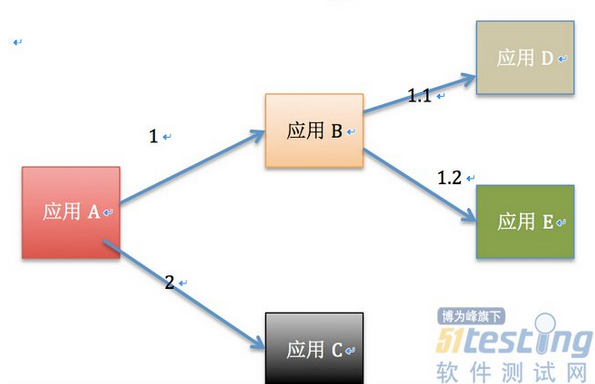
**分布式架构**

## 一、分布式概述

1.分布式系统一定是由**多个节点**组成的系统。其中，节点指的是计算机服务器，而且这些节点一般不是孤立的，而是互通的。

2.这些连通的节点上部署了我们的节点，**并且相互的操作会有协同**。分布式系统对于用户而言，他们面对的就是一个服务器，提供用户需要的服务而已，而实际上这些服务是通过背后的众多服务器组成的一个分布式系统，因此分布式系统看起来像是一个超级计算机一样。

例如淘宝，平时大家都会使用，它本身就是一个分布式系统，我们通过浏览器访问淘宝网站时，这个请求的背后就是一个庞大的分布式系统在为我们提供服务，整个系统中有的负责请求处理，有的负责存储，有的负责计算，最终他们相互协调把最后的结果返回并呈现给用户。



**3 分布式系统往往是把应用拆分成多个应用**，每个团队维护一个应用，应用与应用通过远程过程调用或者消息中间件通信。这种系统的优点是能够做到高内聚低耦合，可以支撑业务的快速发展，缺点则是运维成本大大提高了，系统出了问题，需要全链路排查。如图所示，用户使用应用A，应用A首先需要调用应用B，然后调用应用C，其中应用B又要调用应用D和E，其中任何一个应用挂了，都会影响整个系统的链路。以上只是一个很简单的例子，真实的系统之间的依赖关系往往非常复杂。

**4 拆分的2种方式**

**a.垂直拆分（分布式）**

垂直拆分就是把**一个数据库中不同业务单元的数据分到不同的数据库里面**。比如用户相关信息放到一个库中，订单先关数据放到另一个库中，费用相关信息放到另外一个数据库中等等，这种通过划分不同的库的方式就是数据的垂直划分方式。带来的影响：

(1) 单机的事务ACID特性的保证被打破了。数据到了多机后，原来在单机通过事务进行的处理逻辑会收到很大的影响。我们面临的选择是，要么放弃原来的单机事务，修改实现。要么引入分布式事务机制。

(2) 一些联表操作会变得比较困难，因为数据可能在不同的数据库中了，所以不能很方便的利用数据库自身的join，需要应用或者其他方式来解决。

(3) 靠外键去进行约束的场景会收到影响。

**b.水平拆分（集群）**

水平拆分是根据一定的规则把**同一业务单元的数据拆分到多个数据库**中。比如将订单相关的库，部署到多台服务器中，每台服务器中的订单库的表结构完全一样，但是不同的服务器中的订单库中的数据分别存放了不同的订单相关数据。带来的影响：

(1) 同样具有与垂直划分带来的三种影响。

(2) 依赖单库的自增序列生成唯一id会受影响。

(3) 针对单个逻辑意义上的表的查询要跨库查询了。

**5分布式系统涉及的技术**

1）、远程服务框架：远程服务框架是分布式系统中不可或缺的，对于远程服务框架来说，关键的几个点是：服务消费者、服务提供者、协议（java序列号、反序列化、hessian协议等等）、网络（NIO、多路复用等等）、地址列表。其中协议的选择、网络的优化以及地址列表的获取是难点。目前业界比较有名的开源框架有：Apache Thrift（跨语言）、Dubbo等等。

2）、消息中间件：引入消息中间件是用来降低应用之间的耦合的。对于上游应用来说，它往往会依赖非常多的下游应用，如果全部用同步调用（远程过程调用就是同步调用）的方式，那么上游应用必须等到所有下游应用全部执行成功后，才能返回成功，这种等待往往是不可忍受的。为此，需要将同步异步化，而消息中间件可以实现这种异步化。使用消息中间件后，上游应用只需要向消息服务器发送消息就可以了，它不用等待所有应用执行完，它可以提前返回执行成功，后续，消息服务器会去向这些下游应用发消息，通知它们执行。虽然异步化降低了应用之间的耦合，但也引入了新的问题。比如，下游系统执行的时候挂了，而这时候上游系统其实是返回的成功，这样就导致了状态不一致。要解决这个问题，就需要使用消息中间件本身的重试机制，对于一些系统异常，需要重试直到成功，对于应用本身，又需要对这种重试进行幂等处理，并且使用正向状态机管理整个系统的状态。消息中间件的规范可以参考JMS，业界也有成熟的开源方案，比如ActivityMQ。

3）、wrapper层：warpper层主要是上面两个入口的包装层，进行一些异常管理，日志管理和AOP拦截

4）、MVC框架：这个不用多说了吧，个人比较喜欢spring MVC，有空可以阅读下源代码

5）、分布式缓存

6）、分布式日志

7）、文件存储系统

8）、IOC容器：目前用的最多的是spring容器，对于spring的核心的几个模块，可以参考《spring源码深度解析》

9）、ORM框架：ORM最好使用ibatis，不推荐hibernate，因为ibatis可以控制sql，在大数据量读写下，必须对sql进行优化。至于ibatis本身的架构，有空我会单独写文章。

10）、调度引擎：用于执行异步的定时任务的引擎

11）、分布式数据访问层：正在学习中。。

12）、配置中心：统一的配置管理中心

13）、web容器：tomcat，结合《tomcat源代码分析》和tomcat本身的源代码，应该可以很容易的懂tomcat的架构和实现。

14）、业务逻辑层：流程引擎和规则引擎，配合spring来治理极易变动的业务逻辑，后续有空单独写个流程引擎

## 二、分布式事务

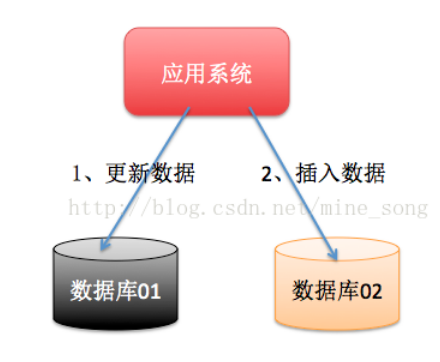
**1 分布式事务就是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。**

**一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。**

**2 分布式事务的产生的原因**

2.1、数据库分库分表

当数据库单表一年产生的数据超过1000W，那么就要考虑分库分表，具体分库分表的原理在此不做解释，以后有空详细说，简单的说就是原来的一个数据库变成了多个数据库。这时候，如果一个操作既访问01库，又访问02库，而且要保证数据的一致性，那么就要用到分布式事务。



2.2、应用分布式化

### 所谓的分布式化，就是业务的服务化。比如原来单机支撑了整个电商网站，现在对整个网站进行拆解，分离出了订单中心、用户中心、库存中心。对于订单中心，有专门的数据库存储订单信息，用户中心也有专门的数据库存储用户信息，库存中心也会有专门的数据库存储库存信息。这时候如果要同时对订单和库存进行操作，那么就会涉及到订单数据库和库存数据库，为了保证数据一致性，就需要用到分布式事务。

### 

### **3 支付**

最经典的场景就是支付了，一笔支付，是对买家账户进行扣款，同时对卖家账户进行加钱，这些操作必须在一个事务里执行，要么全部成功，要么全部失败。而对于买家账户属于买家中心，对应的是买家数据库，而卖家账户属于卖家中心，对应的是卖家数据库，对不同数据库的操作必然需要引入分布式事务。

### **4 在线下单**

买家在电商平台下单，往往会涉及到两个动作，一个是扣库存，第二个是更新订单状态，库存和订单一般属于不同的数据库，需要使用分布式事务保证数据一致性。

**5 分布式事务解决方案**

**（1）基于XA协议的两阶段提交**

XA是一个分布式事务协议，由Tuxedo提出。XA中大致分为两部分：事务管理器和本地资源管理器。其中本地资源管理器往往由数据库实现，比如Oracle、DB2这些商业数据库都实现了XA接口，而事务管理器作为全局的调度者，负责各个本地资源的提交和回滚。XA实现分布式事务的原理如下：

总的来说，XA协议比较简单，而且一旦商业数据库实现了XA协议，使用分布式事务的成本也比较低。但是，XA也有致命的缺点，**那就是性能不理想，特别是在交易下单链路，往往并发量很高，XA无法满足高并发场景**。XA目前在商业数据库支持的比较理想，在mysql数据库中支持的不太理想，mysql的XA实现，没有记录prepare阶段日志，主备切换回导致主库与备库数据不一致。许多nosql也没有支持XA，这让XA的应用场景变得非常狭隘。

因为XA 事务是基于两阶段提交协议的，所以需要有一个事务协调者（transaction manager）来保证所有的事务参与者都完成了准备工作(第一阶段)。如果事务协调者（transaction manager）收到所有参与者都准备好的消息，就会通知所有的事务都可以提交了（第二阶段）。MySQL 在这个XA事务中扮演的是参与者的角色，而不是事务协调者（transaction manager）。

对于一些较大的规模的应用，单个数据源已经无法支撑起庞大的用户量，需要引入多数据源，水平层面进行分库分表，降低DB的负载。另外，跨库意味着单DB的事务就失效了，所以J2EE提出了JTA，分布式事务管理，简单的说，就是分2步提交，实际它有2个容器来管理，一个资源管理器，一个事务管理，

在第一个阶段中，所有参与全局事务的节点都开始准备，告诉事务管理器它们准备好提交了。

第二阶段，事务管理器告诉资源管理器执行commit或者rollback，如果任何一个节点显示不能commit，那么所有的节点全部rollback，下面我们来实现分布式事务：

第一步：XA数据源定义

选定义一个抽象的父类源，这样子类可以直接继承

<!-- 两个数据源的功用配置，方便下面直接引用 -->

<bean id="abstractXADataSource" class="com.atomikos.jdbc.AtomikosDataSourceBean"

destroy-method="close">

<property name="xaDataSourceClassName" value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"/>

<property name="poolSize" value="10" />

<property name="minPoolSize" value="10"/>

<property name="maxPoolSize" value="30"/>

<property name="borrowConnectionTimeout" value="60"/>

<property name="reapTimeout" value="20"/>

<!-- 最大空闲时间 -->

<property name="maxIdleTime" value="60"/>

<property name="maintenanceInterval" value="60" />

<property name="loginTimeout" value="60"/>

<property name="logWriter" value="60"/>

<property name="testQuery">

<value>select 1</value>

</property>

</bean>

master源

<bean id="masterSource" parent="abstractXADataSource">

<property name="uniqueResourceName">

<value>master</value>

</property>

<property name="xaDataSourceClassName" value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource" />

<property name="xaProperties">

<props>

<prop key="user">库用户名</prop>

<prop key="password">库密码</prop>

<prop key="URL">master库连接</prop>

</props>

</property>

</bean>

slave源

<bean id="slaveSource" parent="abstractXADataSource">

<property name="uniqueResourceName">

<value>slave</value>

</property>

<property name="xaDataSourceClassName">

<value>com.mysql.jdbc.jdbc2.optional.MysqlXADataSource</value>

</property>

<property name="xaProperties">

<props>

<prop key="user">库用户名</prop>

<prop key="password">库密码</prop>

<prop key="URL">slave库连接</prop>

</props>

</property>

</bean>

基于spring的AbstractRoutingDataSource动态数据路由定义

<bean id="dataSource" class="com.icz.carcare.datasource.DynamicDataSource">

<property name="targetDataSources">

<map key-type="java.lang.String">

<!-- write 数据更新和实时数据查询-->

<entry key="master" value-ref="masterSource"/>

<!-- read 非实时数据查询-->

<entry key="slave" value-ref="slaveSource"/>

</map>

</property>

<property name="defaultTargetDataSource" ref="masterSource"/> <!-- 默认使用master的数据源 -->

</bean>

mybatis中进行ORM映射

<bean id="sqlSessionFactorya" class="org.mybatis.spring.SqlSessionFactoryBean">

<property name="dataSource" ref="masterSource"/>

<!-- 自动扫描sqlmaps目录, 省掉Configuration.xml里的手工配置 -->

<property name="mapperLocations" value="classpath\*:Msqlmaps/\*.xml" />

</bean>

<bean id="sqlSessionFactoryb" class="org.mybatis.spring.SqlSessionFactoryBean">

<property name="dataSource" ref="slaveSource" />

<!-- 自动扫描sqlmaps目录, 省掉Configuration.xml里的手工配置 -->

<property name="mapperLocations" value="classpath\*:Msqlmaps/\*.xml" />

</bean>

配置sessionTemplate模板

<!-- 配置自定义的SqlSessionTemplate模板，注入相关配置 -->

<bean id="sqlSessionTemplate" class="com.icz.carcare.sqlSessionTemplate.CustomSqlSessionTemplate">

<constructor-arg ref="sqlSessionFactorya" />

<property name="targetSqlSessionFactorys">

<map>

<entry value-ref="sqlSessionFactorya" key="master"/>

<entry value-ref="sqlSessionFactoryb" key="slave"/>

</map>

</property>

</bean>

jta配置

<bean id="atomikosTransactionManager" class="com.atomikos.icatch.jta.UserTransactionManager"

init-method="init" destroy-method="close">

<property name="forceShutdown">

<value>true</value>

</property>

</bean>

<bean id="atomikosUserTransaction" class="com.atomikos.icatch.jta.UserTransactionImp">

<property name="transactionTimeout" value="300" />

</bean>

<bean id="springTransactionManager"

class="org.springframework.transaction.jta.JtaTransactionManager">

<property name="transactionManager">

<ref bean="atomikosTransactionManager" />

</property>

<property name="userTransaction">

<ref bean="atomikosUserTransaction" />

</property>

</bean>

**（2）消息事务+最终一致性**

所谓的消息事务就是基于消息中间件的两阶段提交，本质上是对消息中间件的一种特殊利用，它是将本地事务和发消息放在了一个分布式事务里，保证要么本地操作成功成功并且对外发消息成功，要么两者都失败，开源的RocketMQ就支持这一特性，具体原理如下：

1、A系统向消息中间件发送一条预备消息

2、消息中间件保存预备消息并返回成功

3、A执行本地事务

4、A发送提交消息给消息中间件

**（3）TCC编程模式**

也是两阶段提交的一个变种。TCC提供了一个编程框架，将整个业务逻辑分为三块：Try、Confirm和Cancel三个操作。以在线下单为例，**Try阶段会去扣库存，Confirm阶段则是去更新订单状态，如果更新订单失败，则进入Cancel阶段，会去恢复库存。**总之，**TCC就是通过代码人为实现了两阶段提交**，不同的业务场景所写的代码都不一样，复杂度也不一样，因此，这种模式并不能很好地被复用。

//调用分布式服务返回结果码

int code = httpclient.updateState();

//如果成功跟新本地事务

if('200'==code)

{

dao.update();

}

//如果失败回滚本地事务

else

{

dao.rollback();

}

**（4）webservice保持事务一致性**

A系统调用B系统，callWebservice(true) ,

B系统调用C系统，callWebservice(true)

## 三、SOA技术

**1 面向服务架构soa**以其独特的优势越来越受到企业的重视，它可以根据需求通过网络对松散耦合的粗粒度应用组件进行分布式部署、组合和使用。服务层是SOA的基础，可以直接被应用调用，从而有效控制系统中与软件代理交互的人为依赖性。Soa的开发方法一般主要有开源的**dubbo、dubbox、mule、wso2、cxf，以及付费的oracle soa、ibm soa**等。

SOA是一种粗粒度、松耦合服务架构，服务之间通过简单、精确定义接口进行通讯，不涉及底层编程接口和通讯模型。SOA可以看作是B/S模型、XML（标准通用标记语言的子集）/Web Service技术之后的自然延伸。

**2分布式：**不同模块部署在不同服务器上

作用：分布式解决网站高并发带来问题

**3 集群：**多台服务器部署相同应用构成一个集群

作用：通过负载均衡设备共同对外提供服务

**4 SOA：**业务系统分解为多个组件，让每个组件都独立提供离散，自治，可复用的服务能力，通过服务的组合和编排来实现上层的业务流程

作用：简化维护,降低整体风险,伸缩灵活

**5 微服务：**架构设计概念,各服务间隔离（分布式也是隔离）,自治（分布式依赖整体组合）其它特性(单一职责,边界,异步通信,独立部署)是分布式概念的跟严格执行SOA到微服务架构的演进过程

作用：各服务可独立应用，组合服务也可系统应用

### **6 SOA的好处**

1. 松耦合：由于服务自治，有一定封装边界，服务调用交互是通过发布接口。这意味着应用程序不感兴趣的服务如何被实现。  
   2.位置透明：服务的消费者不必关系服务位于什么地方。  
   3.可在异构平台间复用。可以将遗留系统包装成服务。  
   4.便于测试，能并行开发，较高可靠性和良好可伸缩性。

**7 REST的特性**

它基于HTTP协议，是一种明确构建在客户端/服务端体系结构上的一种风格。特征如下：

1）、网络上的资源都被抽象为资源，这些资源都具有唯一的统一资源标识符(URI：Uniform Resource Identiter)，这些资源都是自我们描述的。这些资源使用HTTP内容标头类型指定。如：XML、JSON、HTML、PNG等。

2）、服务的使用者通过HTTP协议的标准动作(Get、Put、Post、Delete)通过统一的接口对资源进行操作。

3）、对资源进行的操作不会改变它的URI。

4）、客户端、服务端之间的交互是没有状态的。由于这种无状态行，服务端不需要为每个客户端维护Context

**8 SOA的特征**

1、通过网络终结点对外提供服务。

2、粗粒度的服务接口。

**9 REST与SOA的共同点**

REST与SOA两者，虽有不同点，但是他们都作为服务的不同架构风格，具有服务的一般属性。具体如下：

1、统一的服务契约接口与服务接口

2、松散的耦合。

3、只要有权限都可以进行访问

**10 REST与SOA的不同点**

1、REST风格下的，只有一种协议，那就是**HTTP**。而**SOA下的WCF就和多种协议了。如：TCP、HTTP、MSMQ等多种协议**

2、使用方式上的不同。REST只要客户端能够模拟HTTP请求，通过标准的HTTP动作，都可以进行访问。它使用的是HTTPChannel管道，而SOA使用的管道有HTTPChannel、TcpChannel、RPC等多种。

3、REST寄宿时，虽然可以选择多种寄宿方式，但必须有应用服务器的支持。

## 四、分布式跟踪系

1 随着分布式服务架构的流行，特别是微服务等设计理念在系统中的应用，业务的调用链越来越复杂，可以看到，随着服务的拆分，系统的模块变得越来越多，不同的模块可能由不同的团队维护，一个请求可能会涉及到几十个服务的协同处理， 牵扯到多个团队的业务系统，那么如何快速准确的定位到线上故障？

同时，缺乏一个自上而下全局的调用id，如何有效的进行相关的数据分析工作？

对于大型网站系统，如淘宝、京东等电商网站，这些问题尤其突出。

**比较成熟的解决方案是通过调用链的方式，把一次请求调用过程完整的串联起来，这样就实现了对请求调用路径的监控**。

大的互联网公司都有自己的分布式跟踪系统，

比如Google的Dapper，Twitter的zipkin，淘宝的鹰眼，新浪的Watchman，京东的Hydra等。

**2 调用跟踪系统的业务场景**

（1）故障快速定位

通过调用链跟踪，一次请求的逻辑轨迹可以用完整清晰的展示出来。开发中可以在业务日志中添加调用链ID，可以通过调用链结合业务日志快速定位错误信息。

（2）各个调用环节的性能分析

在调用链的各个环节分别添加调用时延，可以分析系统的性能瓶颈，进行针对性的优化。

（3）各个调用环节的可用性，持久层依赖等

通过分析各个环节的平均时延，QPS等信息，可以找到系统的薄弱环节，对一些模块做调整，如数据冗余等。

（4）数据分析等

调用链是一条完整的业务日志，可以得到用户的行为路径，汇总分析应用在很多业务场景。

**3 分布式调用跟踪系统的设计**

**（1）分布式调用跟踪系统的设计目标**

低侵入性，应用透明，低损耗，大范围部署，扩展性，

**（2）埋点和生成日志（上下文信息）**

埋点内容：

TraceId、RPCId、调用的开始时间，调用类型，协议类型，调用方ip和端口，请求的服务名等信息，调用耗时，调用结果，异常信息，消息报文等；

**（3）抓取和存储日志**

+实时的方式去存储日志，主要是分布式日志采集的方式。

典型的解决方案如Flume结合Kafka等MQ。

**（4）分析和统计调用链数据**

一条调用链的日志散落在调用经过的各个服务器上，首先需要按 TraceId 汇总日志，然后按照RpcId 对调用链进行顺序整理。

**调用链数据不要求百分之百准确，可以允许中间的部分日志丢失。**

**（5）计算和展示**

汇总得到各个应用节点的调用链日志后，可以针对性的对各个业务线进行分析。

需要对具体日志进行整理，进一步储存在HBase或者关系型数据库中，可以进行可视化的查询。

## 五、分布式缓存系统

**1 计算机存储**

计算机中的存储器分为硬盘、内存、3级缓存、和2级缓存，随着存储器容量的减少，计算机从存储器中调用数据的速度也在加快。我们常见的数据库，**比如oracle、mysql等，都是将数据存放在磁盘中**。这种存储机制并不足以应对业务系统对数据库产生的增、删、查、改带来的庞大IO压力。所以数据库缓存技术就此诞生，用以实现热点数据的高速缓存，提高应用的响应速度，缓解后端数据库的压力。

**2 分布式数据库缓存**

如今的互联网平台常常要面对百万级的并发用户访问，解决每秒数以千计的并发事务处理，既要实现低延时、保证24\*7\*365的可用性，还要拥有灵活的弹性与可伸缩性...可以说，传统Web 架构已经无法满足用户日益增涨的需求。

而分布式数据库缓存是在内存中管理数据并提供数据的一致性保障，采用数据复制技术实现高可用性，具有较优的扩展性与性能组合。这种数据存储机制，实现了更短的响应时间，同时极大地降低数据库的事务处理负载，极好地解决了大流量情况下数据库服务器和Web服务器之间的瓶颈。

**3 主流的分布式缓存产品**

在实际开发中，比较主流的分布式缓存系统主要有Memcached 和 Redis，这两款系统都采用key-value存储方案，却有各自的优缺点。

**4 Memcached**

Memcached是一个高性能的分布式内存对象缓存系统，用于动态Web应用以减轻数据库负载。Memcached通过在内存中缓存数据和对象来减少读取数据库的次数，从而提高动态、数据库驱动网站的速度。

哈希方式存储；全内存操作；简单文本协议进行数据通信；只操作字符型数据；集群由应用进行控制，采用一致性哈希算法。

数据保存在内存当中的，一旦机器重启，数据会全部丢失；只能操作字符型数据，数据类型贫乏；以root权限运行，而且Memcached本身没有任何权限管理和认证功能，安全性不足；能存储的数据长度有限，最大键长250个字符，储存数据不能超过1M。

**5 Redis**

Redis是一个开源的使用ANSI C语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value数据库，并提供多种语言的API。

Redis支持的数据类型包括：字符串、string、hash、set、sortedset、list；Redis实现持久化的方式：定期将内存快照写入磁盘；写日志；Redis支持主从同步。

单核运行，在存储大数据的时候性能会有降低；不是全内存操作；主从复制是全量复制，对实际的系统运营造成了一定负担。

**6 Memcached VS Redis**

Memcached与Redis都属于内存操作、KV存储的缓存系统，同时都是高度关注数据吞吐量与延迟状况的开源项目。而这两个项目在实际应用中的差异，则决定了其在不同应用场景中的表现。

数据支持不同

Memcached只支持字符型，储存数据不能超过1M；Redis除了字符型还支持List、Hash、Set、有序Set等，长度为512M。

内存管理机制不同

Memcached是全内存的数据缓存系统；Redis支持数据的持久化，会周期性地将更新数据写入磁盘，并实现主从同步。

集群管理不同

Memcached本身并不支持分布式，需要在客户端通过像一致性哈希这样的分布式算法来实现Memcached的分布式存储；Redis可以直接使用Redis Cluster对集群进行管理(Redis Cluster是一个高性能高可用的分布式系统)。

由此可见，Memcached更适合对小型的静态数据进行缓存，对数据的安全性要求不高。而Redis适合处理对数据安全性有一定要求、或对数据结构有一定需求，或者需要存储的大型的数据、需要进行主从复制的项目。

## 六、分布式秒杀系统

**1 秒杀业务分析**

（1）查询商品；（2）创建订单；（3）扣减库存；（4）更新订单；（5）付款；（6）卖家发货

**2 秒杀业务的特性**

（1）低廉价格；（2）大幅推广；（3）瞬时售空；（4）一般是定时上架；（5）时间短、瞬时并发量高

**3 请求接口的合理设计**

一个秒杀或者抢购页面，通常分为2个部分，一个是静态的HTML等内容，另一个就是参与秒杀的Web后台请求接口。

**通常静态HTML等内容，是通过CDN的部署**，一般压力不大，核心瓶颈实际上在后台请求接口上。这个后端接口，必须能够支持高并发请求，同时，非常重要的一点，必须尽可能“快”，在最短的时间里返回用户的请求结果。为了实现尽可能快这一点，接口的后端存储使用内存级别的操作会更好一点。仍然直接面向MySQL之类的存储是不合适的，如果有这种复杂业务的需求，都建议采用异步写入。

**4 高并发的挑战：一定要“快”**

我们通常衡量一个Web系统的吞吐率的指标是**QPS（QueryPer Second，每秒处理请求数）**，解决每秒数万次的高并发场景，这个指标非常关键。举个例子，我们假设处理一个业务请求平均响应时间为100ms，同时，系统内有20台Apache的Web服务器，配置MaxClients为500个（表示Apache的最大连接数目）。

那么，我们的Web系统的理论峰值QPS为（理想化的计算方式）：

20\*500/0.1 = 100000 （10万QPS）

系统似乎很强大，1秒钟可以处理完10万的请求，5w/s的秒杀似乎是“纸老虎”哈。

实际情况，当然没有这么理想。在高并发的实际场景下，机器都处于高负载的状态，在这个时候平均响应时间会被大大增加。

就Web服务器而言，Apache打开了越多的连接进程，CPU需要处理的上下文切换也越多，额外增加了CPU的消耗，然后就直接导致平均响应时间增加。因此上述的**MaxClient**数目，要根据CPU、内存等硬件因素综合考虑，绝对不是越多越好。可以通过Apache自带的abench来测试一下，取一个合适的值。

那么问题来了，假设我们的系统，在5w/s的高并发状态下，平均响应时间从100ms变为250ms（实际情况，甚至更多）：

20\*500/0.25 = 40000 （4万QPS）

于是，我们的系统剩下了4w的QPS，面对5w每秒的请求，中间相差了1w。

**5 高并发下的数据安全**

我们知道在多线程写入同一个文件的时候，会存现“线程安全”的问题（多个线程同时运行同一段代码，如果每次运行结果和单线程运行的结果是一样的，结果和预期相同，就是线程安全的）。如果是MySQL数据库，可以使用它自带的锁机制很好的解决问题，但是，在大规模并发的场景中，是不推荐使用MySQL的。秒杀和抢购的场景中，还有另外一个问题，就是“超发”，如果在这方面控制不慎，会产生发送过多的情况。我们也曾经听说过，某些电商搞抢购活动，买家成功拍下后，商家却不承认订单有效，拒绝发货。这里的问题，也许并不一定是商家奸诈，而是系统技术层面存在超发风险导致的。

**6 超发的原因**

假设某个抢购场景中，我们一共只有100个商品，在最后一刻，我们已经消耗了99个商品，仅剩最后一个。这个时候，系统发来多个并发请求，这批请求读取到的商品余量都是99个，然后都通过了这一个余量判断，最终导致超发。

**7 悲观锁思路**

解决线程安全的思路很多，可以从“悲观锁”的方向开始讨论。

悲观锁，也就是在修改数据的时候，采用锁定状态，排斥外部请求的修改。遇到加锁的状态，就必须等待。虽然上述的方案的确解决了线程安全的问题，但是，别忘记，我们的场景是“高并发”。也就是说，会很多这样的修改请求，每个请求都需要等待“锁”，某些线程可能永远都没有机会抢到这个“锁”，这种请求就会死在那里。同时，这种请求会很多，瞬间增大系统的平均响应时间，结果是可用连接数被耗尽，系统陷入异常。

**8 FIFO队列思路**

那好，那么我们稍微修改一下上面的场景，我们直接将请求放入队列中的，采用FIFO（First Input First Output，先进先出），这样的话，我们就不会导致某些请求永远获取不到锁。看到这里，是不是有点强行将多线程变成单线程的感觉哈。

然后，我们现在解决了锁的问题，全部请求采用“先进先出”的队列方式来处理。那么新的问题来了，高并发的场景下，因为请求很多，很可能一瞬间将队列内存“撑爆”，然后系统又陷入到了异常状态。或者设计一个极大的内存队列，也是一种方案，但是，系统处理完一个队列内请求的速度根本无法和疯狂涌入队列中的数目相比。也就是说，队列内的请求会越积累越多，最终Web系统平均响应时候还是会大幅下降，系统还是陷入异常。

**9 乐观锁思路**

这个时候，我们就可以讨论一下“乐观锁”的思路了。乐观锁，是相对于“悲观锁”采用更为宽松的加锁机制，大都是采用带版本号（Version）更新。实现就是，这个数据所有请求都有资格去修改，但会获得一个该数据的版本号，只有版本号符合的才能更新成功，其他的返回抢购失败。这样的话，我们就不需要考虑队列的问题，不过，它会增大CPU的计算开销。但是，综合来说，这是一个比较好的解决方案。

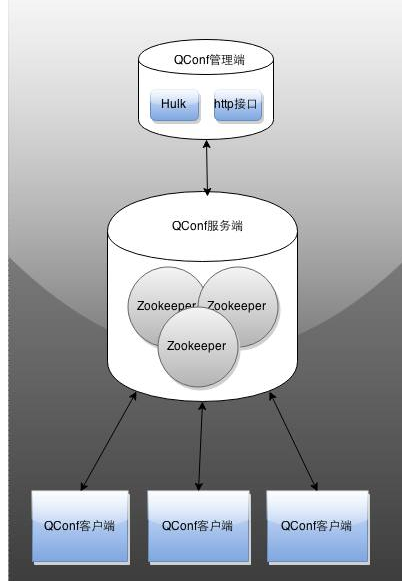
有很多软件和服务都“乐观锁”功能的支持，例如**Redis中的watch**就是其中之一。通过这个实现，我们保证了数据的安全。WATCH命令可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行。监控一直持续到EXEC命令（事务中的命令是在EXEC之后才执行的，所以在MULTI命令后可以修改WATCH监控的键值）

**10 ConcurrentHashMap思路**

ConcurrentHashMap支持同步和高并发

## 七、分布式配置系统

**1 在分布式环境中，出于负载、容错等种种原因**，几乎所有的服务都需要在不同的机器节点上部署多个实例。同时，业务项目中总少不了各种类型的配置文件。这种情况下，有时仅仅是一个配置内容的修改，便需要重新进行代码提交svn/git，打包，分发上线的流程。当部署的机器有很多时，分发上线本身也是一个很繁杂的工作。而配置文件的修改频率又远远大于代码本身。 追本溯源，我们认为所有的这些麻烦是由于我们对配置和代码在管理和发布过程中不加区分造成的。配置本身源于代码，是为了提高代码的灵活性而提取出来的一些经常变化的或需要定制的内容，而正是配置的这种天生的变化特征给我们带了很大的麻烦。因此，开发了分布式配置管理系统QConf，并依托QConf在360内部提供了一整套配置管理服务，QConf致力于将配置内容从代码中完全分离出来，及时可靠高效地提供配置访问和更新服务。



上图展示的是QConf的基本结构，从角色上划分主要包括QConf客户端，QConf服务端和QConf管理端。

**2 QConf使用ZooKeeper集群作为服务端提供服务。**众所周知，**ZooKeeper是一套分布式应用程序协调服务，根据上面提到的对配置内容的定位，我们认为可以将单条配置内容直接存储在ZooKeeper的一个ZNode上**，并利用ZooKeeper的Watch监听功能实现配置变化时对客户端的及时通知。 按照ZooKeeper的设计目标，其只提供最基础的功能，包括顺序一致，原子性，单一系统镜像，可靠性和及时性。另外Zookeeper还有如下特点：

类文件系统的节点组织；稳定无单点问题；订阅通知机制

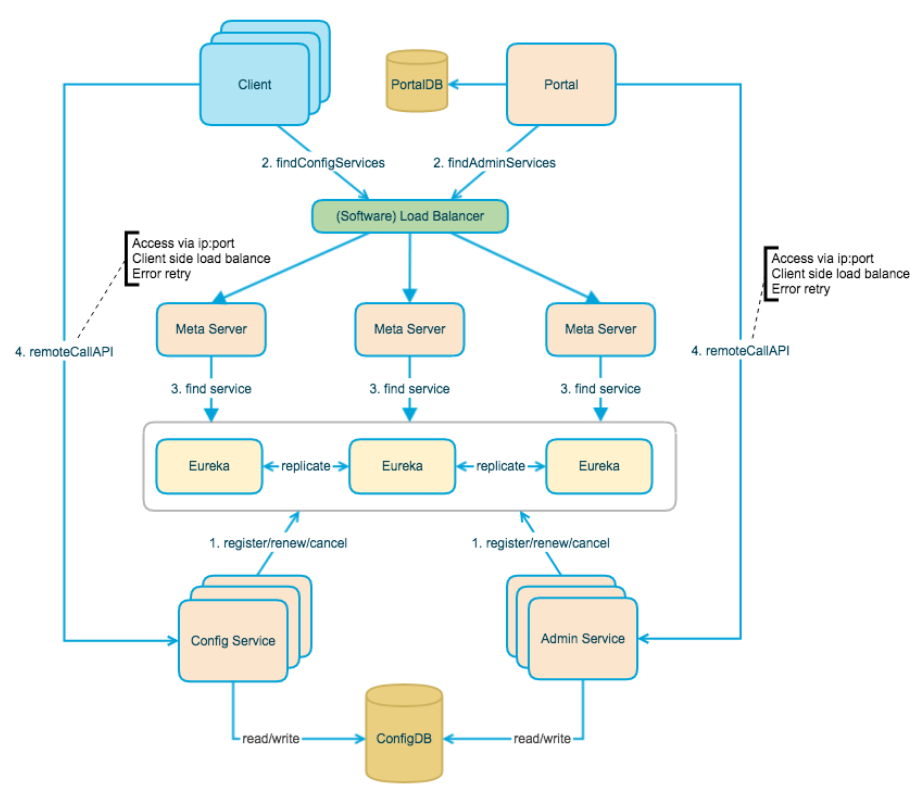
**3 用户进程通过消息队列发送的需获取key；**

ZooKeeper 配置修改删除等触发Watcher通知，需更新；

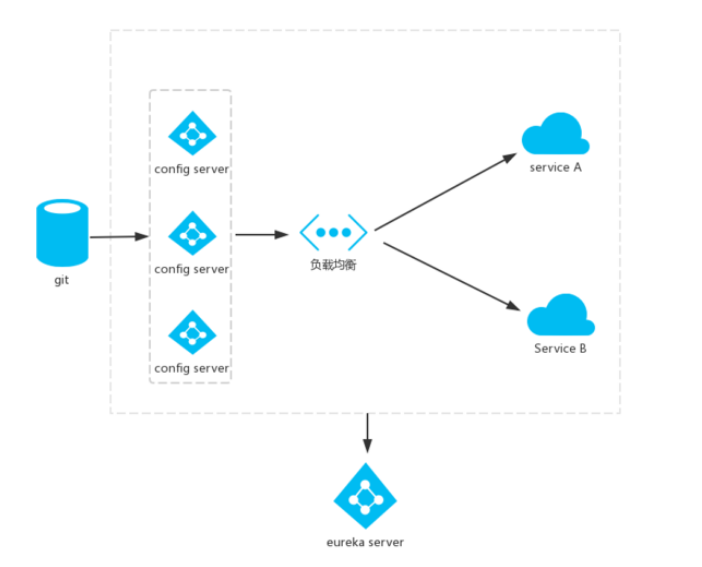
为了消除watcher丢失造成的不一致，需要定时对共享内存中的所有配置重新注册watcher，此时可能会需要更新；

发生agent重启、网络中断、ZooKeeper会话过期等异常情况之后，需重新拉数据，此时可能需要更新。

**4 apollo分布式配置**



**5 Spring Cloud Config**



## 八、分布式消息系统

1 **使用消息系统**

**@解耦**

在项目启动之初来预测将来项目会碰到什么需求，是极其困难的。消息系统在处理过程中间插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口。

这允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。

**@冗余**

有些情况下，处理数据的过程会失败。除非数据被持久化，否则将造成丢失。消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的"插入-获取-删除"范式中，在把一个消息从队列中删除之前，需要你的处理系统明确的指出该消息已经被处理完毕，从而确保你的数据被安全的保存直到你使用完毕。

**@扩展性**

因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。不需要改变代码、不需要调节参数。扩展就像调大电力按钮一样简单。

**@灵活性 & 峰值处理能力**

在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见；如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。

**@可恢复性**

系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。

**@顺序保证**

在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。Kafka保证一个Partition内的消息的有序性。

**@缓冲**

在任何重要的系统中，都会有需要不同的处理时间的元素。例如，加载一张图片比应用过滤器花费更少的时间。消息队列通过一个缓冲层来帮助任务最高效率的执行———写入队列的处理会尽可能的快速。该缓冲有助于控制和优化数据流经过系统的速度。

**@异步通信**

很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

**2 常用Message Queue对比**

**RabbitMQ**

RabbitMQ是使用Erlang编写的一个开源的消息队列，本身支持很多的协议：AMQP，XMPP, SMTP, STOMP，也正因如此，它非常重量级，更适合于企业级的开发。同时实现了Broker构架，这意味着消息在发送给客户端时先在中心队列排队。对路由，负载均衡或者数据持久化都有很好的支持。

**Redis**

Redis是一个基于Key-Value对的NoSQL数据库，开发维护很活跃。虽然它是一个Key-Value数据库存储系统，但它本身支持MQ功能，所以完全可以当做一个轻量级的队列服务来使用。对于RabbitMQ和Redis的入队和出队操作，各执行100万次，每10万次记录一次执行时间。测试数据分为128Bytes、512Bytes、1K和10K四个不同大小的数据。实验表明：入队时，当数据比较小时Redis的性能要高于RabbitMQ，而如果数据大小超过了10K，Redis则慢的无法忍受；出队时，无论数据大小，Redis都表现出非常好的性能，而RabbitMQ的出队性能则远低于Redis。

**ZeroMQ**

ZeroMQ号称最快的消息队列系统，尤其针对大吞吐量的需求场景。ZeroMQ能够实现RabbitMQ不擅长的高级/复杂的队列，但是开发人员需要自己组合多种技术框架，技术上的复杂度是对这MQ能够应用成功的挑战。ZeroMQ具有一个独特的非中间件的模式，你不需要安装和运行一个消息服务器或中间件，因为你的应用程序将扮演这个服务器角色。你只需要简单的引用ZeroMQ程序库，可以使用NuGet安装，然后你就可以愉快的在应用程序之间发送消息了。但是ZeroMQ仅提供非持久性的队列，也就是说如果宕机，数据将会丢失。其中，Twitter的Storm 0.9.0以前的版本中默认使用ZeroMQ作为数据流的传输（Storm从0.9版本开始同时支持ZeroMQ和Netty作为传输模块）。

**ActiveMQ**

ActiveMQ是Apache下的一个子项目。 类似于ZeroMQ，它能够以代理人和点对点的技术实现队列。同时类似于RabbitMQ，它少量代码就可以高效地实现高级应用场景。

**Kafka/Jafka**

Kafka是Apache下的一个子项目，是一个高性能跨语言分布式发布/订阅消息队列系统，而Jafka是在Kafka之上孵化而来的，即Kafka的一个升级版。具有以下特性：快速持久化，可以在O(1)的系统开销下进行消息持久化；高吞吐，在一台普通的服务器上既可以达到**10W/s**的吞吐速率；完全的分布式系统，Broker、Producer、Consumer都原生自动支持分布式，自动实现负载均衡；支持Hadoop数据并行加载，对于像Hadoop的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。Kafka通过Hadoop的并行加载机制统一了在线和离线的消息处理。Apache Kafka相对于ActiveMQ是一个非常轻量级的消息系统，除了性能非常好之外，还是一个工作良好的分布式系统。

## 九、分布式压测系统

1 压力测试是用来检测系统承载能力的有效手段。在系统规模较小的时候，在一台空闲的服务器上使用ab，wrk，siege等工具发起一定量的并发请求即可得到一个初步的测试结果。

但在系统复杂度逐步提高，特别是引入了负载均衡，微服务等架构后，单机的压力测试方案不再可用，企业需要搭建分布式测试集群或者付费使用外部供应商提供的压力测试服务。

2 基于**Kubernetes**的动态资源调度功能，以及Kubernetes集群的动态伸缩特性，我们可以充分利用集群内的闲置计算资源，在需要进行压力测试时启动测试节点，在测试结束后释放资源给其他业务，甚至通过集群扩容和缩容临时为压力测试提供更多的计算资源。

## 分布式精华

**高性能，高可用，易扩展，可伸缩**

**1 单机架构**

服务器，数据库，文件同一台物理机

**2 集群架构**

nginx集群，服务器集群，redis集群，文件服务器集群，mysql主从同步，CDN加速

**3 分布式架构**

垂直拆分，首页，用户，搜索，广告，购物，订单，降低耦合，分而治之，提升体统容错性。

**4 服务化架构**

子系统共享业务重复建设，资源连接受限，多人维护一个系统，服务两两强耦合，缺少统计监控和治理的手段，缺少熔断机制，所以SOA改造刻不容缓。

**服务治理**

服务的动态注册和发现，扩容评估，降级处理

**dubbo-admin**

路由规则，动态配置，服务降级，访问控制，权重调节

**分布式跟踪系统**

可视化一个完整请求调用链，比如鹰眼，rpcFilter负责拦截请求，traceid，spanid，rpcContext设置到RpcInvacation，后面get。

**大流量限流消峰**

比如双11，如果不加限制，容易资源耗尽，内存爆，数据库吞吐量降低。

双11手段～扩容，cdn动静分离，缓存，服务降级，限流

**常见限流算法**

令牌桶～平均速率向桶内放入token，桶容积不变，获取token失败则限流，成功处理。

漏桶～任意速率放入，容积固定，固定速率消费，水漫限流。

nginx实现限流～limit\_zone。

**消峰**

时间分片～抢购时间多少次

答题认证～错开高峰期

消息队列

**分表分库带来的问题**

acid，连表查询，序列id

**traceid的生成**

snowflake是一个优秀的分布式Id生成方案。redis单线程inscr