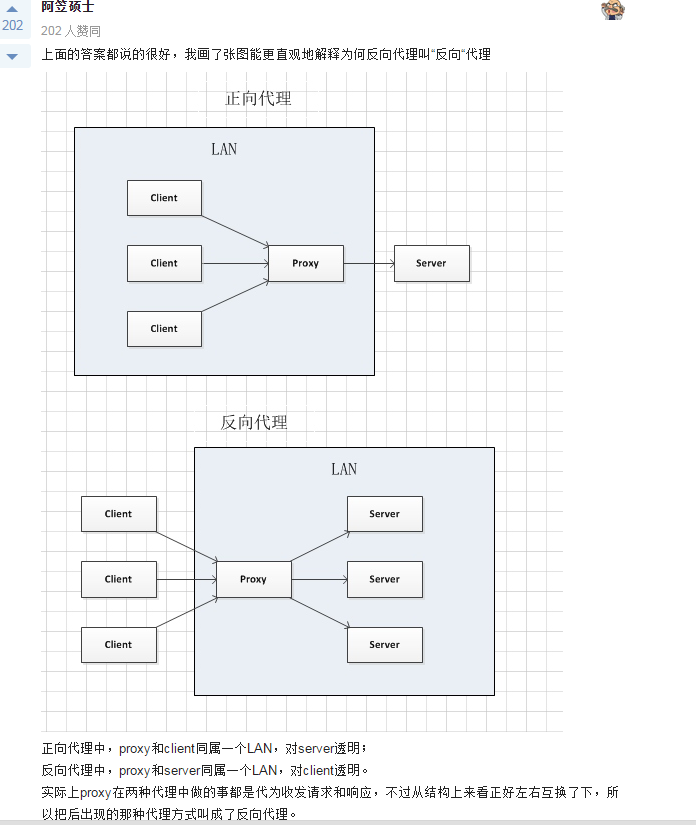
正向代理，架设在客户机和目标主机之间；

反向代理，架设在服务器端；

代理对象不同

正向代理，代理客户端，服务端不知道实际发起请求的客户端；

反向代理，代理服务端，客户端不知道实际提供服务的服务端；



## 长连接和短连接

HTTP协议是无状态的，指的是协议对于事务处理没有记忆能力，服务器不知道客户端是什么状态。也就是说，打开一个服务器上的网页和上一次打开这个服务器上的网页之间没有任何联系。HTTP是一个无状态的面向连接的协议，无状态不代表HTTP不能保持TCP连接，更不能代表HTTP使用的是UDP协议（无连接）。

短连接

连接->传输数据->关闭连接

比如HTTP是无状态的的短链接，浏览器和服务器每进行一次HTTP操作，就建立一次连接，但任务结束就中断连接。

长连接

连接->传输数据->保持连接 -> 传输数据-> ………..->直到一方关闭连接，多是客户端关闭连接。 长连接指建立SOCKET连接后不管是否使用都保持连接，但安全性较差。

具体网络中的应用的话：http 1.0一般就指短连接，smtp,pop3,telnet这种就可以认为是长连接。一般的网络游戏应用都是长连接

长连接多用于操作频繁，点对点的通讯，而且连接数不能太多情况，。每个TCP连接都需要三步握手，这需要时间，如果每个操作都是先连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，次处理时直接发送数据包就OK了，不用建立TCP连接。例如：数据库的连接用长连接， 如果用短连接频繁的通信会造成socket错误，而且频繁的socket 创建也是对资源的浪费。

而像WEB网站的http服务一般都用短链接，因为长连接对于服务端来说会耗费一定的资源，而像WEB网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源，如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。所以并发量大，但每个用户无需频繁操作情况下需用短连好。

## 项目上线流程

www和@的区别：www与正常的三w对应，@就是不输入三w时的。

HTTP协议如何工作：



## 身份验证

**传统身份验证的方法**

HTTP 是一种没有状态的协议，也就是它并不知道是谁是访问应用。这里我们把用户看成是客户端，客户端使用用户名还有密码通过了身份验证，不过下回这个客户端再发送请求时候，还得再验证一下。

解决的方法就是，当用户请求登录的时候，如果没有问题，我们在服务端生成一条记录，这个记录里可以说明一下登录的用户是谁，然后把这条记录的 ID 号发送给客户端，客户端收到以后把这个 ID 号存储在 Cookie 里，下次这个用户再向服务端发送请求的时候，可以带着这个 Cookie ，这样服务端会验证一个这个 Cookie 里的信息，看看能不能在服务端这里找到对应的记录，如果可以，说明用户已经通过了身份验证，就把用户请求的数据返回给客户端。

上面说的就是 Session，我们**需要在服务端存储为登录的用户生成的 Session** ，这些 Session 可能会存储在内存，磁盘，或者数据库里。我们可能需要在服务端定期的去清理过期的 Session 。

**基于 Token 的身份验证方法**

使用基于 Token 的身份验证方法，**在服务端不需要存储用户的登录记录**。大概的流程是这样的：

1. 客户端使用用户名跟密码请求登录
2. 服务端收到请求，去验证用户名与密码
3. 验证成功后，服务端会签发一个 Token，再把这个 Token 发送给客户端
4. 客户端收到 Token 以后可以把它存储起来，比如放在 Cookie 里或者 Local Storage 里
5. 客户端每次向服务端请求资源的时候需要带着服务端签发的 Token
6. 服务端收到请求，然后去验证客户端请求里面带着的 Token，如果验证成功，就向客户端返回请求的数据

**JWT**

实施 Token 验证的方法挺多的，还有一些标准方法，比如 JWT，读作：jot ，表示：JSON Web Tokens 。JWT 标准的 Token 有三个部分：

header

payload

signature

中间用点分隔开，并且都会使用 Base64 编码，所以真正的 Token 看起来像这样：

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJuaW5naGFvLm5ldCIsImV4cCI6IjE0Mzg5NTU0NDUiLCJuYW1lIjoid2FuZ2hhbyIsImFkbWluIjp0cnVlfQ.SwyHTEx\_RQppr97g4J5lKXtabJecpejuef8AqKYMAJc

header 部分主要是两部分内容，一个是 Token 的类型，另一个是使用的算法，比如下面类型就是 JWT，使用的算法是 HS256。

{

"typ": "JWT",

"alg": "HS256"

}

上面的内容要用 Base64 的形式编码一下，所以就变成这样：

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

Payload 里面是 Token 的具体内容，这些内容里面有一些是标准字段，你也可以添加其它需要的内容。下面是标准字段：

iss：Issuer，发行者

sub：Subject，主题

aud：Audience，观众

exp：Expiration time，过期时间

nbf：Not before

iat：Issued at，发行时间

jti：JWT ID

比如下面这个 Payload ，用到了 iss 发行人，还有 exp 过期时间。另外还有两个自定义的字段，一个是 name ，还有一个是 admin 。

{

"iss": "ninghao.net",

"exp": "1438955445",

"name": "wanghao",

"admin": true

}

使用 Base64 编码以后就变成了这个样子：

eyJpc3MiOiJuaW5naGFvLm5ldCIsImV4cCI6IjE0Mzg5NTU0NDUiLCJuYW1lIjoid2FuZ2hhbyIsImFkbWluIjp0cnVlfQ

Signature

Signature

JWT 的最后一部分是 Signature ，这部分内容有三个部分，先是用 Base64 编码的 header.payload ，再用加密算法加密一下，加密的时候要放进去一个 Secret ，这个相当于是一个密码，这个密码秘密地存储在服务端。

* header
* payload
* secret

var encodedString = base64UrlEncode(header) + "." + base64UrlEncode(payload);

HMACSHA256(encodedString, 'secret');

处理完成以后看起来像这样：

SwyHTEx\_RQppr97g4J5lKXtabJecpejuef8AqKYMAJc

最后这个在服务端生成并且要发送给客户端的 Token 看起来像这样：

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJuaW5naGFvLm5ldCIsImV4cCI6IjE0Mzg5NTU0NDUiLCJuYW1lIjoid2FuZ2hhbyIsImFkbWluIjp0cnVlfQ.SwyHTEx\_RQppr97g4J5lKXtabJecpejuef8AqKYMAJc

客户端收到这个 Token 以后把它存储下来，下回向服务端发送请求的时候就带着这个 Token 。服务端收到这个 Token ，然后进行验证，通过以后就会返回给客户端想要的资源。

## 跨域通信和跨页面通信

由于浏览器同源策略，凡是发送请求url的协议、域名、端口三者之间任意一与当前页面地址不同即为跨域。

几种处理方式：

1. JSONP

这种方式主要是通过动态插入一个script标签。浏览器对script的资源引用没有同源限制，同时资源加载到页面后会立即执行（没有阻塞的情况下）。

实际项目中JSONP通常用来获取json格式数据，这时前后端通常约定一个参数callback，该参数的值，就是处理返回数据的函数名称。

缺点：

1.这种方式无法发送post请求（这里）

2.另外要确定jsonp的请求是否失败并不容易，大多数框架的实现都是结合超时时间来判定。

1. Proxy代理

这种方式首先将请求发送给后台服务器，通过服务器来发送请求，然后将请求的结果传递给前端。

这种方式首先将请求发送给后台服务器，通过服务器来发送请求，然后将请求的结果传递给前端。

还需要注意一点，对于同一请求浏览器通常会从缓存中读取数据，我们有时候不想从缓存中读取，所以会加一个preventCache参数，这个时候请求url变成：url?preventCache=12345567....;这本身没有什么问题，问题出在当使用某些前端框架（比如jquery）发送proxy代理请求时，请求url为proxy?url，同时设置preventCache：true，框架不能正确处理这个参数，结果发出去的请求变成proxy?url&preventCache=123456（正长应为proxy?url?preventCache=12356）;后端截取后发送的请求为url&preventCache=123456，根本没有这个地址，所以你得不到正确结果。

1. Cors（聚宝台）

这是现代浏览器支持跨域资源请求的一种方式。

CORS是一个W3C标准，全称是"跨域资源共享"（Cross-origin resource sharing）。

它允许浏览器向跨源服务器，发出XMLHttpRequest请求，从而克服了AJAX只能同源使用的限制。

OPTIONS请求头部中会包含以下头部：Origin、Access-Control-Request-Method、Access-Control-Request-Headers，发送这个请求后，服务器可以设置如下头部与浏览器沟通来判断是否允许这个请求。

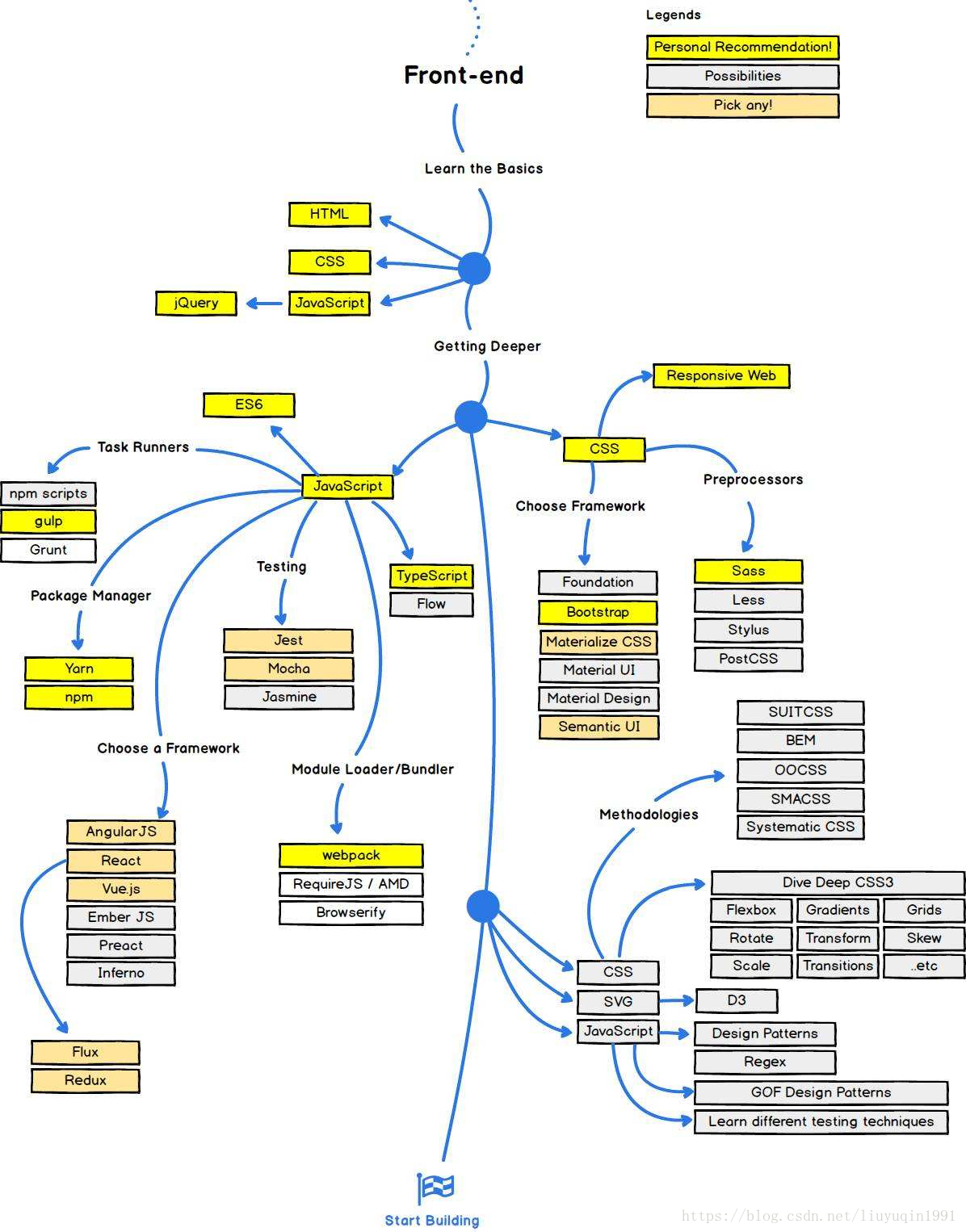
1. xdr

这是IE8、IE9提供的一种跨域解决方案，功能较弱只支持get跟post请求，而且对于协议不同的跨域是无能为力的，比如在http协议下发送https请求。

有一种跨域需要特别注意就是在https协议下发送https请求，除了使用proxy代理外其他方法都无解，会被浏览器直接block掉。

## 前端

### 知识图谱



### MVVM框架

随着前端页面越来越复杂，用户对于交互性要求也越来越高，想要写出Gmail这样的页面，仅仅用jQuery是远远不够的。MVVM模型应运而生。

MVVM最早由微软提出来，它借鉴了桌面应用程序的MVC思想，在前端页面中，把Model用纯JavaScript对象表示，View负责显示，两者做到了最大限度的分离。

把Model和View关联起来的就是ViewModel。ViewModel负责把Model的数据同步到View显示出来，还负责把View的修改同步回Model。

ViewModel如何编写？需要用JavaScript编写一个通用的ViewModel，这样，就可以复用整个MVVM模型了。

一个MVVM框架和jQuery操作DOM相比有什么区别？

我们先看用jQuery实现的修改两个DOM节点的例子：

<!-- HTML -->

<p>Hello, <span id="name">Bart</span>!</p>

<p>You are <span id="age">12</span>.</p>

Hello, Homer!

You are 51.

用jQuery修改name和age节点的内容：

var name = 'Homer';

var age = 51;

$('#name').text(name);

$('#age').text(age);

如果我们使用MVVM框架来实现同样的功能，我们首先并不关心DOM的结构，而是关心数据如何存储。最简单的数据存储方式是使用JavaScript对象：

var person = {

name: 'Bart',

age: 12

};

我们把变量person看作Model，把HTML某些DOM节点看作View，并假定它们之间被关联起来了。

要把显示的name从Bart改为Homer，把显示的age从12改为51，我们并不操作DOM，而是直接修改JavaScript对象：

Hello, Homer!

You are 51.

person.name = 'Homer';

person.age = 51;

执行上面的代码，我们惊讶地发现，改变JavaScript对象的状态，会导致DOM结构作出对应的变化！这让我们的关注点从如何操作DOM变成了如何更新JavaScript对象的状态，而操作JavaScript对象比DOM简单多了！

这就是MVVM的设计思想：关注Model的变化，让MVVM框架去自动更新DOM的状态，从而把开发者从操作DOM的繁琐步骤中解脱出来！

目前，常用的MVVM框架有：

Angular：Google出品，名气大，但是很难用；

Backbone.js：入门非常困难，因为自身API太多；

Ember：一个大而全的框架，想写个Hello world都很困难。

我们选择MVVM的目标应该是入门容易，安装简单，能直接在页面写JavaScript，需要更复杂的功能时又能扩展支持。

所以，综合考察，最佳选择是尤雨溪大神开发的MVVM框架：Vue.js

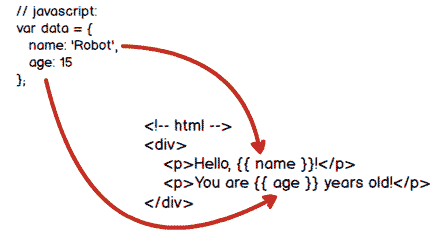
### 双向数据绑定

**单向绑定：**

MVVM框架会自动监听Model的任何变化，在Model数据变化时，更新View的显示。这种Model到View的绑定我们称为单向绑定。

MVVM就是在前端页面上，应用了扩展的MVC模式，我们关心Model的变化，MVVM框架自动把Model的变化映射到DOM结构上，这样，用户看到的页面内容就会随着Model的变化而更新。

例如，我们定义好一个JavaScript对象作为Model，并且把这个Model的两个属性绑定到DOM节点上：



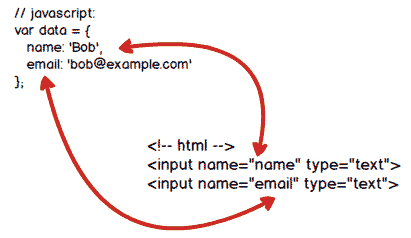
经过MVVM框架的自动转换，浏览器就可以直接显示Model的数据了

**双向绑定：**

单向绑定非常简单，就是把Model绑定到View，当我们用JavaScript代码更新Model时，View就会自动更新。

有单向绑定，就有双向绑定。如果用户更新了View，Model的数据也自动被更新了，这种情况就是双向绑定。

什么情况下用户可以更新View呢？填写表单就是一个最直接的例子。当用户填写表单时，View的状态就被更新了，如果此时MVVM框架可以自动更新Model的状态，那就相当于我们把Model和View做了双向绑定：



在浏览器中，当用户修改了表单的内容时，我们绑定的Model会自动更新

双向绑定最大的好处是我们不再需要用jQuery去查询表单的状态，而是直接获得了用JavaScript对象表示的Model。

指的是vue实例中的data与其渲染的DOM元素的内容保持一致，无论谁被改变，另一方会相应的更新为相同的数据。这是通过设置属性访问器实现的。

就是UI视图与数据绑定在了一块，也就是数据和视图是同步改变的，双向数据绑定最常见的应用场景就是表单。

脏检查（Angular.js原理）

访问器监听（Vue.js原理）

### 闭包

闭包就是能够读取其他函数内部变量的函数。例如在javascript中，只有函数内部的子函数才能读取[局部变量](https://baike.baidu.com/item/%E5%B1%80%E9%83%A8%E5%8F%98%E9%87%8F/9844788)，所以闭包可以理解成“定义在一个[函数](https://baike.baidu.com/item/%E5%87%BD%E6%95%B0/301912)内部的函数“。在本质上，闭包是将函数内部和函数外部连接起来的桥梁。

### Node.js

Node.js: allows to package an application with all of its dependencies into a standardized unit.

Node.js 就是运行在服务端的 JavaScript。Node.js 是一个基于Chrome JavaScript   
运行时建立的一个平台。Node.js是一个事件驱动I/O服务端JavaScript环境，基于Google的V8引擎。相当与JDK。

### NPM（node package manager）

是 [Node.js](http://lib.csdn.net/base/nodejs) 的包管理和分发工具。它类似于[PHP](http://lib.csdn.net/base/php)的Composer，Ruby的gem，[Python](http://lib.csdn.net/base/python)的pip，[Java](http://lib.csdn.net/base/java)的Maven……它可以让 [JavaScript](http://lib.csdn.net/base/javascript) 开发者能够更加轻松的共享代码和共用代码片段，并且通过npm管理你分享的代码也很方便快捷和简单。

### Gulp

是基于Nodejs的自动任务运行器， 能自动化地完成   
javascript、coffee、sass、less、html/image、css   
等文件的测试、检查、合并、压缩、格式化、浏览器自动刷新、部署文件生成，并监听文件在改动后重复指定的这些步骤。在实现上，它借鉴了Unix操作系统的管道（pipe）思想，前一级的输出，直接变成后一级的输入，使得在操作上非常简单。

### 单页面Web应用

一、定义

单页 Web 应用 (single-page application 简称为 SPA) 是一种特殊的 Web 应用。它将所有的活动局限于一个Web页面中，仅在该Web页面初始化时加载相应的HTML、JavaScript 和 CSS。一旦页面加载完成了，SPA不会因为用户的操作而进行页面的重新加载或跳转。取而代之的是利用 JavaScript 动态的变换HTML的内容，从而实现UI与用户的交互。由于避免了页面的重新加载，SPA 可以提供较为流畅的用户体验。

二、路由

在这样的应用中，我们可以看到单页面应用的基本元素： 页面路由，通过某种方式，如 URL hash 来说明表明当前所在的页面，并拥有从一个页面跳转到另外一个页面的入口。

我们可以在 Angular 应用、React 应用、Vue.js 应用 看到这些基本要素的影子，如：Vue Router、React Router、Angular 2 RouterModule 都是负责路由（页面跳转及模块关系）的。在 Vue 和 React 里，它们都是由辅助模块来实现的。因为 React 只是层 UI 层，而 Vue.js 也是用于构建用户界面的框架。

当我们做后台应用的时候，我们只需要关心上述过程中的最后一步。即，将对应的路由交给对应的函数来处理。这一点，在不同的后台框架的表现形式都是相似的。

如 Python 语言里的 Web 开发框架 Django 的 URLConf，使用正规表达式来表正

url(r'^articles/2003/$', views.special\_case\_2003),

而在 Laravel 里，则是通过参数的形式来呈现

Route::get('posts/{post}/comments/{comment}', function ($postId, $commentId) {

//

});

虽然表现形式有一些差别，但是总体来说也是差不多的。而对于前端应用来说，也是如此，**将对应的 URL 的逻辑交由对应的函数来处理。**

React Router 使用了类似形式来处理路由，代码如下所示：

<Route path="blog" component={BlogList} />

<Route path="blog/:id" component={BlogDetail} />

当页面跳转到 blog 的时候，会将控制权将给 BlogList 组件来处理。

当页面跳转到 blog/fasfasf-asdfsafd 的时候，将匹配到这二个路由，并交给 BlogDetail 组件 来处理。而路由中的 id 值，也将作为参数 BlogDetail 组件来处理。

相似的，而 Angular 2 的形式则是：

{ path: 'blog', component: BlogListComponent },

{ path: 'blog/:id', component: BlogDetailComponent },

相似的，这里的 BlogDetailComponent 是一个组件，path 中的 id 值将会传递给 BlogDetailComponent 组件。

从上面来看，尽管表现形式上有所差异，但是其行为是一致的：**使用规则引擎来处理路由与函数的关系。**稍有不同的是，后台的路由完全交由服务器端来控制，而前端的请求则都是在本地改变其状态。

并且同时在不同的前端框架上，他们在行为上还有一些区别。这取决于我们是否需要后台渲染，即刷新当前页面时的表现形式。

* 使用 Hash （#）或者 Hash Bang （#!） 的形式。即 # 开头的参数形式，诸如 ued.party/#/blog。当我们访问 blog/12 时，URL 的就会变成 ued.party/#/blog/12
* 使用新的 HTML 5 的 history API。用户看到的 URL 和正常的 URL 是一样的。当用户点击某个链接进入到新的页面时，会通过 history 的 pushState 来填入新的地址。当我们访问 blog/12 时，URL 的就会变成 ued.party/blog/12。当用户刷新页面的时候，请通过新的 URL 来向服务器请求内容。

幸运的是，大部分的最新 Router 组件都会判断是否支持 history API，再来决定先用哪一个方案。

三、优缺点

单页Web程序的出现是富客户端发展的必然结果，但是该技术也是有些局限性，所以采用之前需要了解清楚它的优缺点。

1、优点：

1).良好的交互体验

用户不需要重新刷新页面，获取数据也是通过Ajax异步获取，页面显示流畅。

2).良好的前后端工作分离模式

单页Web应用可以和RESTful规约一起使用，通过REST API提供接口数据，并使用Ajax异步获取，这样有助于分离客户端和服务器端工作。更进一步，可以在客户端也可以分解为静态页面和页面交互两个部分。

3).减轻服务器压力

服务器只用出数据就可以，不用管展示逻辑和页面合成，吞吐能力会提高几倍；

4).共用一套后端程序代码

不用修改后端程序代码就可以同时用于Web界面、手机、平板等多种客户端；

2、缺点：

1).SEO（Search Engine Optimization:汉译为搜索引擎优化）难度较高

由于所有的内容都在一个页面中动态替换显示，所以在SEO上其有着天然的弱势，所以如果你的站点对SEO很看重，且要用单页应用，那么就做些静态页面给搜索引擎用吧。

2).前进、后退管理

由于单页Web应用在一个页面中显示所有的内容，所以不能使用浏览器的前进后退功能，所有的页面切换需要自己建立堆栈管理，当然此问题也有解决方案，比如利用URI中的散列+iframe实现。

3).初次加载耗时多

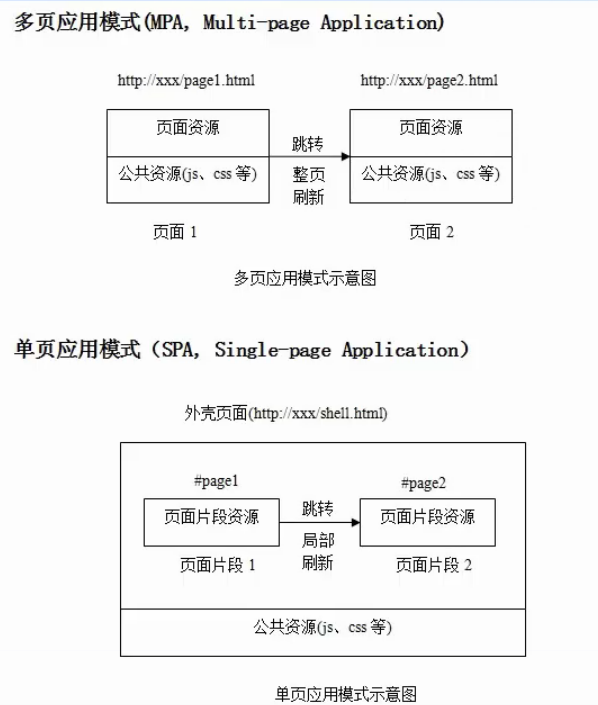
为实现单页Web应用功能及显示效果，需要在加载页面的时候将JavaScript、CSS统一加载，部分页面可以在需要的时候加载。所以必须对JavaScript及CSS代码进行合并压缩处理，如果使用第三方库，建议使用一些大公司的CDN，因此带宽的消耗是必然的。

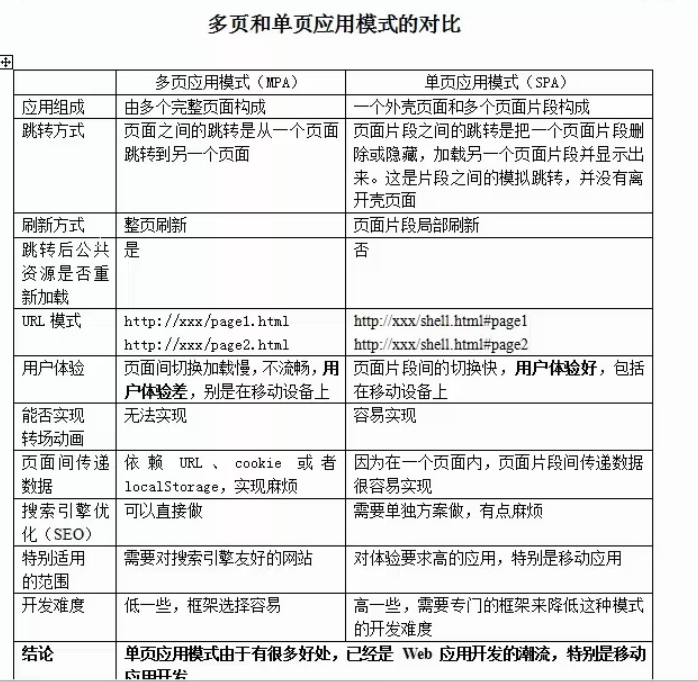
做成单页不是说要一次把所有页面打开，是通过路由动态加载,，路由组件可以按需加载。

四、框架AngularJS

AngularJS是一个 MV\* 框架，最适于开发客户端的单页面应用。它不是个功能库，而是用来开发动态网页的框架。它专注于扩展HTML的功能，提供动态数据绑定（data binding），而且它能跟其它框架（如jQuery）合作融洽。

如果你要开发的是单页应用，AngularJS就是你的上上之选。Gmail、Google Docs、Twitter和Facebook这样的应用，都很能发挥AngularJS的长处。但是像游戏开发之类对DOM进行大量操纵、又或者单纯需要极高运行速度的应用，就不是AngularJS的用武之地了。





### 三大前端框架：react、vue、angular

**Vue**: API设计上简单，语法简单，学习成本低；性能好，容易优化；更快的渲染速度和更小的体积。双向数据绑定。

缺点：不支持移动端,

**React**：适用于：大型应用和更好的可测试性；同时适用于web端和原生app;更大的生态圈；缺点：React是目标是UI组件，只负责VIEW层，通常可以和其它框架组合使用，目前并不适合单独做一个完整的框架。所以如果是大型项目想要一套完整的框架的话，也许还需要引入Redux和route相关的东西。而Angular在这方面提供的东西比React多多了。单向数据流。

**Angular**

优点：AngularJS是一套完整的框架，生产效率高

缺点：整体作为MVVM框架过重，性能问题，UI组件相对复杂，不利于重用。在Angular的作用域中任何操作的执行都会引发Dirty-checking，随着绑定数量的增加性能就会越低。（Angular 2.0推翻重做使得目前不宜采用此框架 ）；学习成本高

### React

this

this的本质就是：this跟作用域无关的，只跟执行上下文有关。

React组件生命周期函数中的this指向组件实例；

自定义组件方法的this会因调用者不同而不同；

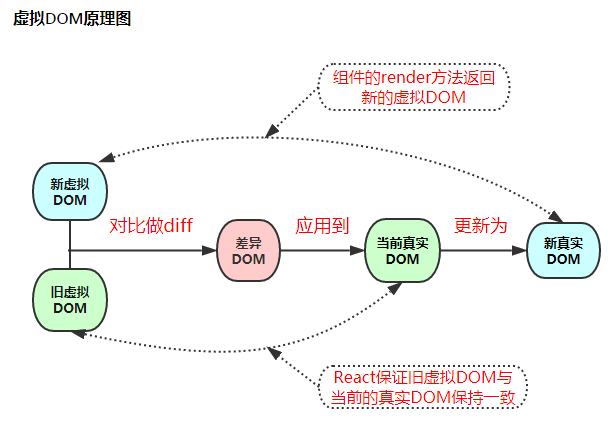
为了在组件的自定义方法中获取组件实例，需要手动绑定this到组将实例。

**虚拟DOM原理**

React将新的虚拟DOM（由组件的render方法返回，也就是真实DOM未来要显示的模样）与旧的虚拟DOM（结构、数据与当前的真实DOM一致）做比较，生成差异diff，并将diff应用到当前真实的DOM中，当前真实DOM更新为新的真实DOM（结构、数据与新的虚拟DOM一致）

注意 ：

在React中，开发者只需要在组件的render方法中返回新的虚拟DOM就可以了，具体真实DOM与虚拟DOM如何做对比生成diff，以及真实DOM如何通过最快的路径渲染出变动的部分，均由React来完成，不用开发者操心。

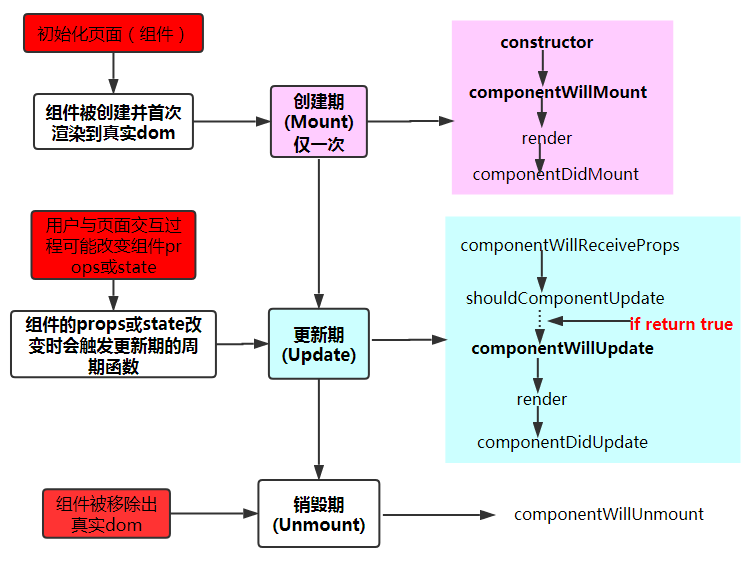


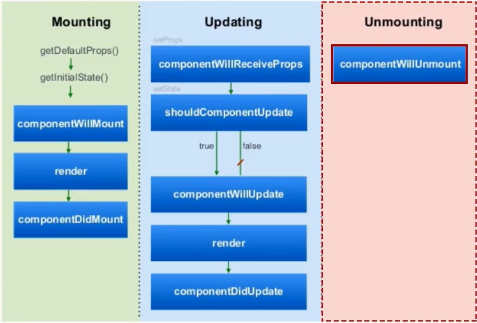
**使用虚拟DOM需注意**

1、组件（component）是虚拟 DOM ，并不是真实的 DOM 节点，而是存在于内存之中的一种数据结构。只有当它插入文档以后，才会变成真实的 DOM节点，大部分情况下，我们都是在 构建React的组件，也就是操作虚拟DOM ，components 的存在让计算 DOM diff 更高效。

2、为了同浏览器交互，我们有时候需要获取到 真实的DOM节点 。我们可以通过调用React的React.findDOMNode(component)获取到组件中真实的DOM。

3、React.findDOMNode()只在mounted组件中调用，mounted组件就是已经渲染在浏览器DOM结构中的组件。如果你在组件的render()方法中调用React.findDOMNode()就会抛出异常，是因为 render返回的是虚拟dom ，并没有被渲染到真实dom中。





使用create-react-app脚手架工具

**react-router**

**react-redux**

脚手架工具

Flux

## Spring

Spring核心的配置文件applicationContext.xml或者叫bean.xml

### applicationContext & BeanFactory

* BeanFactory 接口

spring原始接口.最底层的接口。针对原始接口的实现类功能较为单一

BeanFactory接口实现类的容器.特点是每次在获得对象时才会创建对象，为了节省内存

* ApplicationContext

每次容器启动时就会创建容器中配置的所有对象.并提供更多功能

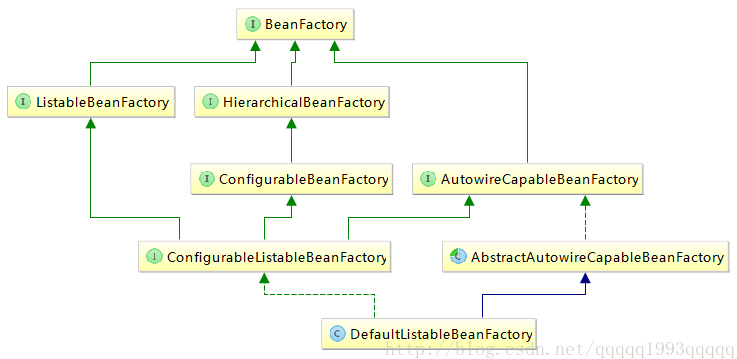
从类路径下加载配置文件:ClassPathXmlApplicationContext

从硬盘绝对路径下加载配置文件:FileSystemXmlApplicationContext(“d:/xxx/yyy/xxx”)

### Bean组件解析

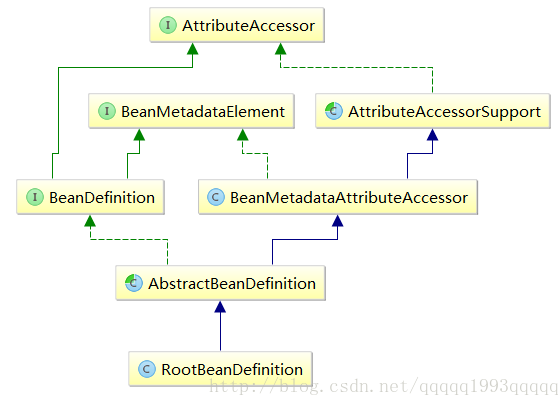
Spring Bean 的创建是典型的工厂模式， 它的顶级接口是BeanFactory。

Bean工厂的类层次关系图：



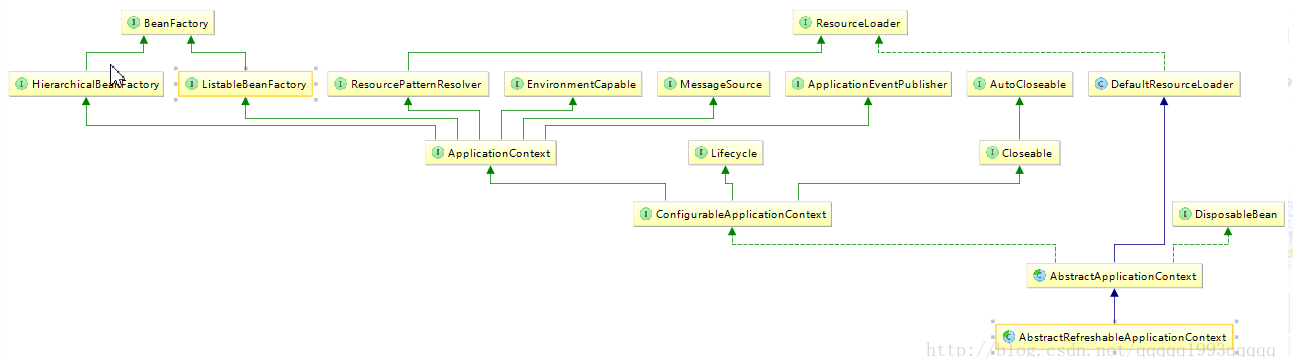
4个接口，共同定义了Bean 的集合、Bean 之间的关系和Bean 的行为。

Bean定义的类层次关系图：



Bean 的定义完整地描述了在Spring 的配置文件中你定义的< /bean>节点中所有的信息，包括各种子节点。

### Context组件分析



ApplicationContext 继承了BeanFactory，这也说明了Spring 容器中运行的主体对象是Bean 。另外ApplicationContext 继承了Re sourceLoader 接口，使得ApplicationContext 可以访问到任何外部资源。

ApplicationContext 必须要完成以下几件事情：

* 标识一个应用环境。
* 利用Bean Factory 创建Bean 对象。
* 保存对象关系表。
* 能够捕获各种事件。

### IOC(Inverse Of Control)

//1.创建容器对象，相对于src下的路径

ApplicationContext ac=new ClassPathXmlApplicationContext("applicationContext.xml");

//2.向容器“要”user对象

User u=(User) ac.getBean("user");

控制反转，就是反转了对象的创建方式，从我们自己创建反转给了程序（spring）

Spring提供了两类容器：

Spring BeanFactory 容器，它是最简单的容器，给 DI 提供了基本的支持，它用 org.springframework.beans.factory.BeanFactory 接口来定义。

Spring ApplicationContext 容器：

该容器添加了更多的企业特定的功能，例如从一个属性文件中解析文本信息的能力，发布应用程序事件给感兴趣的事件监听器的能力。该容器是由 org.springframework.context.ApplicationContext 接口定义。

ApplicationContext 容器包括 BeanFactory 容器的所有功能。

IOC初始化过程：

关于Spirng IoC容器的初始化过程在《Spirng技术内幕：深入解析Spring架构与设计原理》一书中有明确的指出，IoC容器的初始化过程可以分为三步：

Resource定位（Bean的定义文件定位）

将Resource定位好的资源载入到BeanDefinition

将BeanDefiniton注册到容器中

### DI（Dependency Injection）

依赖注入，向类的属性设置值

IoC与DI的关系：依赖注入不能单独存在，需要在IoC基础之上完成操作。

依赖关系（即 SpellChecker 类）通过类构造函数被注入到 TextEditor 类中。因此，控制流通过依赖注入（DI）已经“反转”，因为你已经有效地委托依赖关系到一些外部系统。

依赖注入的第二种方法是通过 TextEditor 类的 Setter 方法，我们将创建 SpellChecker 实例，该实例将被用于调用 setter 方法来初始化 TextEditor 的属性。

**value** 属性来配置基本数据类型

<property>标签的 **ref** 属性来配置对象引用

### 自动装配：

Spring 容器可以在不使用<constructor-arg>和<property> 元素的情况下自动装配相互协作的 bean 之间的关系，这有助于减少编写一个大的基于 Spring 的应用程序的 XML 配置的数量。

### AOP

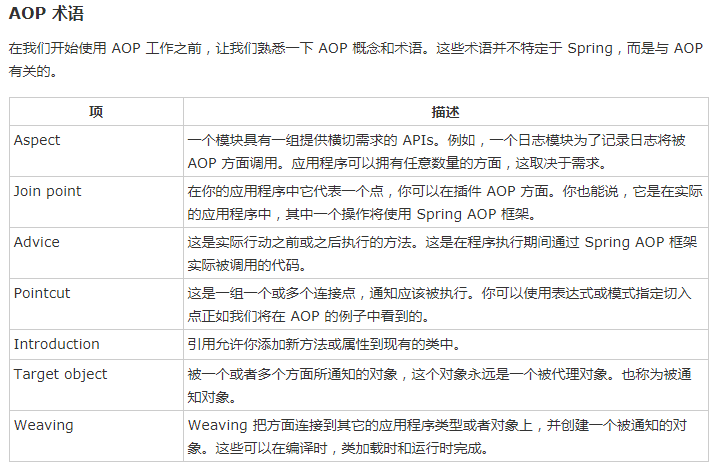
横向重复、纵向抽取

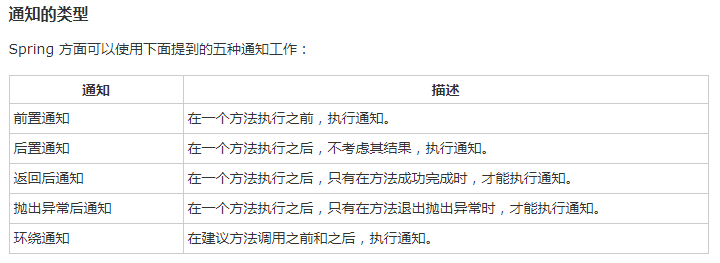
* Spring能够为容器中管理的对象生成动态代理对象
* 以前我们要使用动态代理，我们需要自己调用下面的方法生成对象Proxy.newProxyInstance(xx,xx,xx) 生成代理对象，Spring能够帮我们生成代理对象
* 动态代理（优先）

被代理对象必须要实现实现接口，才能产生代理对象。如果没有接口不能使用动态代理技术。

* cglib代理(没有接口使用)

第三方代理技术，cglib代理，可以对任何类实现代理，代理的原理是对目标对象进行继承代理。如果目标对象被final修饰，该类无法被cglib代理。





### 配置对象：

在Spring中总体来看可以通过三种方式来配置对象:

使用XML文件配置

使用注解来配置

使用JavaConfig来配置

### 事件（event）：

Spring的AppilcaitionContext能够发布事件和注册相对应的事件监听器，因此，它有一套完整的事件发布和监听机制。

**流程分析：**

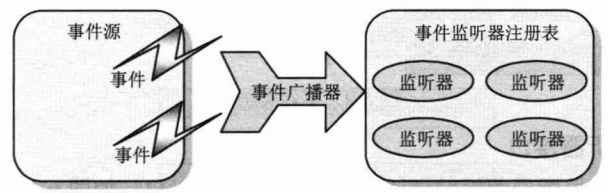
在一个完整的事件体系中，除了事件和监听器以外，还应该有3个概念；

1. 事件源：事件的产生者，任何一个event都必须有一个事件源；

2. 事件广播器：它是事件和事件监听器之间的桥梁，负责把事件通知给事件监听器；

3. 事件监听器注册表：就是spring框架为所有的监听器提供了一个存放的地方；

通过流程图，可以看出它们是如何各司其职的，如下：



其实通过流程图，我们很容易发现事件体系就是观察者模式的具体实现，它并没有任何的神秘之处。

**结构分析：**

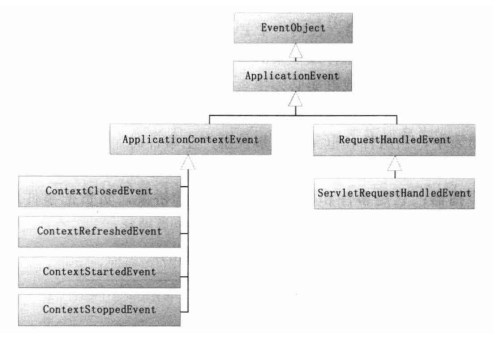
1. 事件类（ApplicaitonEvent）：目前spring框架本身仅仅提供了几个事件，很多的事件都是需要自定义的。

　ApplicationEvent唯一的构造函数是ApplicaitonEvent（Object source），通过source指定事件源。 它有两个子类；

（1）ApplicationContextEvent：容器事件，也就是说事件源是ApplicationContext，框架提供了四个子类，分别代表容器启动，刷新，停止和关闭事件。

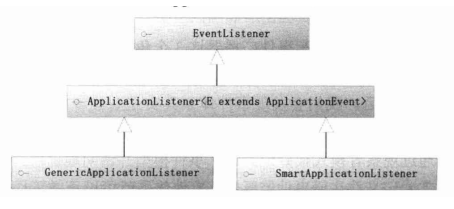
（2）RequestHandleEvent：这是一个与Web应用相关的事件，当一个请求被处理后，才会产生该事件。

一般来说，我们都是扩展ApplicationEvent来自定义事件。下面会有栗子。



2. 事件监听器接口（ApplicationListener）

所有的监听器都需要实现该接口，该接口只定义了一个方法：onApplicaitonEvent (E event)，该方法接收事件对象，在该方法中编写事件的响应处理逻辑。



**实例分析：**

通过一个实例来讲解事件发布和事件监听的过程。这个例子包括一个模拟的邮件发送器MailSender，它在向目的地发送邮件的时候 会产生一个MailSendEvent事件，我们同时向容器中注册了该事件的监听器MailSendListener；

上代码：MailSendEvent

/\*\*

\* 继承了ApplicationContextEvent，就是个容器事件

\*/

public class MailSendEvent extends ApplicationContextEvent {

private String to; //目的地

public MailSendEvent(ApplicationContext source, String to) {

super(source);

this.to = to;

}

public String getTo(){

return this.to;

}

}

很明显，这里的source是ApplicationContext容器。

我们再定义一个该事件的监听器MailSendListener，如下

@Component

public class MailSendListener implements ApplicationListener<MailSendEvent>{

@Override

public void onApplicationEvent(MailSendEvent mailSendEvent) {

MailSendEvent event = mailSendEvent;

System.out.println("MailSender向"+ event.getTo()+ "发送了邮件");

}

}

上述提及到所有的监听器都必须要实现ApplicationListener，这里是个泛型类，指定了该监听器只监听MailSendEvent事件。

OK，那我们现在来触发事件：MailSender，也就是事件源

@Component("mailSender")

public class MailSender {

@Autowired

private ApplicationContext applicationContext; //容器事件由容器触发

public void sendMail(String to){

System.out.println("MailSender开始发送邮件");

MailSendEvent event = new MailSendEvent(applicationContext,to);

applicationContext.publishEvent(event);

}

}

最后写个测试类，容器启动：

public class SpringEventTest {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("web/WEB-INF/applicationContext.xml");

MailSender sender = (MailSender)context.getBean("mailSender");

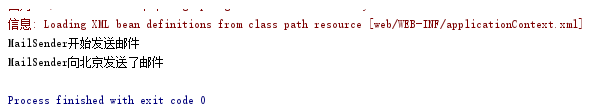
sender.sendMail("北京");

}

}

记得在applicationContext.xml里面加上注解扫描啊，我这里都是放在springEvent包里的，所以加上 <context:component-scan base-package="springEvent"/>

好了，启动容器，发送邮件：执行结果如下



### 事务管理：

一个数据库事务是一个被视为单一的工作单元的操作序列。这些操作应该要么完整地执行，要么完全不执行。事务管理是一个重要组成部分，RDBMS 面向企业应用程序，以确保数据完整性和一致性。事务的概念可以描述为具有以下四个关键属性说成是**ACID**：

* 原子性（Atomicity）：事务应该当作一个单独单元的操作，这意味着整个序列操作要么是成功，要么是失败的。
* 一致性（Consistency）：这表示数据库的引用完整性的一致性，表中唯一的主键等。
* 隔离性（Isolation）：可能同时处理很多有相同的数据集的事务，每个事务应该与其他事务隔离，以防止数据损坏。
* 持久性（Durability）：一个事务一旦完成全部操作后，这个事务的结果必须是永久性的，不能因系统故障而从数据库中删除。

一个真正的 RDBMS 数据库系统将为每个事务保证所有的四个属性。使用 SQL 发布到数据库中的事务的简单视图如下：

1、使用 begin transaction 命令开始事务。

2、使用 SQL 查询语句执行各种删除、更新或插入操作。

3、如果所有的操作都成功，则执行提交操作，否则回滚所有操作。

Spring 框架在不同的底层事务管理APIs的顶部提供了一个抽象层。Spring 的事务支持旨在通过添加事务能力到POJOs来提供给 EJB 事务一个选择方案。Spring支持编程式和声明式事务管理。EJBs需要一个应用程序服务器，但Spring事务管理可以在不需要应用程序服务器的情况下实现。

**局部事物 vs. 全局事务**

局部事务是特定于一个单一的事务资源，如一个 JDBC 连接，而全局事务可以跨多个事务资源事务，如在一个分布式系统中的事务。

局部事务管理在一个集中的计算环境中是有用的，该计算环境中应用程序组件和资源位于一个单位点，而事务管理只涉及到一个运行在一个单一机器中的本地数据管理器。局部事务更容易实现。

全局事务管理需要在分布式计算环境中，所有的资源都分布在多个系统中。在这种情况下事务管理需要同时在局部和全局范围内进行。分布式或全局事务跨多个系统执行，它的执行需要全局事务管理系统和所有相关系统的局部数据管理人员之间的协调。

**编程式 vs. 声明式**

Spring 支持两种类型的事务管理:

* 编程式事务管理 ：这意味着你在编程的帮助下有管理事务。这给了你极大的灵活性，但却很难维护。
* 声明式事务管理 ：这意味着你从业务代码中分离事务管理。你仅仅使用注释或 XML 配置来管理事务。

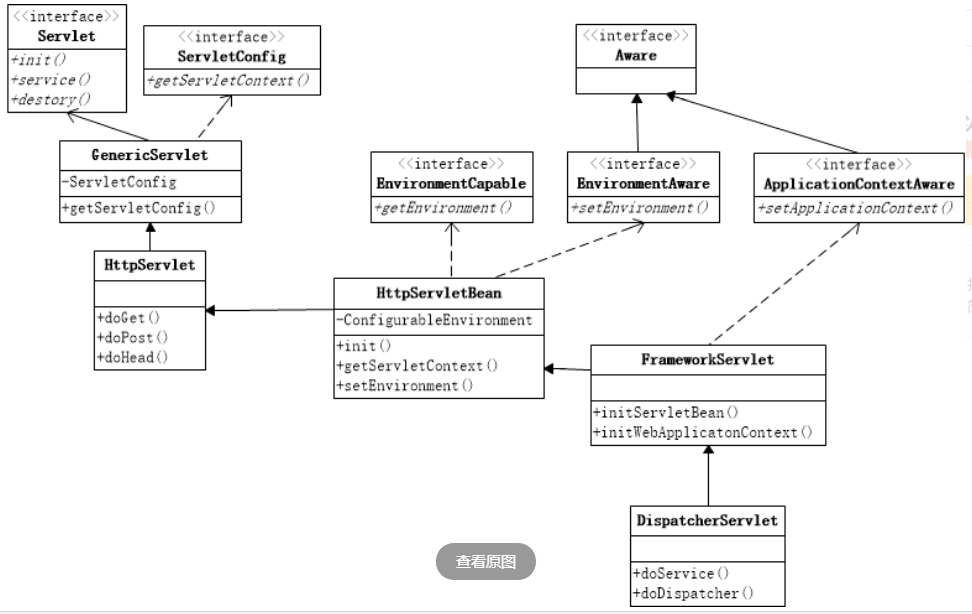
声明式事务管理比编程式事务管理更可取，尽管它不如编程式事务管理灵活，但它允许你通过代码控制事务。但作为一种横切关注点，声明式事务管理可以使用 AOP 方法进行模块化。Spring 支持使用 Spring AOP 框架的声明式事务管理。

编程式事务管理：

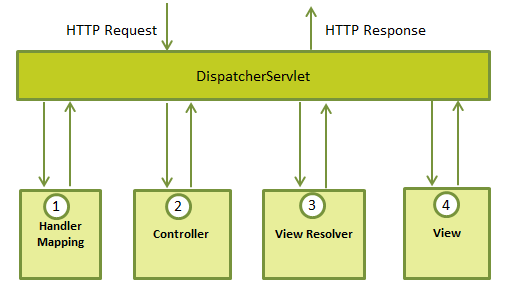
让我们直接使用 PlatformTransactionManager 来实现编程式方法从而实现事务。要开始一个新事务，你需要有一个带有适当的 transaction 属性的 TransactionDefinition 的实例。这个例子中，我们使用默认的 transaction 属性简单的创建了 DefaultTransactionDefinition 的一个实例。

当 TransactionDefinition 创建后，你可以通过调用 getTransaction() 方法来开始你的事务，该方法会返回 TransactionStatus 的一个实例。 TransactionStatus 对象帮助追踪当前的事务状态，并且最终，如果一切运行顺利，你可以使用 PlatformTransactionManager 的 commit() 方法来提交这个事务，否则的话，你可以使用 rollback() 方法来回滚整个操作。

## Spring MVC



Spring Web 模型-视图-控制（MVC）框架是围绕 DispatcherServlet 设计的，DispatcherServlet 用来处理所有的 HTTP 请求和响应。Spring Web MVC DispatcherServlet 的请求处理的工作流程如下图所示：



下面是对应于 DispatcherServlet 传入 HTTP 请求的事件序列：

* 收到一个 HTTP 请求后，DispatcherServlet 根据 HandlerMapping 来选择并且调用适当的控制器。
* 控制器接受请求，并基于使用的 GET 或 POST 方法来调用适当的 service 方法。Service 方法将设置基于定义的业务逻辑的模型数据，并返回视图名称到 DispatcherServlet 中。
* DispatcherServlet 会从 ViewResolver 获取帮助，为请求检取定义视图。
* 一旦确定视图，DispatcherServlet 将把模型数据传递给视图，最后呈现在浏览器中。

上面所提到的所有组件，即 HandlerMapping、Controller 和 ViewResolver 是 WebApplicationContext 的一部分，而 WebApplicationContext 是带有一些对 web 应用程序必要的额外特性的 ApplicationContext 的扩展。

**配置：**

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.1.0</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>javax.servlet.jsp-api</artifactId>

<version>2.3.1</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

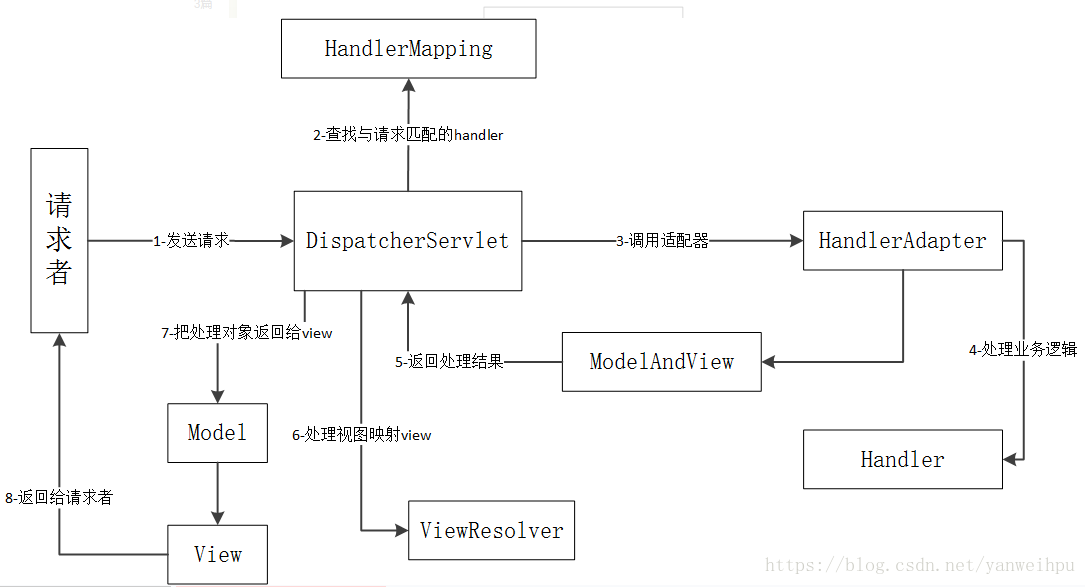
<version>1.2</version>

</dependency>

**加载：**

web.xml 文件将被保留在你的应用程序的 WebContent/WEB-INF 目录下。在初始化 SpringMVCDemo DispatcherServlet 时，该框架将尝试加载位于该应用程序的 WebContent/WEB-INF 目录中文件名为 [servlet-name]-servlet.xml 的应用程序内容。在这种情况下，我们的文件将是 servlet-servlet.xml。

**SpringMVC实现原理**



流程说明：

（1）客户端（浏览器）发送请求，直接请求到DispatcherServlet。

（2）DispatcherServlet根据请求信息调用HandlerMapping，解析请求对应的Handler。

（3）解析到对应的Handler后，开始由HandlerAdapter适配器处理。

（4）HandlerAdapter会根据Handler来调用真正的处理器开处理请求，并处理相应的业务逻辑。

（5）处理器处理完业务后，会返回一个ModelAndView对象，Model是返回的数据对象，View是个逻辑上的View。

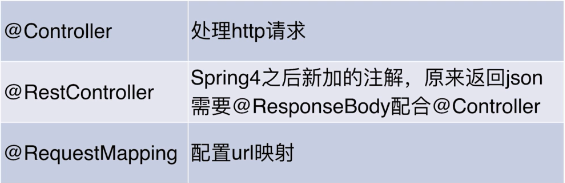
（6）ViewResolver会根据逻辑View查找实际的View。

（7）DispaterServlet把返回的Model传给View。

（8）通过View返回给请求者（浏览器）

## Spring Boot

Spring Boot依赖使用的groupId为 org.springframework.boot 。通常，你的Maven POM文件会继承 spring-boot-starter-parent 工程，并声明一个或多个“Starter POMs”依赖。此外，Spring Boot提供了一个可选的Maven插件，用于创建可执行jars。



@Controller 处理http请求，如果不加此注解，浏览器无法访问到。

@RestController = @Controller + @ResponseBody

Example类上使用的第一个注解是@RestController，这被称为构造型（stereotype）注解。它为阅读代码的人提供暗示（这是一个支持REST的控制器），对于Spring，该类扮演了一个特殊角色。在本示例中，我们的类是一个web @Controller，所以当web请求进来时，Spring会考虑是否使用它来处理。

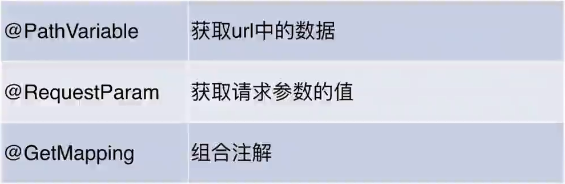
@RequestMapping注解提供路由信息，它告诉Spring任何来自"/"路径的HTTP请求都应该被映射到home方法。@RestController注解告诉Spring以字符串的形式渲染结果，并直接返回给调用者。+

注：@RestController和@RequestMapping是Spring MVC中的注解（它们不是Spring Boot的特定部分）

第二个类级别的注解是@EnableAutoConfiguration，这个注解告诉Spring Boot根据添加的jar依赖猜测你想如何配置Spring。由于spring-boot-starter-web添加了Tomcat和Spring MVC，所以auto-configuration将假定你正在开发一个web应用，并对Spring进行相应地设置。

Starters和Auto-Configuration：Auto-configuration设计成可以跟"Starters"一起很好的使用，但这两个概念没有直接的联系。你可以自由地挑选starters以外的jar依赖，Spring Boot仍会尽最大努力去自动配置你的应用。

@SpringBootApplication注解等价于以默认属性使用@Configuration，@EnableAutoConfiguration和@ComponentScan：



\*RequestMethod.POST 会报405错误，用postman工具来测试。

Spring Data Jpa:

spring.jpa.hibernate.ddl-auto=create

create：每次删掉之前的表，创建新的表

update：第一次运行创建表，如果里面有数据，则不删除表

//todo Spring boot security模块

//todo 如何做分页读取数据

## Spring Cloud

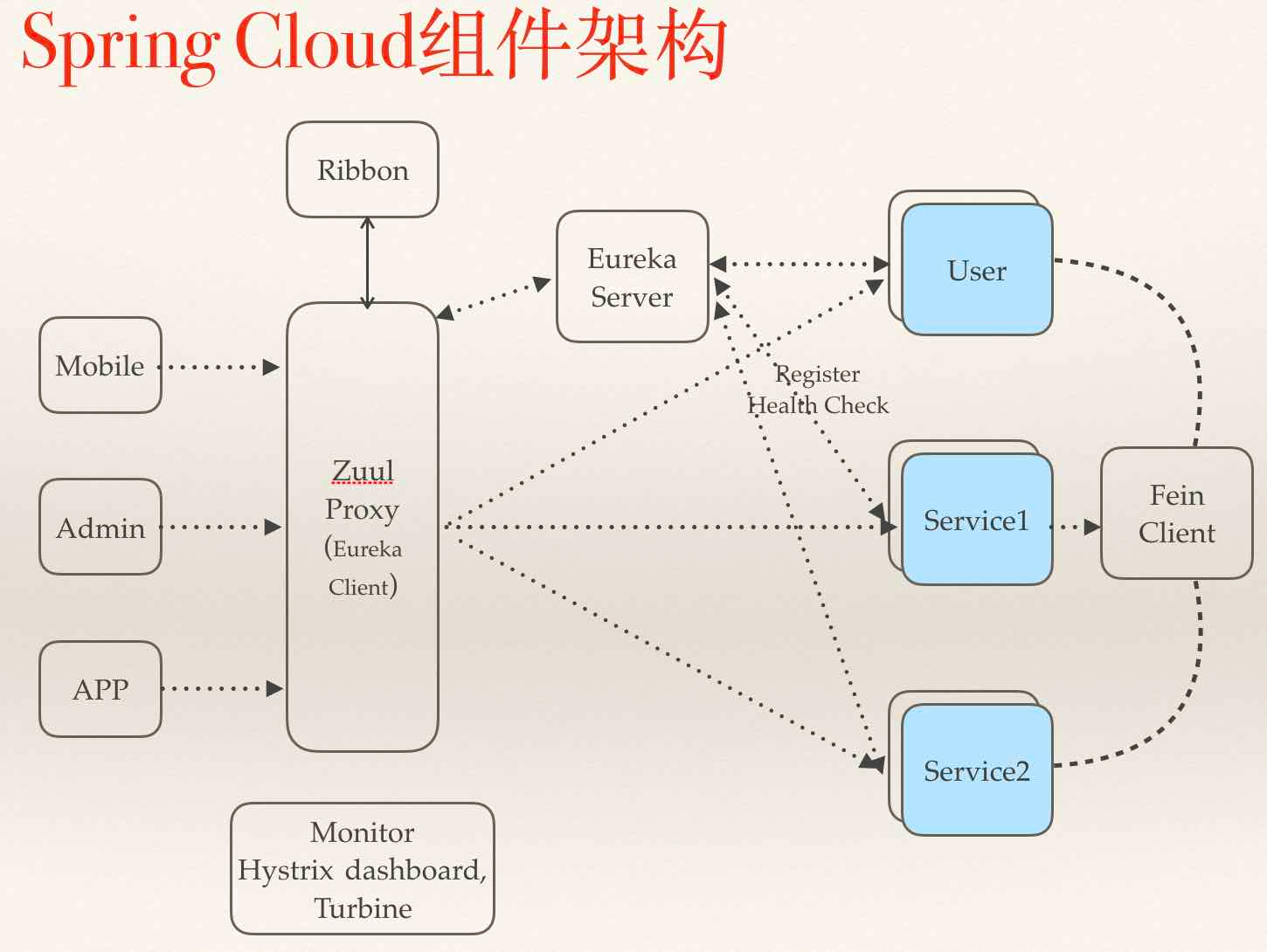
Spring Cloud是一系列框架的有序集合。它利用Spring Boot的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用Spring Boot的开发风格做到一键启动和部署。Spring并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过Spring Boot风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

微服务是可以独立部署、水平扩展、独立访问（或者有独立的数据库）的服务单元，springcloud就是这些微服务的大管家，采用了微服务这种架构之后，项目的数量会非常多，springcloud做为大管家需要管理好这些微服务，自然需要很多小弟来帮忙。

**和Spring Boot 是什么关系**

Spring Boot 是 Spring 的一套快速配置脚手架，可以基于Spring Boot 快速开发单个微服务，Spring Cloud是一个基于Spring Boot实现的云应用开发工具；Spring Boot专注于快速、方便集成的单个个体，Spring Cloud是关注全局的服务治理框架；Spring Boot使用了默认大于配置的理念，很多集成方案已经帮你选择好了，能不配置就不配置，Spring Cloud很大的一部分是基于Spring Boot来实现,可以不基于Spring Boot吗？不可以。

Spring Boot可以离开Spring Cloud独立使用开发项目，但是Spring Cloud离不开Spring Boot，属于依赖的关系。



1、外部或者内部的非Spring Cloud项目都统一通过API网关（Zuul）来访问内部服务.

2、网关接收到请求后，从注册中心（Eureka）获取可用服务

3、由Ribbon进行均衡负载后，分发到后端的具体实例

4、微服务之间通过Feign进行通信处理业务

5、Hystrix负责处理服务超时熔断

6、Turbine监控服务间的调用和熔断相关指标

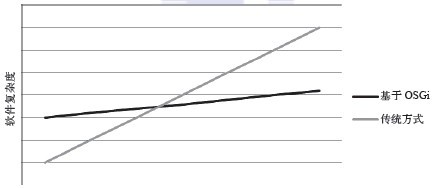
图中没有画出配置中心，配置中心管理各微服务不同环境下的配置文件。

以上就是一个完整的Spring Cloud生态图。

## OSGI架构简述

Open Service Gateway Initiative，开放服务网关。OSGi联盟给出的最新OSGi定义是The Dynamic Module System for Java，即面向Java的**动态模块化**系统。

1. 引入OSGi对软件复杂度的影响



1. 对于开发效率的提升

具有“自约束能力”的系统架构。

1. 模块化

在OSGi环境下，“大杂烩”形式的模块是很难生存的，如果某个模块有非常多的依赖项，那么没有人愿意为了使用其中少量功能去承担这些间接依赖的代价。因此设计者必须把模块设计得粒度合理，精心挑选对外发布的接口和引入的依赖，把每个模块视为一个商业产品来对待，这样才能积累出可重用的模块，也利于提高程序稳定性。

1. 强鲁棒性（抗干扰能力）

在设计时自然会考虑到模块自治和动态化，当某部分不可用时如何处理是每时每刻都会考虑的问题

1. 动态修复缺陷

可以做到不停机地增加或禁止某项功能、更新某个模块，甚至建立一个统一更新的模块仓库，让系统在不中断运行的情况下做到自动更新升级。

1. 性能

从执行和内存两方面来讨论。在执行上不可避免地会有一些损耗；从内存用量来看，OSGi允许不同版本的Package同时存在，这是个优点，但是客观上会占用更多内存。

1. 在企业级开发中的角色

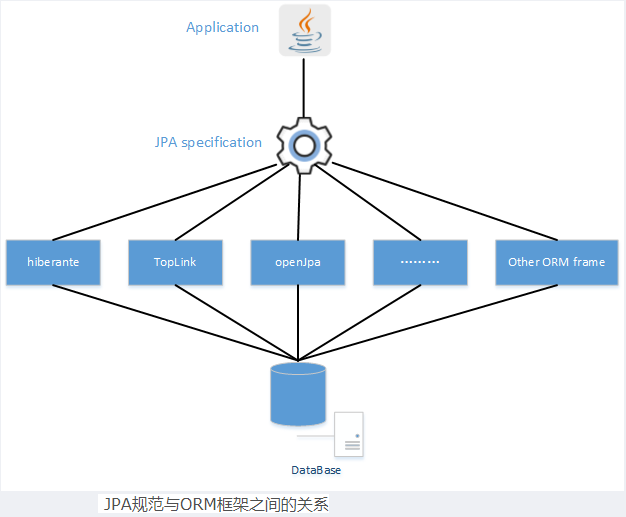
没有把Java EE或SSH中积累的东西推倒重来，OSGi更像是在扮演一个“组织者”的角色，把各种企业级技术变为它的模块和服务，使以前的企业级开发技术在OSGi中依然能够发挥作用。

## JPA

JPA Java Persistence API，是Java EE 5的标准ORM接口，也是ejb3规范的一部分。是一种ORM规范，注意不是ORM框架——因为JPA并未提供ORM实现，它只是制订了一些规范，提供了一些编程的API接口，但具体实现则由服务厂商来提供实现，JBoss应用服务器底层就以Hibernate作为JPA的实现。

Hibernate，当今很流行的ORM框架，是JPA的一个实现，但是其功能是JPA的超集。

JPA和Hibernate之间的关系，可以简单的理解为JPA是标准接口，Hibernate是实现。



Spring Data JPA与JPA规范的关系

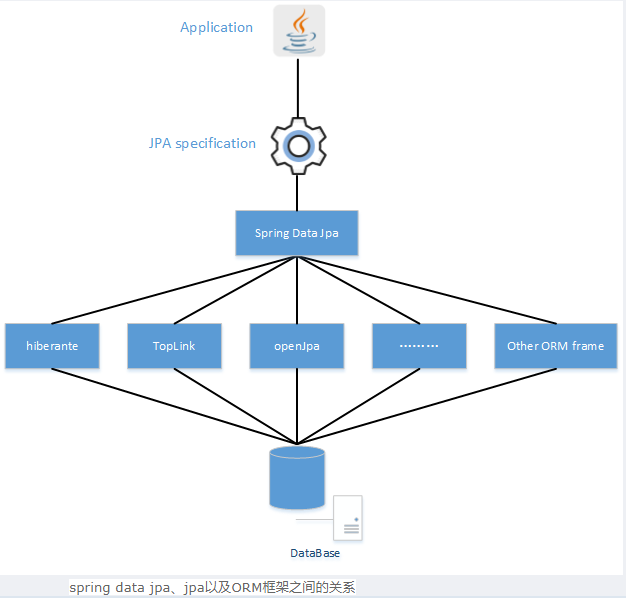
实现应用程序的数据访问层已经很麻烦了好一阵子。太多的样板代码必须被写入。Domain classes，并没有被设计成面向一个真正的对象或领域驱动的方式。

使用spring data jpa能够使丰富的Domain classes的持久性开发变得轻松很多，即使样板代码来实现存储库量特别还是相当高的。所以Spring data jpa的目标是简化关于各种持久存储数据访问层而努力。

备注：Domain classes 指的是POJO类，例如数据库中有一张表：Student，那么我们会在程序中定义与之对应的Student.java，而这个Student.java就是属于Domain classes。长话短说，Spring Data JPA 是在JPA规范的基础下提供了Repository层的实现，但是使用那一款ORM需要你自己去决定。

springdata jpa是在hibernate的基础上更上层的封装实现。

我的理解是：虽然ORM框架都实现了JPA规范，但是在不同ORM框架之间切换是需要编写的代码有一些差异，而通过使用Spring Data Jpa能够方便大家在不同的ORM框架中间进行切换而不要更改代码。并且Spring Data Jpa对Repository层封装的很好，可以省去不少的麻烦。



## Hibernate

**持久化：**将我们想要保存的数据保存到硬盘上，也就是我们电脑的磁盘上，为什么叫持久化呢，就是数据能够保存的很久，所以叫持久化，现在对持久化的实现过程大多通过各种关系型数据库完成，所以我们常说的，将数据保存到数据库中，其实是数据库帮我们帮数据保存到硬盘中了。

**持久层：**既然知道了什么是持久化，那么持久化层也就应该有点思路了，这里吧数据库看成是内存的一部分，我们就当做将数据保存到数据库中，就保存到了硬盘中一样，所以在操作数据库的或者跟数据库打交道的那一层就是就持久层，比如我们之前知道了三层架构，不就有专门跟数据库打交道的一层叫做持久化层吗

**ORM：**Object Relational Mapping，对象关系映射，这个是一个思想，模型，或者说是规范。关系数据库中的记录映射成为程序语言中的对象实例，然后通过操作对象，来达到操作数据库的这样一种思想。如果没有ORM思想，我们之前就是直接操作数据库中的记录字段，来达到存储数据的目的。

**持久化类：**通过上面解释的，持久化类就是可以将类保存到数据库中，并还可以从数据库中拿到该类，这就叫持久化类，也就是下面说到的POJO类

**持久化对象：**持久化类的实例对象，能保存到数据库中，也能从数据库中取出来。

**JPA的概念：**Java Persistence API java持久化API，也就是java持久化的规范，ORM就是这JPA中所定义的，它还规定了其它很多规范，JPA维护一个Persistence Context(持久化上下文)，这就是这个持久化上下文来事。那些ORM框架都要依据JPA规范来设计，那么各个ORM框架也就度有这么个持久化上下文。持久化上下文大体内容：1、ORM元数据，JPA支持annotion(注解)或xml两种形式描述对象/关系映射 2、实体操作API，实现对实体对象的CRUD操作 3、查询语言，约定了面向对象的查询语言JPQL(javaPersistence Query Language)

知道了一些大概的名词解释，现在来看看hibernate在一个工程中到底处于一个什么样的地位

**简要体系架构结构**

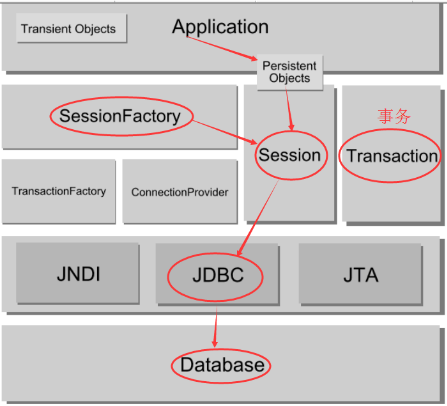


简要的hibernate体系架构就如上图所示，就如我们说的那样，hibernate在Application和Database之间，所以我们使用hibernate来实现对Database的操作。

我们配置xxx.hbm.xml：目的就是为了hibernate与我们的application想关联

我们配置hibernate.cfg.xml:与数据库相关的服务，例如：用户名和密码等。还有自身hibernate的服务。

**Hibernate全面解决方案体系架构**



**hibernate有什么作用**

前面知道了什么事hibernate，其实也就是知道hibernate的一些特性，很虚的东西，但是我们用它到底来做什么呢？

1、通过hibernate，完成POJO类到数据库表的映射关系

2、通过hibernate，只需要操作对象，hibernate就帮我们生成数据库语句去操作数据库，我们不必管下面的语句是怎么样的。

3、大概就是上面说的这两个，就是让用户来对对象的增加、删除、修改操作，来达到对数据库表中数据的这种增删改的操作。

如何实现POJO类到数据库表的映射关系，xxx.hbm.xml配置文件发威了因为有这个配置文件，pojo类具有操作的能力了。

有了xxx.hbm.xml这个映射文件还不够，因为hibernate需要连接数据库，那么这些操作放哪里呢，就提取出一个公用的配置文件出来，hibernate.cfg.xml就是这种公共的配置文件，加载数据库连接的信息，和将各种映射文件加载进来，其实就是抽取出来的，因为有很多映射文件，每个映射文件都需要连接数据库等操作，那么久将共同的操作提取出来形成了hibernate.cfg.xml。

## NoSQL

Mango DB

## redis简述

1 redis是什么？

通常而言目前的数据库分类有几种，包括 SQL/NSQL,，关系数据库，键值数据库等等 等，分类的标准也不以，Redis本质上也是一种键值数据库的，但它在保持键值数据库简单快捷特点的同时，又吸收了部分关系数据库的优点。从而使它的位置处于关系数据库和键值数 据库之间。Redis不仅能保存Strings类型的数据，还能保存Lists类型（有序）和Sets类型（无序）的数据，而且还能完成排序（SORT） 等高级功能，在实现INCR，SETNX等功能的时候，保证了其操作的原子性，除此以外，还支持主从复制等功能。

2 Redis用来做什么？

通常局限点来说，Redis也以消息队列的形式存在，作为内嵌的List存在，满足实时的高并发需求。而通常在一个电商类型的数据处理过程之中，有关商品，热销，推荐排序的队列，通常存放在Redis之中，期间也包扩Storm对于Redis列表的读取和更新。

3 Redis的优点

性能极高 – Redis能支持超过 100K+ 每秒的读写频率。

丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。

原子 – Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行。

丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

4 Redis的缺点

是数据库容量受到物理内存的限制,不能用作海量数据的高性能读写,因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。

 总结： Redis受限于特定的场景，专注于特定的领域之下，速度相当之快，目前还未找到能替代使用产品。

## 大数据

Spark、hadoop

## Docker

Docker:an open source project to pack, ship and run any application as a lightweight container.

简述

Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。容器是完全使用沙箱机制，相互之间不会有任何接口（类似 iPhone 的 app）。几乎没有性能开销，可以很容易地在机器和数据中心中运行。最重要的是，他们不依赖于任何语言、框架包括系统。

Docker是什么？

Docker是基于Go语言实现的云开源项目，诞生于2013年初，最初是dotCloud公司发起。

Docker通过对应用组件的封装、分发、部署、运行等生命周期的管理，达到应用组件级别的“一次封装，到处运行”。这里应用组件，即可以是Web应用，也可以是一套数据库服务，甚至是一个操作系统或编译器。

Docker由镜像(Image)、容器(Container)、仓库(Repository) 三大核心组成。

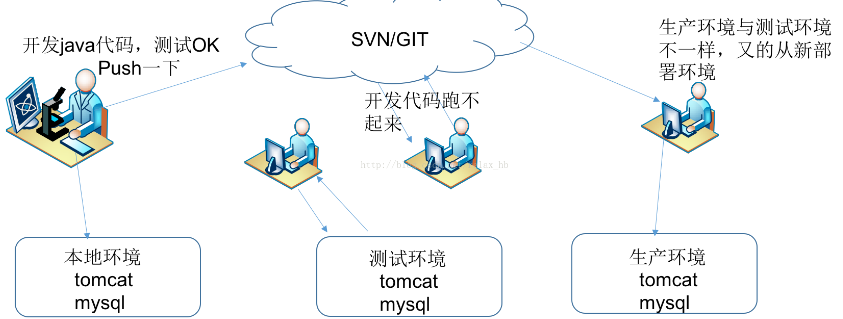
为什么要用Docker？

更快速的交付和部署(使用docker，开发人员可以用镜像来快速构建一套标准的开发环境;开发完成之后，测试和运维人员可以直接使用相同环境来部署代码。)

更轻松的迁移和扩展(docker容器几乎可以在任意平台上运行，包括物理机、虚拟机、公有云、私有云、个人电脑、服务器等。可以在不同的平台轻松地迁移应用)

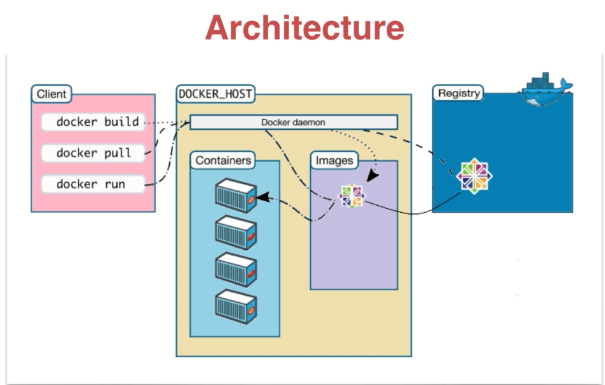
更简单的更新管理(使用Dockerfile，只需要修改小小的配置，就可以替代以往大量的更新工作)

场景示例-传统开发流程



场景示例-docker环境开发流程





什么是 Docker？

装应用的容器。

Docker思想：

* 集装箱
* 标准化：1、运输方式标准化。不同机器/环境之间的运输标准化；2、存储方式标准化。3、API接口标准化。
* 隔离：底层采用linux的一种内核限制机制，LXC，一种轻量级容器虚拟化技术，最大效率的隔离了进程和资源。

Docker解决了什么问题？

* 运行环境不一致所带来的问题。
* 隔离性解决了多用户共用服务器的问题。例如，限定了某用户的最大使用内存等。
* 标准化让服务器快速扩展、弹性伸缩变得简单。

Docker核心技术：

Docker运行程序的过程：去仓库把镜像拉到本地，然后用一条命令把镜像运行起来，变成容器。

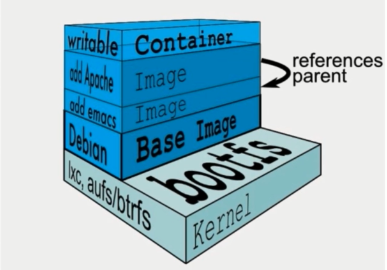
Build，构建镜像

Ship，运输镜像，从仓库和主机上运输

Run，运行容器

* 镜像Image：集装箱。一系列的文件，应用程序文件、运行环境的文件，保存在本地上。构建进项的目的：在其他环境上运行程序。联合文件系统，一种分层文件系统，可以将不同的目录挂到同一个虚拟文件系统下。

最直观的的镜像存储方式：



* 容器Container：运行程序的地方。本质就是一个进程。如果程序运行期间需要写一个镜像里的文件，就把文件拷贝到容器中再进行修改。因为容器层可以修改，而镜像层不能修改，因此可以保证同一个镜像可以生成多个容器独立运行，之间没有任何干扰。
* 仓库：超级码头

把镜像传到其他服务器，这个传输过程即利用了仓库，先把镜像传到仓库，再由目的地从仓库拉取镜像。Docker自己提供了中央服务器来存储仓库。

Docker自己提供的仓库：hub.docker.com

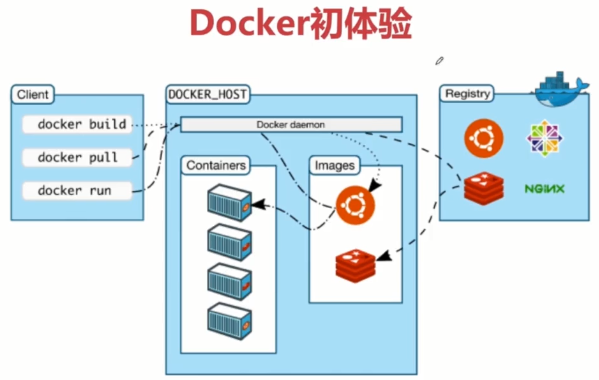
网易蜂巢：c.163.com 带有鲸鱼标志的镜像是从docker官网的镜像中信同步过来的，与docker的镜像一模一样。

拉取一个镜像：

docker pull hello-world

说明：1、镜像名可以在docker的官网上可以查到；2、默认从hub.docker.com上根据镜像名下载镜像。

docker run hello-world



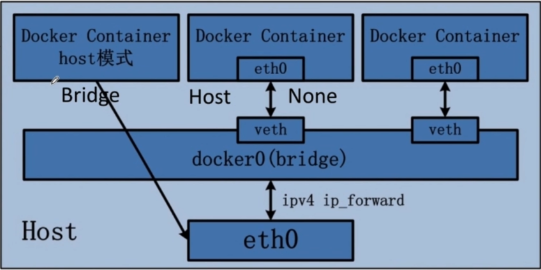
执行docker run的时候，如果镜像在本机不存在，会去仓库pull镜像。

docker run –d 在容器后台运行

docker exec 进入到容器内部。容器就相当于一台linux

Docker网络：

* 网络类型：bridge，host，none
* 端口映射：docker可以指定容器内的某一个端口和容器所在主机上的某个端口做一个映射



\*eth0是主机的网卡

## Kubernetes

## netty框架简述

1. netty是**基于Java NIO**（Nonblocking I/O，非阻塞IO）**的网络应用框架**，client-server框架
2. Netty是一个高性能、异步事件驱动的NIO框架，它提供了对TCP、UDP和文件传输的支持，作为一个异步NIO框架，**Netty的所有IO操作都是异步非阻塞**的，通过**Future-Listener机制，用户可以方便的主动获取或者通过通知机制获得IO操作结果**。
3. 作为当前最流行的NIO框架，Netty在互联网领域、大数据分布式计算领域、游戏行业、通信行业等获得了广泛的应用，一些业界著名的开源组件也基于Netty的NIO框架构建。
4. Netty三大特性：

并发高：NIO

\*NIO，同步非阻塞IO，阻塞业务处理但不阻塞数据接收，适用于高并发且处理简单的场景，比如聊天软件。

\*异步IO，他的数据请求和数据处理都是异步的，数据请求一次返回一次，适用于长连接的业务场景。

传输快：zero copy

封装好：Channel、ByteBuf、Codec

## nginx简述

## Python

html.parser

django

## XML报文和json报文

## 分布式（微服务）、高并发

https://www.cnblogs.com/uttu/p/6553307.html

幂等性

分层→异步回调（非阻塞I/O）

缓存技术

存储技术（NoSQL）

分布式系统在可管理性上造成的问题：

1. 硬件故障率
2. 资源利用率优化。如何扩容/缩容服务器；
3. 软件服务内容更新。批量安装部署；不同版本升级；
4. 数据统计和决策。

解决分布式系统可管理性的基本手段：

1. 目录服务（ZooKeeper）
2. 消息队列服务（ActiveMQ、ZeroMQ、Jgroups）
3. 事务系统。
4. 自动部署工具（Docker）
5. 日志服务（log4j）。分布式日志存储（类似Map Reduce系统），Hadoop或Kafka系统。

分布式系统在开发效率上造成的问题和解决思路：

1. 微服务框架：EJB、WebService。
2. 异步编程工具：协程、Futrue、Lamda。回调→“协程”

分布式系统解决方案：

1、水平拆分（1个请求平均分）；2、垂直扩展（多个请求分别分）；

一般采用水平+垂直协作。

例如：数据库层，可以按用户id进行水平拆表；按业务做垂直拆表；

常见互联网分布式架构如上，分为：

（1）客户端层：典型调用方是浏览器browser或者手机应用APP

（2）反向代理层：系统入口，反向代理

（3）站点应用层：实现核心应用逻辑，返回html或者json

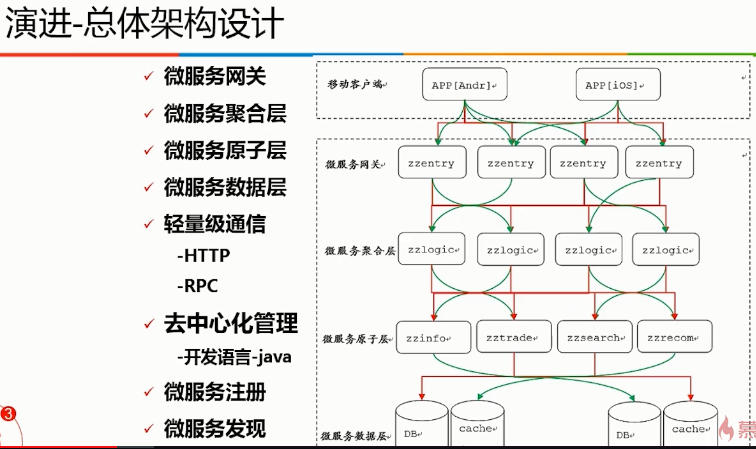
（4）服务层：如果实现了服务化，就有这一层

（5）数据-缓存层：缓存加速访问存储

（6）数据-数据库层：数据库固化数据存储

微服务架构在转转（二手交易网站）的演进



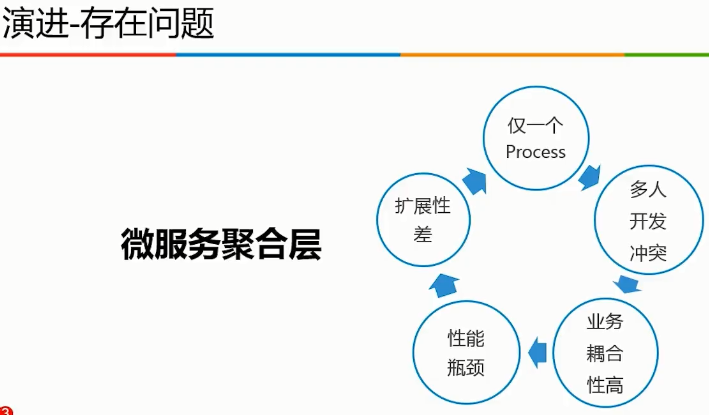


微服务网关层：对外提供HTTP和HTTPS接口。维护海量的练级（HTTP，tcp长连接）；对用户请求用户身份进行校验（合法性，登录情况，session管理）；对请求做转化，转到到聚合层。

微服务聚合层：采用RPC接口。即业务逻辑层。根据用户请求，把请求拆分成多个微服务原子，发送给微服务原子层。

微服务原子层：采用RPC接口。提供微服务（增删改查）接口。汇聚请求，最终提供给微服务网关层。

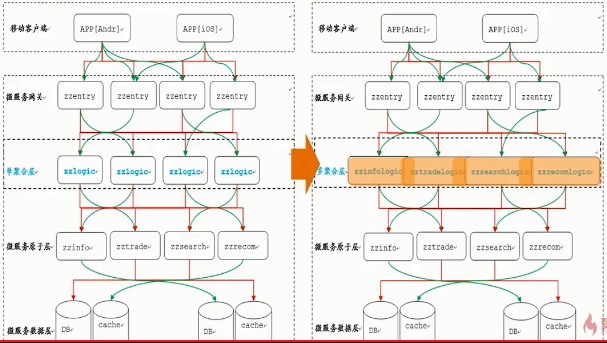
微服务数据层：对每一个微服务单独保存数据库。



因此，将zzlogic进行拆分：



架构演进：

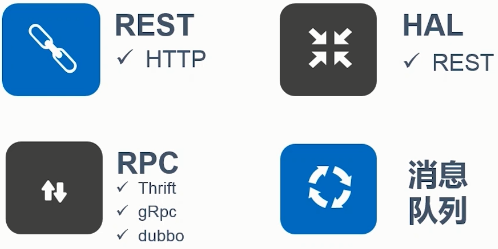


按照业务模块对聚合层进行拆分，拆分成多聚合层。

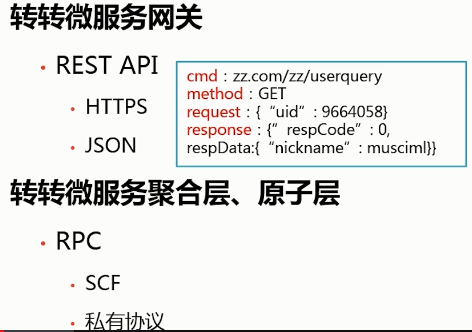
优点：



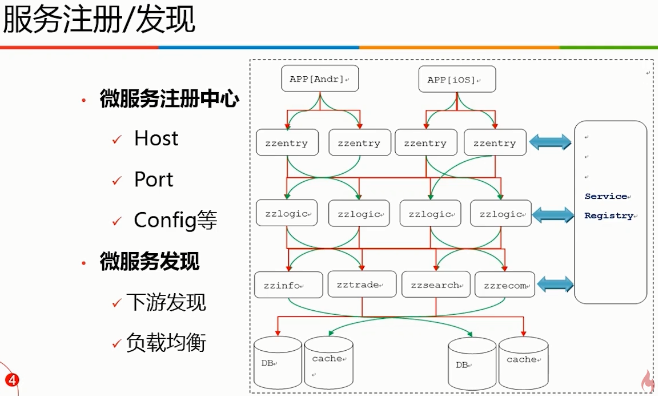
轻量级通信协议：

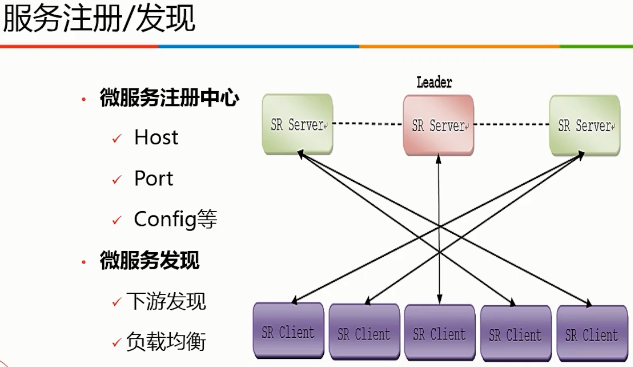


\*HAL，国内用的比较少。



\*通信协议用HTTPS，数据传输协议用JSON。





## 中间件

Weblogic，tomcat

Zookeeper，kafka，MQ（消息中间件），

什么是中间件

中间件是在操作系统功能范围外为应用提供服务的多用途软件。任何位于内核和用户应用之间的软件都可以是中间件。

分析师兼系统理论家 Nick Gall 表示，“中间件是作用于软件的软件。” 中间件不提供传统应用的功能，而是将软件与其他软件衔接。中间件是您 IT 基础架构的输水管，因为它使得数据能从一个应用流动到另一个中。

中间件分为两大类：

一类是底层中间件，用于支撑单个应用系统或解决单一类问题，包括

交易中间件（TPM）

应用服务器（WAS）

消息中间件（MOM）

数据访问中间件（UDA）

等

另一类是高层中间件，更多用于系统整合，包括

企业应用集成中间件（EAI Suites）

工作流中间件（Workflow）

门户中间件（Portal）

等

它们通常会与多个应用系统打交道，在系统中的层次较高，并大多基于底层中间件运行。

小结：

你只需要【消息中间件】只是【中间件】的其中一种就行了。

什么是消息中间件？

关注于数据的发送和接收，利用高效可靠的异步消息传递机制集成分布式系统。

消息中间件与分布式对象调用的比较

分布式对象调用，如CORBA，RMI和DCOM，提供了一种通讯机制，透明地在异构的分布式计算环境中传递对象请求，而这些对象可以位于本地或远程机器。它通过在对象与对象之间提供一种统一的接口，使对象之间的调用和数据共享不再关心对象的位置、实现语言及所驻留的操作系统。这个接口就是面向对象的中间件。

尽管面向对象的中间件是一种很强大的规范被广泛应用，但是面对大规模的复杂分布式系统，这些技术也显示出了局限性：

1．同步通信：客户发出调用后，必须等待服务对象完成处理并返回结果后才能继续执行。

2．客户和服务对象的生命周期紧密耦合：客户进程和服务对象进程都必须正常运行，如果由于服务对象崩溃或网络故障导致客户的请求不可达，客户会接收到异常。

为了解决这些问题，出现了面向消息的中间件，它较好地解决了以上的问题。

消息中间件作为一个中间层软件，它为分布式系统中创建、发送、接收消息提供了一套可靠通用的方法，实现了分布式系统中可靠的、高效的、实时的跨平台数据传输。消息中间件减少了开发跨平台和网络协议软件的复杂性，它屏蔽了不同操作系统和网络协议的具体细节，面对规模和复杂度都越来越高的分布式系统，消息中间件技术显示出了它的优越性：

1．采用异步通信模式：发送消息者可以在发送消息后进行其它的工作，不用等待接收者的回应，而接收者也不必在接到消息后立即对发送者的请求进行处理；

2．客户和服务对象生命周期的松耦合关系：客户进程和服务对象进程不要求都正常运行，如果由于服务对象崩溃或者网络故障导致客户的请求不可达，客户不会接收到异常，消息中间件能保证消息不会丢失。

MQ框架非常之多，比较流行的有RabbitMq、ActiveMq、ZeroMq、kafka。这几种MQ到底应该选择哪个？要根据自己项目的业务场景和需求。下面我列出这些MQ之间的对比数据和资料。

MQ 我们可以理解为消息队列，队列我们可以理解为管道。以管道的方式做消息传递。

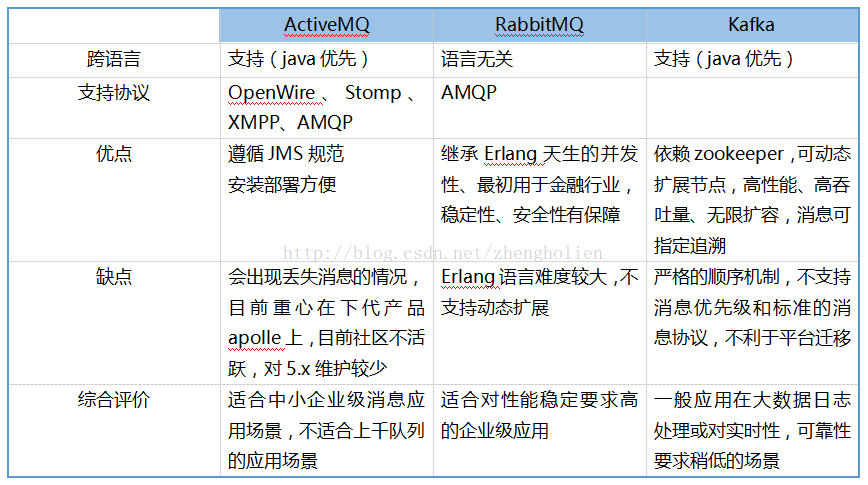
场景：

1.其实我们在双11的时候，当我们凌晨大量的秒杀和抢购商品，然后去结算的时候，就会发现，界面会提醒我们，让我们稍等，以及一些友好的图片文字提醒。而不是像前几年的时代，动不动就页面卡死，报错等来呈现给用户。

在这业务场景中，我们就可以采用队列的机制来处理，因为同时结算就只能达到这么多。

2.在我们平时的超市中购物也是一样，当我们在结算的时候，并不会一窝蜂一样涌入收银台，而是排队结算。这也是队列机制。

对，就是排队。一个接着一个的处理，不能插队。



**RabbitMq、ActiveMq、ZeroMq、kafka之间的比较**

MQ框架非常之多，比较流行的有RabbitMq、ActiveMq、ZeroMq、kafka。这几种MQ到底应该选择哪个？要根据自己项目的业务场景和需求。下面我列出这些MQ之间的对比数据和资料。

第一部分：RabbitMQ,ActiveMq,ZeroMq比较

1、 TPS比较 一

ZeroMq 最好，RabbitMq 次之， ActiveMq 最差。这个结论来自于以下这篇文章。

<http://blog.x-aeon.com/2013/04/10/a-quick-message-queue-benchmark-activemq-rabbitmq-hornetq-qpid-apollo/>

测试环境：

     Model: Dell Studio 1749

     CPU: Intel Core i3 @ 2.40 GHz

     RAM: 4 Gb

     OS: Windows 7 64 bits

其中包括持久化消息和瞬时消息的测试。注意这篇文章里面提到的MQ，都是采用默认配置的，并无调优。

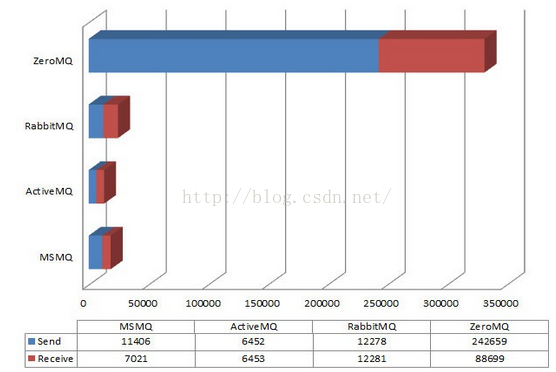


更多的统计图请参看我提供的文章url。

2、TPS比较二

*ZeroMq*最好，*RabbitMq*次之，*ActiveMq*最差。这个结论来自于一下这篇文章。<http://www.cnblogs.com/amityat/archive/2011/08/31/2160293.html>

显示的是发送和接受的每秒钟的消息数。整个过程共产生1百万条1K的消息。测试的执行是在一个Windows Vista上进行的。



3、持久化消息比较

*zeroMq*不支持，*activeMq*和*rabbitMq*都支持。持久化消息主要是指：*MQ down*或者*MQ*所在的服务器*down*了，消息不会丢失的机制。

4、技术点：可靠性、灵活的路由、集群、事务、高可用的队列、消息排序、问题追踪、可视化管理工具、插件系统、社区

      RabbitMq最好，ActiveMq次之，ZeroMq最差。当然ZeroMq也可以做到，不过自己必须手动写代码实现，代码量不小。尤其是可靠性中的：持久性、投递确认、发布者证实和高可用性。

      所以在可靠性和可用性上，RabbitMQ是首选，虽然ActiveMQ也具备，但是它性能不及RabbitMQ。

 5、高并发

从实现语言来看，RabbitMQ最高，原因是它的实现语言是天生具备高并发高可用的erlang语言。

总结：

按照目前网络上的资料，RabbitMQ、activeM、zeroMQ三者中，综合来看，RabbitMQ是首选。下面提供一篇文章，是淘宝使用RabbitMQ的心得，可以参看一些业务场景。

<http://www.docin.com/p-462677246.html>

第二部分：kafka和RabbitMQ的比较

关于这两种MQ的比较，网上的资料并不多，最权威的的是kafka的提交者写一篇文章。<http://www.quora.com/What-are-the-differences-between-Apache-Kafka-and-RabbitMQ>

里面提到的要点：

1、  RabbitMq比kafka成熟，在可用性上，稳定性上，可靠性上，RabbitMq超过kafka

2、  Kafka设计的初衷就是处理日志的，可以看做是一个日志系统，针对性很强，所以它并没有具备一个成熟MQ应该具备的特性

3、  Kafka的性能（吞吐量、tps）比RabbitMq要强，这篇文章的作者认为，两者在这方面没有可比性。

这里在附上两篇文章，也是关于kafka和RabbitMq之间的比较的：

1、http://www.mrhaoting.com/?p=139

2、http://www.liaoqiqi.com/post/227

总结：

两者对比后，我仍然是选择RabbitMq，性能其实是很强劲的，同时具备了一个成熟的MQ应该具有的特性，我们无需重新发明轮子。

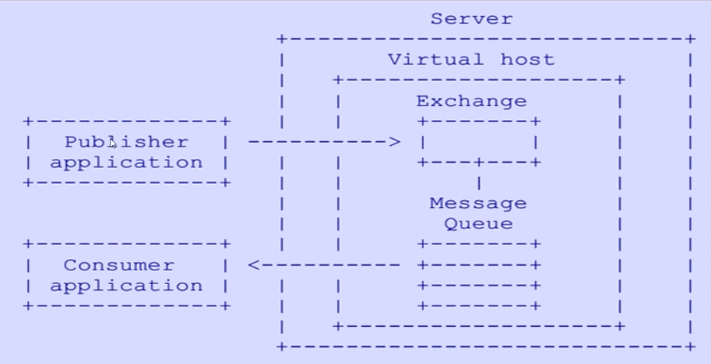
好资料推荐：

            1、最全最给力的kafka博客：http://blog.csdn.net/lizhitao/article/category/2194509

           2、淘宝对rabbitmq的使用：http://www.docin.com/p-462677246.html

### RabbitMQ

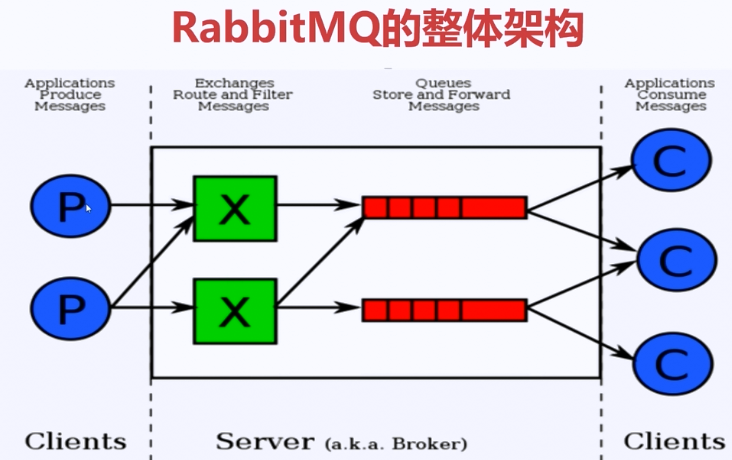
AMQP，即Advanced Message Queuing Protocol，高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。消息中间件主要用于组件之间的解耦，消息的发送者无需知道消息使用者的存在，反之亦然。 AMQP的主要特征是面向消息、队列、路由（包括点对点和发布/订阅）、可靠性、安全。 RabbitMQ是一个开源的AMQP实现，服务器端用Erlang语言编写，支持多种客户端，如：Python、Ruby、.NET、Java、JMS、C、PHP、ActionScript、XMPP、STOMP等，支持AJAX。用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。 下面将重点介绍RabbitMQ中的一些基础概念，了解了这些概念，是使用好RabbitMQ的基础。



Exchange和Message Queue通过路由（routing key）进行关联

RabbitMQ是一个开源的消息代理和队列服务器，用来通过普通协议在完全不同的应用之间共享数据，RabbitMQ是使用Erlang语言编写的，并且RabbitMQ是基于AMQP协议的。

RabbitMQ整体架构



AMQP核心概念

Server：又称Broker，接受客户端的连接，实现AMQP实体服务；

Connection：连接，应用程序与Broker的连接；

Channel：网络通道，几乎所有的操作都在Channel中进行，Channel是进行消息读写的通道。客户端可建立多个Channel，每个Channel代表一个会话任务；

Message：消息，服务器和应用程序之间传送的数据，由Properties和Body组成。Properties可以对消息进行修饰，比如消息的优先级、延迟等高级特性；Body则就是消息体内容；

Virtual Host：虚拟地址，用于进行逻辑隔离，最上层的消息路由。一个Virtual Host里面可以有若干个Exchange和Queue，同一个Virtual Host里面不能有相同名称的Exchange或Queue；

Exchange：交换机，接受消息，根据路由键转发消息到绑定的队列；

Binding：Exchange和Queue之间的虚拟连接，Binging中可以包含routing key；

## JAVA架构师面试题

### Java线程的状态

线程共包括以下 5 种状态:

1. 新建状态(New): 线程对象被创建后，就进入了新建状态。例如，Thread thread = new Thread()。

2. 就绪状态(Runnable): 也被称为“可执行状态”。线程对象被创建后，其它线程调用了该对象的start()方法，从而来启动该线程。例如，thread.start()。处于就绪状态的线程，随时可能被CPU调度执行。

3. 运行状态(Running): 线程获取CPU权限进行执行。需要注意的是，线程只能从就绪状态进入到运行状态。

4. 阻塞状态(Blocked): 阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：

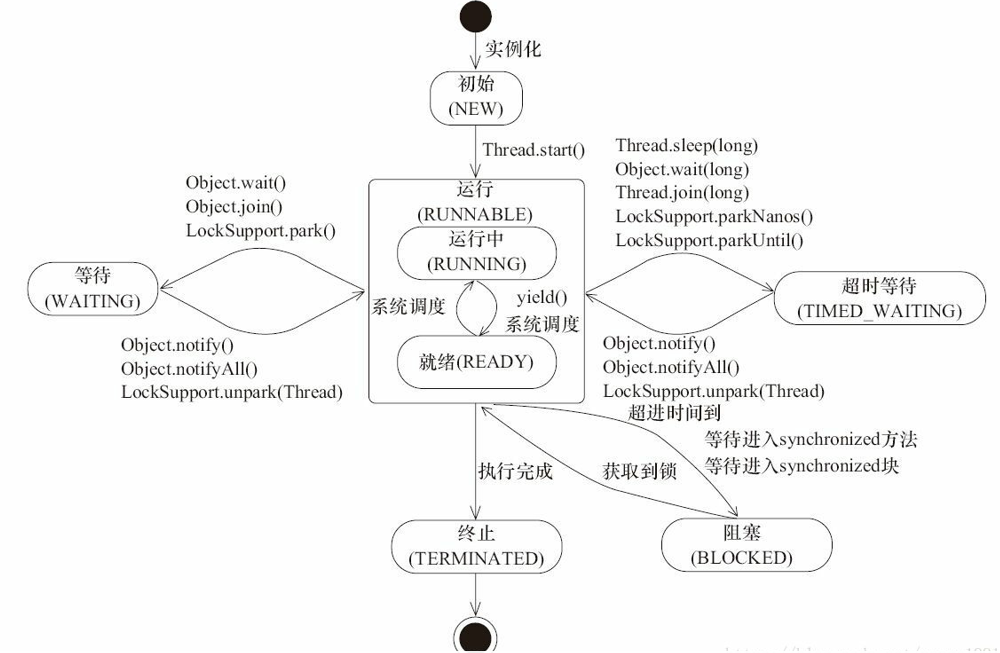
(01) 等待阻塞 -- 通过调用线程的wait()方法，让线程等待某工作的完成。

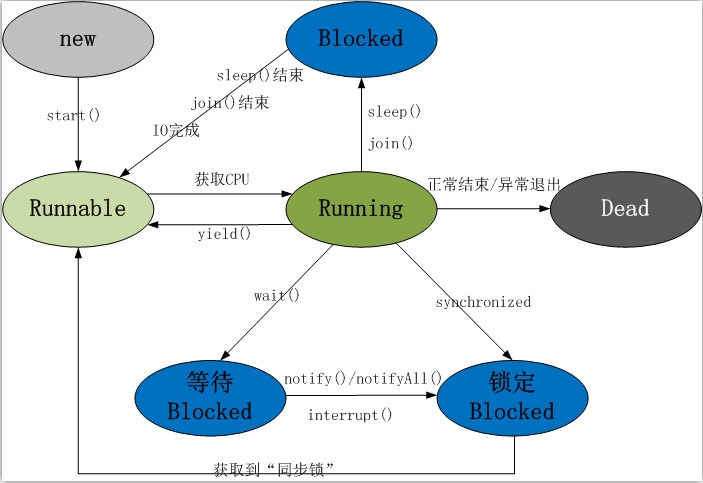
(02) 同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态。

(03) 其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

5. 死亡状态(Dead): 线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

线程的状态图





### 进程和线程的区别

进程是操作系统分配资源（包括cpu）的基本单位。是具有一定独立功能的程序、它是系统进行资源分配和调度的一个独立单位，重点在系统调度和单独的单位，也就是说进程是可以独 立运行的一段程序。

线程是cpu执行的基本单位，多个线程共享系统分配给进程的资源。是进程的一个实体，是CPU调度和分派的基本单位，他是比进程更小的能独立运行的基本单位，线程自己基本上不拥有系统资源。

在运行时，只是暂用一些计数器、寄存器和栈 。

进程和线程的关系：

1、一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程（通常说的主线程）。

2、资源分配给进程，同一进程的所有线程共享该进程的所有资源。

3、线程在执行过程中，需要协作同步。不同进程的线程间要利用消息通信的办法实现同步。

4、处理机分给线程，即真正在处理机上运行的是线程。

5、线程是指进程内的一个执行单元，也是进程内的可调度实体。

从三个角度来剖析二者之间的区别

1、调度：线程作为调度和分配的基本单位，进程作为拥有资源的基本单位。

2、并发性：不仅进程之间可以并发执行，同一个进程的多个线程之间也可以并发执行。

3、拥有资源：进程是拥有资源的一个独立单位，线程不拥有系统资源，但可以访问隶属于进程的资源。

进程间通信：

rpc

mq

socket

线程间通信：

共享内存

wait/notify

pipleline

在多线程编程时，你需要了解以下几个概念：

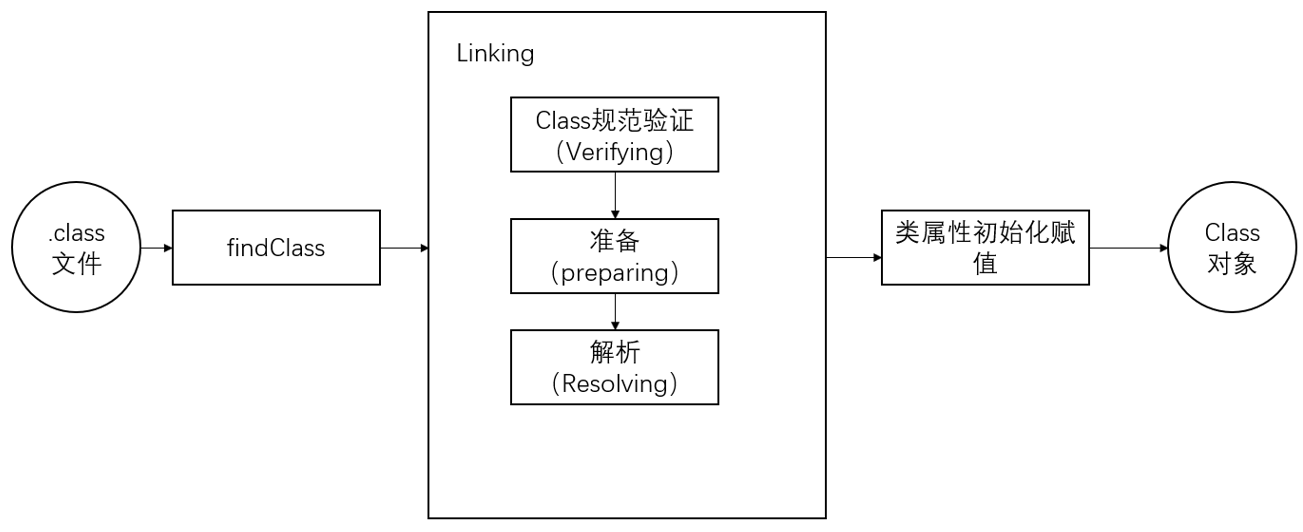
线程同步

线程间通信

线程死锁

线程控制：挂起、停止和恢复

1. JVM如何加载字节码文件



第一阶段是找到.class文件并把这个文件包含的字节码加载到内存中。

第二阶段又可以分为三个步骤，分别是字节码验证、Class类数据结构分析及相应的内存分配和最后的符号表的链接。

第三阶段是类中静态属性和初始化赋值，以及静态块的执行。

### JVM GC，GC算法

1、哪些对象可以被回收。

2、何时回收这些对象。

3、采用什么样的方式回收。

对象存活判断：

判断对象是否存活一般有两种方式：

1、引用计数：每个对象有一个引用计数属性，新增一个引用时计数加1，引用释放时计数减1，计数为0时可以回收。此方法简单，无法解决对象相互循环引用的问题。

2、可达性分析（Reachability Analysis）：从GC Roots开始向下搜索，搜索所走过的路径称为引用链。当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的。不可达对象。

在Java语言中，GC Roots包括：

1、虚拟机栈中引用的对象。

2、方法区中类静态属性实体引用的对象。

3、方法区中常量引用的对象。

4、本地方法栈中JNI引用的对象。

垃圾收集算法：

标记 -清除算法

复制算法

标记-压缩算法

分代收集算法

垃圾收集器：

串行垃圾回收器Serial

并行垃圾回收器

CMS垃圾回收器

### JVM调优

https://blog.csdn.net/wuzhilon88/article/details/49201891

1、堆大小设置

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。我在Windows Server 2003 系统，3.5G物理内存，JDK5.0下测试，最大可设置为1478m。

2、回收器选择

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。

1. 辅助信息

JVM提供了大量命令行参数，打印信息，供调试使用。主要有以下一些：

调优总结

1、年轻代大小选择

a.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。

b.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。

2、年老代大小选择

a.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

b.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，而年老代尽存放长期存活对象。

3、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、清除方式进行回收。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

### JAVA类如何初始化

本阶段负责为类变量赋正确的初始值。（类变量即静态变量）

Java编译器把所有的类变量初始化语句和静态初始化器通通收集到<clinit>方法中，该方法只能被JVM调用，专门承担初始化工作。

初始化一个类必须保证其直接超类已被初始化。

并非所有类都拥有<clinit>()方法。以下类不会拥有<clinit>方法：

该类既没有声明任何类变量，也没有静态初始化语句。

该类声明了类变量，但没有使用类变量初始化语句或静态初始化语句初始化。

该类只包含静态final变量的类变量初始化语句，并且类变量初始化语句是常量表达式。

Java类初始化的时机：

规范定义类的初始化时机为“initialize on first active use”，即“在首次主动使用时初始化”。装载和链接在初始化之前就要完成。

首次主动使用的情形：

创建类的新实例--new，反射，克隆或反序列化；

调用类的静态方法；

操作类和接口的静态字段；（final字段除外）

调用Java的特定的反射方法；

初始化一个类的子类；

指定一个类作为Java虚拟机启动时的初始化类（含有main方法的启动类）。

除了以上6种情形，java中类的其他使用方式都是被动使用，不会导致类的初始化。

Java对象初始化：

编译器为每个类生成至少一个实例初始化方法，即<init>()方法。此方法与源程序里的每个构造方法对应。如果类没有声明构造方法，则生成一个默认构造方法，该方法仅调用父类的默认构造方法，同时生成与该默认构造方法对应的<init>()方法。

<init>()方法内容大概为：

调用另一个<init>()方法（本类的另外一个<init>()方法或父类的<init>()方法）;

初始化实例变量;

与其对应的构造方法内的字节码

加载过程：

1、访问静态变量；main函数是第一个static，然后基类的static，依次往外部类寻找。

2、创建对象；从基类的基本数据类型开始初始化，设置默认值，然后这里是自动加载构造函数，依次往外部类加载。

### 乐观锁与悲观锁

乐观并发控制多数用于数据争用不大、冲突较少的环境中，这种环境中，偶尔回滚事务的成本会低于读取数据时锁定数据的成本，因此可以获得比其他并发控制方法更高的吞吐量。

相对于悲观锁，在对数据库进行处理的时候，乐观锁并不会使用数据库提供的锁机制。一般的实现乐观锁的方式就是记录数据版本。

优点与不足：

　　乐观并发控制相信事务之间的数据竞争(data race)的概率是比较小的，因此尽可能直接做下去，直到提交的时候才去锁定，所以不会产生任何锁和死锁。但如果直接简单这么做，还是有可能会遇到不可预 期的结果，例如两个事务都读取了数据库的某一行，经过修改以后写回数据库，这时就遇到了问题。

悲观并发控制主要用于数据争用激烈的环境，以及发生并发冲突时使用锁保护数据的成本要低于回滚事务的成本的环境中。

使用

MySQL InnoDB中使用悲观锁

要使用悲观锁，我们必须关闭mysql数据库的自动提交属性，因为MySQL默认使用autocommit模式，也就是说，当你执行一个更新操作后，MySQL会立刻将结果进行提交。set autocommit=0;

优点与不足

悲观并发控制实际上是”先取锁再访问”的保守策略，为数据处理的安全提供了保证。但是在效率方面，处理加锁的机制会让数据库产生额外的开销，还有增 加产生死锁的机会；另外，在只读型事务处理中由于不会产生冲突，也没必要使用锁，这样做只能增加系统负载；还有会降低了并行性，一个事务如果锁定了某行数 据，其他事务就必须等待该事务处理完才可以处理那行数

### 如何搭建一个高可用系统

高可用系统，就是说要保证系统在几乎任务时候都要有正常运行，功能正常

我们来看下哪些情况会造成系统不可用

单机系统下的可用性问题，从nginx->tomcat->db/soa来看，单点问题会影响系统高可用，比如要是这个这个链路上其中一个单点挂了，那么整个系统都不可用了

所以引申出来主备/集群模式，防止单点问题

高并发场景下，请求过多也会因为后端瓶颈点引起整个系统down掉，

所以一般情况下应对高并发场景我们会限流，比如今年的英雄联盟抢票，周杰伦抢票

通过采用mq等队列形式削峰，保证后端系统不会down掉

熔断机制

容灾机制，多机房部署

综上所述：

1，主备/集群模式，防止单点

2，限流，削峰，防止后端压力过大

3，熔断机制，类似与限流

4，容灾机制，多机房/异地部署

### JAVA8新特性

Java8 新增了非常多的特性，我们主要讨论以下几个：

Lambda 表达式 − Lambda允许把函数作为一个方法的参数（函数作为参数传递进方法中。

方法引用 − 方法引用提供了非常有用的语法，可以直接引用已有Java类或对象（实例）的方法或构造器。与lambda联合使用，方法引用可以使语言的构造更紧凑简洁，减少冗余代码。

默认方法 − 默认方法就是一个在接口里面有了一个实现的方法。

新工具 − 新的编译工具，如：Nashorn引擎 jjs、 类依赖分析器jdeps。

Stream API −新添加的Stream API（java.util.stream） 把真正的函数式编程风格引入到Java中。

Date Time API − 加强对日期与时间的处理。

Optional 类 − Optional 类已经成为 Java 8 类库的一部分，用来解决空指针异常。

Nashorn, JavaScript 引擎 − Java 8提供了一个新的Nashorn javascript引擎，它允许我们在JVM上运行特定的javascript应用。

### 电商网站商品秒杀功能实现思路分析

高可用、高并发

如何接入客户服务及用户请求

* 注意点：

脱离原站点部署：别把秒杀服务器部署在原商城服务器上。

装监控。

* 关键点：

高可用：双活

高并发：负载均衡、安全过滤

* 布局/设计思路：

对于静态页面：CDN加速、网址隐藏、页面压缩、缓存机制

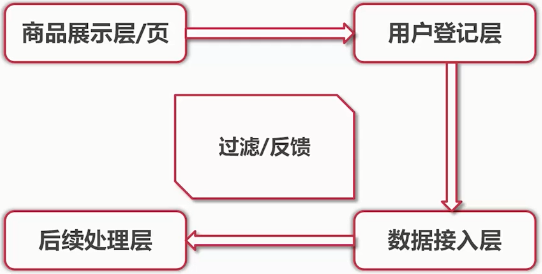
对于动态页面：队列机制、异步传输、资质抢购

架构、构思角度：

业务流程：【前台】商品展示、登记；【后台】数据接入、数据处理

如何分流：策略：分层接待，越到后面，数量越少；CDN加速，不同地区的用户就近选择自己的服务器进行展示，将服务提前发送到离用户最近的服务器上，减少地域访问的延迟。

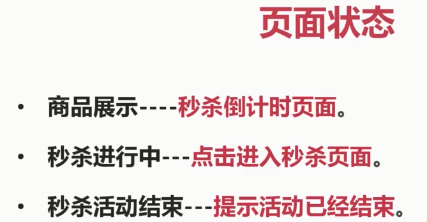
用户组成：识别不同用户，进行安检。

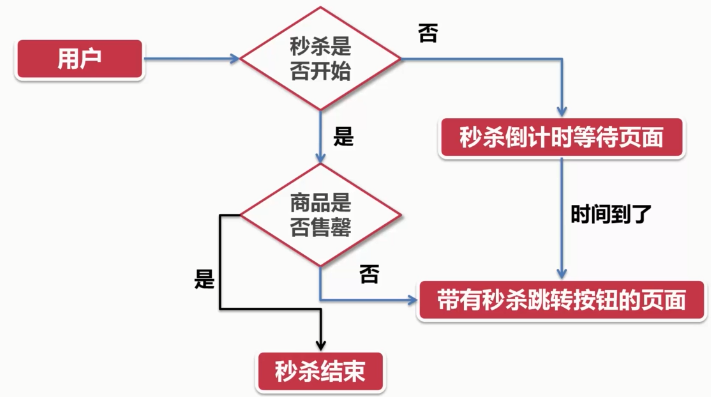


\*后续处理层：数据持久化层。

\*过滤/反馈：层层安检，一定要随时注意数据安全和状态反馈（例如，是否登记成功）

* 商品展示层架构：





最开始，主页显示秒杀准备内容：



第一个界面到第二个界面的内容切换，用linux的定时任务和shell脚本来做。

秒杀开始时，主页显示秒杀开始内容：



点击立即抢购，进行购买。

第二个界面到第三个界面采用服务器端程序（PHP、JAVA）来做。

当活动结束时，显示秒杀结束：



涉及知识点：

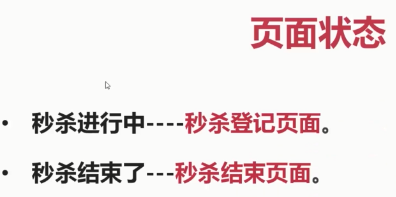
页面/服务器优化、CDN网络加速、隐藏跳转界面、状态切换。

页面优化：CSS、js、image文件的加载、大小，dom处理方式，文件的http请求数等方面进行优化。

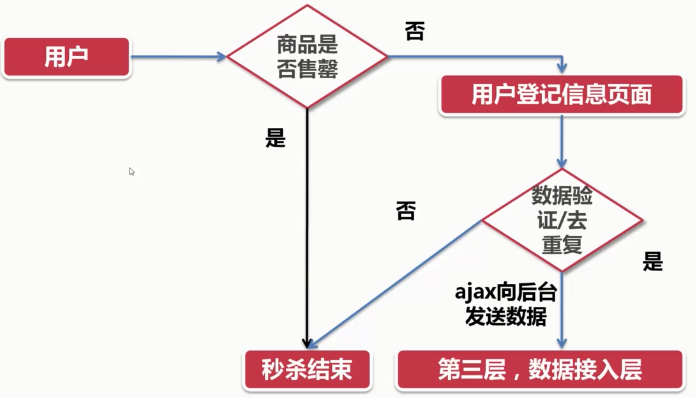
CDN部署：静态文件（css、image）更推荐存放到对象存储（oss）里进行部署。按照云服务商给的官方教程正常配置即可。

* 用户登记层

显示状态被划分成两种。



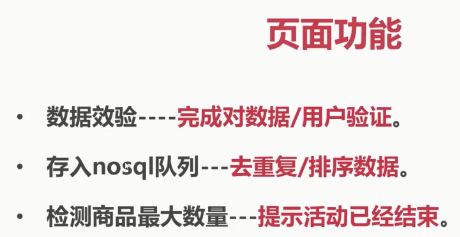
流程：



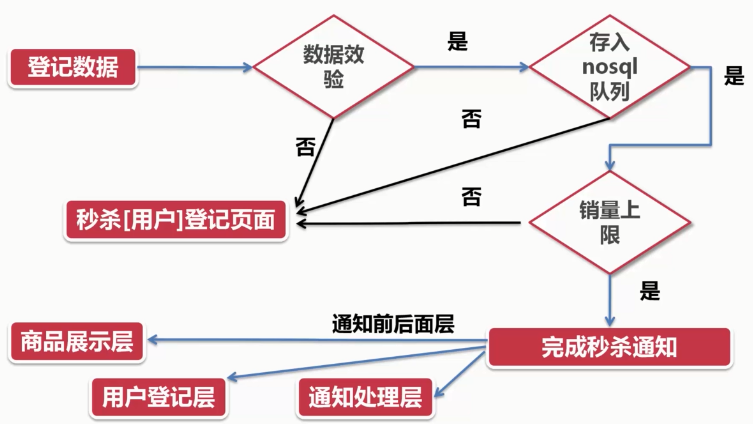
涉及知识点：

Token加/解密、ajax跨域、负载均衡

* 数据接入层



流程：



反馈：只要某个关键节点审核不通过，就返回到上一层，用户登记层，要么重新登记，要么直接禁止登记。

数据校验技术：用到类似于微软序列号生成的加解密算法。

存入nosql队列：使用redis的有序集合。

最大数量监测：使用计数标志位。

通知：通知其他各层进行后续处理。

涉及知识点：

安全过滤、数据校验、存入队列、商品数量检测

* 数据处理层



流程：



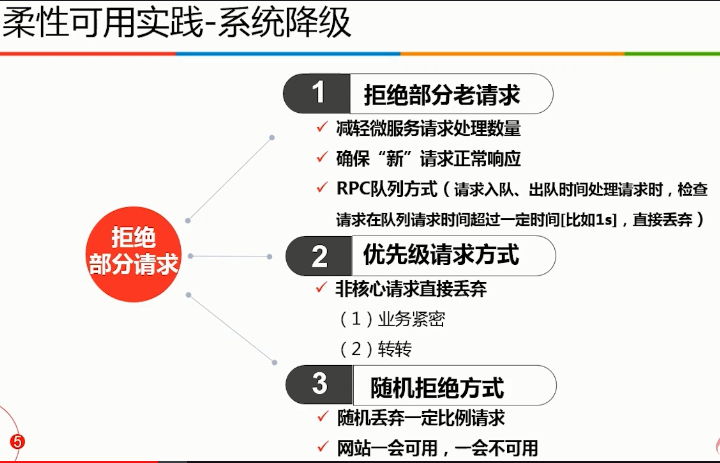
涉及知识点：

数据持久化



系统降级：

1、拒绝部分请求；2、关闭部分服务（业务紧密）





\*高峰期过了之后，需要做消息补齐。

监控：

主要监控两方面：1、机器资源；2、进程状态；

