# 基础项目架构设计

|  |  |  |  |
| --- | --- | --- | --- |
| 版本号 | 修改日期 | 修改人 | 修改内容 |
| 1.0 | 20160421 | 朱瑞泽 | 初始创建 |
| 1.1 | 20160505 | 朱瑞泽 | 补充说明 |

**目录**

[基础项目架构设计 1](#_Toc449090523)

[开发环境 3](#_Toc449090524)

[项目分层 3](#_Toc449090525)

[说明 3](#_Toc449090526)

[结构图 5](#_Toc449090527)

[部署图 6](#_Toc449090528)

## 开发环境

|  |  |  |
| --- | --- | --- |
| JDK | 1.7 |  |
| WEB容器 | Tomcat 7.x | 支持servlet3.0 |
| 版本控制 | SVN/Git | 推荐用Git，SVN只做代码仓库。 |
| 开发工具 | eclipse/myeclipse/IntelliJ IDEA | 推荐使用IntelliJ IDEA |
| 构建工具 | Maven 3.x |  |
| 数据库 | MySQL 5.5+ |  |
| 消息队列 | RabbitMQ |  |
| 缓存 | Redis 3.x |  |
| 文件服务 | FastDFS | 存储图片 |
| 文档数据库 | MongoDB 3.x | 记录日志 |

## 项目分层

### 说明

项目划分为2层，分别为服务层、应用层，依赖的公共服务单独开发并在公共组件里提供接口客户端。

#### *服务层*

* 服务层按模块竖向分隔，彼此之间互相独立，跨模块之间的调用在应用业务Service里完成；
* 模块服务使用Dubbo注册中心暴露服务接口；
* 持久层主要使用Hibernate，通过业务模型自动更新表结构，MyBatis只用于跨表的复杂查询；
* 服务实现类使用Spring的注解事务；

#### *应用层*

* Spring + SpringMVC作为主要开发框架，视图用Freemarker模版引擎生成HTML展示到前端；
* Spring Session过滤器通过Redis共享Session；
* 使用Dubbo注册中心订阅并注入服务供应用业务Service调用；

#### *工程结构*

* Doc为文档目录
* hg-framework-common所有工程的依赖组件
* hg-framework-service-component服务层的依赖组件
* hg-framework-demo示例代码

#### *流程*

* 应用平台接收前端请求或定时任务调用服务层的模块服务，模块服务通过来源切换对应的数据库（每个应用平台有独立的数据库和一个共享总库）来执行Service方法。
* 需要跨模块的异步任务只需要向消息队列发出消息，由执行方监听处理。

### 结构图



### 部署图



## 补充说明

### 项目管理

使用maven管理项目，使用maven命令打包及自动替换不同环境配置。使用nexus建立公司内部JAR包仓库私服，jenkins配置定时自动从GIT拉取代码编绎并自动更新测试环境tomcat，达到持续集成效果。

### 代码版本控制

统一使用GIT，主线master为最新可用版本，原则上只由于合并，开发在新分支上提交，版本发布时在当前分支标记tag，针对发布版本的BUG修复需要在tag的基础上建立分支修改，测试通过后合并。

版本号命名按maven倡议，分为a.b.c-x四段，初始版本号为1.0-SNAPSHOT。

C段为小版本BUG修复， B段为功能性扩展发布， A段为架构或核心变动的发布， X段为稳定性提示，有SNAPSHOT快照版，BETA测试版, RELEASE正式版三种。

### 缓存技术

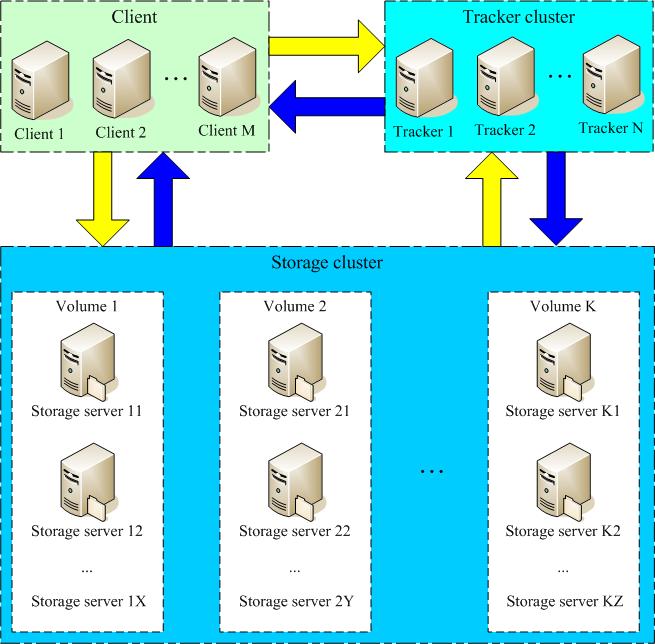
推荐redis，首先是和memcached的操作很类似，功能和支持类型都更丰富，在大量小数据请求的读写并发性能上也比memcached明显要强。还有一个重要考虑是redis自身就支持队列等高级类型，可以用于总线消息的传输和保存。

### 图片资源处理

**Fastfds**：

一个轻量级的分布式文件系统，功能包括：文件存储、文件同步、文件访问（文件上传、文件下载）等，解决了大容量存储和负载均衡的问题。

选择理由，可无限横向扩展，不怕节点故障，动态扩展机器数量，能负载均衡。程序中的代码处理简洁，只需组装文件信息和编写上传操作，保存最终访问地址（由卷和文件名两部分组成的相对地址）。服务端的管理部分都由服务器配置来完成了，而且可以和nginx直接整合，成为http服务。JAVA和PHP都有客户端，都可以上传文件。



fastdfs有三种节点，client客户端是读写文件的程序所在服务器，tracker跟踪器是调度节点的服务器，storage存储节点是存放文件的服务器。三种节点均可以装在同一台机子上。

所有节点都可以由一台或多台服务器构成。跟踪器和存储节点中的服务器均可以随时增加或下线而不会影响线上服务。其中跟踪器中的所有服务器都是对等的，可以根据服务器的压力情况随时增加或减少。

为了支持大容量，存储节点（服务器）采用了分卷（或分组）的组织方式。存储系统由一个或多个卷组成，卷与卷之间的文件是相互独立的，所有卷 的文件容量累加就是整个存储系统中的文件容量。一个卷可以由一台或多台存储服务器组成，一个卷下的存储服务器中的文件都是相同的，卷中的多台存储服务器起 到了冗余备份和负载均衡的作用。

在卷中增加服务器时，同步已有的文件由系统自动完成，同步完成后，系统自动将新增服务器切换到线上提供服务。

当存储空间不足或即将耗尽时，可以动态添加卷。只需要增加一台或多台服务器，并将它们配置为一个新的卷，这样就扩大了存储系统的容量。

Fastfds可以和nginx结合，让用户通过浏览器直接读取系统中存储的文件。

### 数据存储

现在已经不是纯关系数据库处理一切的时代，每个数据存储什么样的数据源需要认真考虑。

数据有以下几种典型情况。

一是必须绝对一致，不可丢失的，保存到关系数据库，并且是同步操作，如保存订单和订单明细。

二是允许一定的操作失败存在，但一旦添加不可丢失的，如各种商品类别、品牌的写操作，个人信息的修改等。也要保存在关系数据库，但可以异步操作，即从操作到生效可以有一定的延时。

三是对二的变种，有时为了用户体验考虑，并不是先保存关系数据库，再异步更新缓存，而是先更新缓存然后返回，之后再异步更新数据库。适用于论坛回贴，这样既能满足高并发情况下的快速保存成功并看到回贴结果，也能接受一定小概率的发贴失败。

对读请求远大于写请求的数据，必须有缓存。

当缓存中的数据只读，不做为任何修改依据时，可以允许有一些不同步，即关系数据库更新后可以稍后更新缓存。

若缓存中的数据是下一个写操作的依据，要有相应措施，可以为缓存中的每个对象加乐观锁版本号，修改完保存时发现版本号过期应撤消操作重新编辑，适用于多人同时编辑一个商品。也可以在最终保存时重新获取即时数据并取得整个集群系统内的独占锁后进行操作，适用于保存订单前的库存变化。

缓存中，要小心使用多对一式的关联对象保存，比如缓存文章时，如果在文章对象内同时缓存文章所属的栏目名，这种情况如果栏目出现改名，将会有非常多的文章对象缓存失效，造成大量的数据库文章重读或者出现脏数据。可以将文章和栏目分开缓存，或者根据系统承受能力和读请求上的压力来判断是否可以接受冗余方案。

### JS模块化工具requirejs

随着网站功能逐渐丰富，网页中的js也变得越来越复杂和臃肿，原有通过script标签来导入一个个的js文件这种方式已经不能满足现在互联网开发模式，我们需要团队协作、模块复用、单元测试等等一系列复杂的需求。

requirejs

RequireJS是一个非常小巧的JavaScript模块载入框架，是AMD规范最好的实现者之一。最新版本的RequireJS压缩后只有14K，堪称非常轻量。它还同时可以和其他的框架协同工作，使用RequireJS必将使您的前端代码质量得以提升。

requirejs能带来什么好处

官方对requirejs的描述：

RequireJS is a JavaScript file and module loader. It is optimized for in-browser use, but it can be used in other JavaScript environments, like Rhino and Node. Using a modular script loader like RequireJS will improve the speed and quality of your code.

大致意思：

在浏览器中可以作为js文件的模块加载器，也可以用在Node和Rhino环境，balabala...。这段话描述了requirejs的基本功能"模块化加载"。

先来看一段常见的场景，通过示例讲解如何运用requirejs

正常编写方式

index.html:

<!DOCTYPE html>

<html>

<head>

<script type="text/javascript" src="a.js"></script>

</head>

<body>

<span>body</span>

</body>

</html>

a.js:

function fun1(){

alert("it works");

}

fun1();

可能你更喜欢这样写

(function(){

function fun1(){

alert("it works");

}

fun1();

})()

第二种方法使用了块作用域来申明function防止污染全局变量，本质还是一样的，当运行上面两种例子时不知道你是否注意到，alert执行的时候，html内容是一片空白的，即<span>body</span>并未被显示，当点击确定后，才出现，这就是JS阻塞浏览器渲染导致的结果。

requirejs写法

当然首先要到requirejs的网站去下载js -> requirejs.rog

index.html:

<!DOCTYPE html>

<html>

<head>

<script type="text/javascript" src="require.js"></script>

<script type="text/javascript">

require(["a"]);

</script>

</head>

<body>

<span>body</span>

</body>

</html>

a.js:

define(function(){

function fun1(){

alert("it works");

}

fun1();

})

浏览器提示了"it works"，说明运行正确，但是有一点不一样，这次浏览器并不是一片空白，body已经出现在页面中，目前为止可以知道requirejs具有如下优点：

防止js加载阻塞页面渲染

使用程序调用的方式加载js，防出现如下丑陋的场景

<script type="text/javascript" src="a.js"></script>

<script type="text/javascript" src="b.js"></script>

<script type="text/javascript" src="c.js"></script>

<script type="text/javascript" src="d.js"></script>

<script type="text/javascript" src="e.js"></script>

<script type="text/javascript" src="f.js"></script>

<script type="text/javascript" src="g.js"></script>

<script type="text/javascript" src="h.js"></script>

<script type="text/javascript" src="i.js"></script>

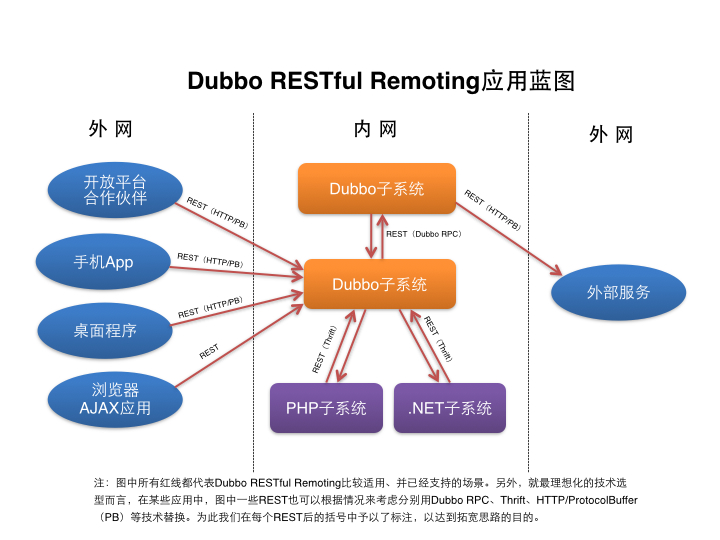
<script type="text/javascript" src="j.js"></script>

### RPC框架Dubbo/Dubbox

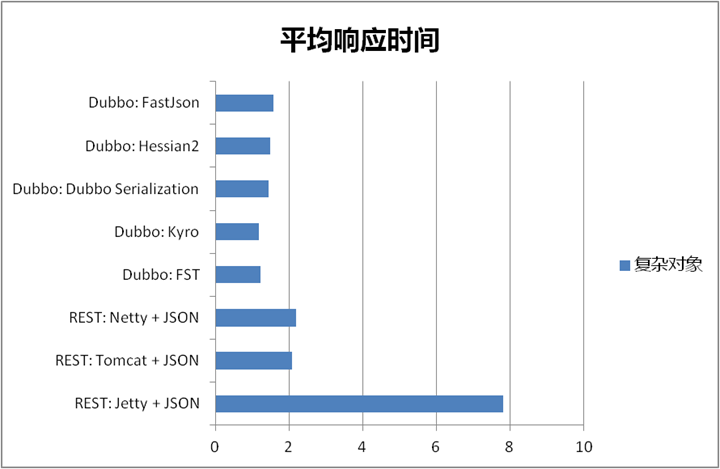
Dubbo是一个被国内很多互联网公司广泛使用的开源分布式服务框架，即使从国际视野来看应该也是一个非常全面的SOA基础框架。作为一个重要的技术研究课题，在当当网我们根据自身的需求，为Dubbo实现了一些新的功能，并将其命名为Dubbox（即Dubbo eXtensions）。

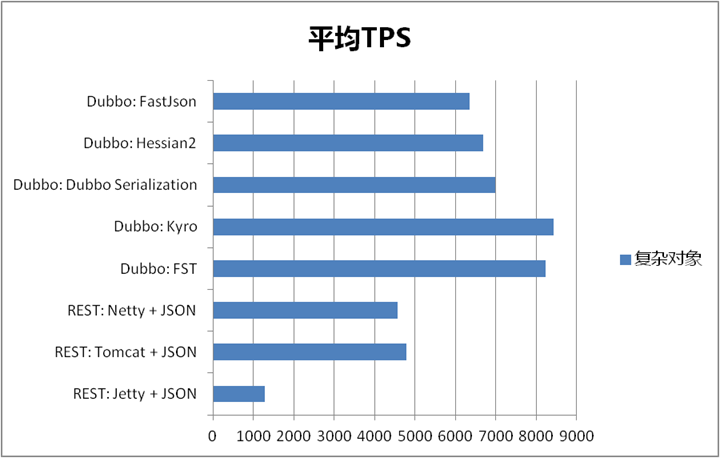
主要的新功能包括：

支持REST风格远程调用（HTTP + JSON/XML)：基于非常成熟的JBoss RestEasy框架，在dubbo中实现了REST风格（HTTP + JSON/XML）的远程调用，以显著简化企业内部的跨语言交互，同时显著简化企业对外的Open API、无线API甚至AJAX服务端等等的开发。事实上，这个REST调用也使得Dubbo可以对当今特别流行的“微服务”架构提供基础性支持。 另外，REST调用也达到了比较高的性能，在基准测试下，HTTP + JSON与Dubbo 2.x默认的RPC协议（即TCP + Hessian2二进制序列化）之间只有1.5倍左右的差距，详见下文的基准测试报告。



支持基于Kryo和FST的Java高效序列化实现：基于当今比较知名的Kryo和FST高性能序列化库，为Dubbo 默认的RPC协议添加新的序列化实现，并优化调整了其序列化体系，比较显著的提高了Dubbo RPC的性能，详见下图和文档中的基准测试报告。





支持基于嵌入式Tomcat的HTTP remoting体系：基于嵌入式tomcat实现dubbo的 HTTP remoting体系（即dubbo-remoting-http），用以逐步取代Dubbo中旧版本的嵌入式Jetty，可以显著的提高REST等的远 程调用性能，并将Servlet API的支持从2.5升级到3.1。（注：除了REST，dubbo中的WebServices、Hessian、HTTP Invoker等协议都基于这个HTTP remoting体系）。

### 分布式系统协调 ZooKeeper

ZooKeeper是Hadoop的正式子项目，它是一个针对大型分布式系统的可靠协调系统，提供的功能包括：配置维护、名字服务、分布式同步、组服务等。ZooKeeper的目标就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

Zookeeper是Google的Chubby一个开源的实现.是高有效和可靠的协同工作系统.Zookeeper能够用来leader选举,配置信息维护等.在一个分布式的环境中,我们需要一个Master实例或存储一些配置信息,确保文件写入的一致性等.Zookeeper能够保证如下3点:

Watches are ordered with respect to other events, other watches, and

asynchronous replies. The ZooKeeper client libraries ensures that

everything is dispatched in order.

A client will see a watch event for a znode it is watching before seeing the new data that corresponds to that znode.

The order of watch events from ZooKeeper corresponds to the order of the updates as seen by the ZooKeeper service.

在Zookeeper中,znode是一个跟Unix文件系统路径相似的节点,可以往这个节点存储或获取数据.如果在创建znode时Flag设置 为EPHEMERAL,那么当这个创建这个znode的节点和Zookeeper失去连接后,这个znode将不再存在在Zookeeper 里.Zookeeper使用Watcher察觉事件信息,当客户端接收到事件信息,比如连接超时,节点数据改变,子节点改变,可以调用相应的行为来处理数 据.Zookeeper的Wiki页面展示了如何使用Zookeeper来处理事件通知,队列,优先队列,锁,共享锁,可撤销的共享锁,两阶段提交.

那么Zookeeper能帮我们作什么事情呢?简单的例子:假设我们我们有个20个搜索引擎的服务器(每个负责总索引中的一部分的搜索任务)和一个 总服务器(负责向这20个搜索引擎的服务器发出搜索请求并合并结果集),一个备用的总服务器(负责当总服务器宕机时替换总服务器),一个web的 cgi(向总服务器发出搜索请求).搜索引擎的服务器中的15个服务器现在提供搜索服务,5个服务器正在生成索引.这20个搜索引擎的服务器经常要让正在 提供搜索服务的服务器停止提供服务开始生成索引,或生成索引的服务器已经把索引生成完成可以搜索提供服务了.使用Zookeeper可以保证总服务器自动 感知有多少提供搜索引擎的服务器并向这些服务器发出搜索请求,备用的总服务器宕机时自动启用备用的总服务器,web的cgi能够自动地获知总服务器的网络 地址变化.这些又如何做到呢?

提供搜索引擎的服务器都在Zookeeper中创建znode,zk.create("/search/nodes/node1",

"hostname".getBytes(), Ids.OPEN\_ACL\_UNSAFE, CreateFlags.EPHEMERAL);

总服务器可以从Zookeeper中获取一个znode的子节点的列表,zk.getChildren("/search/nodes", true);

总服务器遍历这些子节点,并获取子节点的数据生成提供搜索引擎的服务器列表.

当总服务器接收到子节点改变的事件信息,重新返回第二步.

总服务器在Zookeeper中创建节点,zk.create("/search/master", "hostname".getBytes(), Ids.OPEN\_ACL\_UNSAFE, CreateFlags.EPHEMERAL);

备用的总服务器监控Zookeeper中的"/search/master"节点.当这个znode的节点数据改变时,把自己启动变成总服务器,并把自己的网络地址数据放进这个节点.

web的cgi从Zookeeper中"/search/master"节点获取总服务器的网络地址数据并向其发送搜索请求.

web的cgi监控Zookeeper中的"/search/master"节点,当这个znode的节点数据改变时,从这个节点获取总服务器的网络地址数据,并改变当前的总服务器的网络地址.

