支付宝的架构到底有多牛逼！还没看完我就跪了！

汤波 ẆṺąkṺXęnj ⁵ ⁶ 月⁵ ⁶ 日

点击上方“zhisheng”，选择“设为星标”
每日获取精彩技术分享

来源：ő̈XXm̃ Xı⌊ônjĕ b

自 ⁶ ⁰ ⁰ ⁾ 年双 ⁵ ⁵ 以来，在每年双 ⁵ ⁵ 超大规模流量的冲击上，蚂蚁金服都会不断突破现有技术的极限。⁶ ⁰ ⁵ ⁰ 年双 ⁵ ⁵ 的支付峰值为 ⁶ 万笔ı分钟，到 ⁶ ⁰ ⁵ ⁽ 年双 ⁵ ⁵ 时这个数字变为了 ⁶ ⁺ ŋ̂ 万笔ı秒。

⁶ ⁰ ⁵ ⁾ 年双 ⁵ ⁵ 的支付峰值为 ⁸ ⁾ !万笔ı秒，⁶ ⁰ ⁵ ⁿ 年双 ⁵ ⁵ 支付峰值为 ⁺ ⁸ ŋ̂ 万笔ı秒，创下新纪录，是 ⁶ ⁰ ⁰ ⁿ 年第一次双 ⁵ ⁵ 的 ⁵ ⁷ ⁻ ⁰ !倍。

在如此之大的支付 ĶĒg̦ 背后除了削峰等锦上添花的应用级优化，最解渴最实质的招数当数基于分库分表的单元化了，蚂蚁技术称之为 ‡k b（逻辑数据中心）。

本文不打算讨论具体到代码级的分析，而是尝试用最简单的描述来说明其中最大快人心的原理。

我想关心分布式系统设计的人都曾被下面这些问题所困扰过：

- 支付宝海量支付背后最解渴的设计是啥？换句话说，实现支付宝高 ĶĒg̦ 的最关键的设计是啥？
- ‡k b 是啥？‡k b 怎么实现异地多活和异地灾备的？

- b2Ē 魔咒到底是啥？Ē 到底怎么理解？
- 什么是脑裂？跟 b2Ē 又是啥关系？
- 什么是 Ē2ș˙ g，它解决了啥问题？
- Ē2ș˙ g 和 b2Ē 啥关系？Ē2ș˙ g 可以逃脱 b2Ē 魔咒么？
- ˙ ⌊X ę6 ķX 能逃脱 b2Ē 魔咒么？

如果你对这些感兴趣，不妨看一场赤裸裸的论述，拒绝使用晦涩难懂的词汇，直面最本质的逻辑。

‡k b 和单元化

‡k b（Ë njə⌊ l ṁ ⌊XęṁXĮ）是相对于传统的（Æęṁ XĮ ęXṁk ṁ b XęṁXĮ ťńÆb）提出的，逻辑数据中心所表达的中心思想是无论物理结构如何的分布，整个数据中心在逻辑上是协同和统一的。

这句话暗含的是强大的体系设计，分布式系统的挑战就在于整体协同工作（可用性，分区容忍性）和统一（一致性）。

单元化是大型互联网系统的必然选择趋势，举个最最通俗的例子来说明单元化。

我们总是说 ĶĒg 很难提升，确实任何一家互联网公司（比如淘宝、携程、新浪）它的交易 ĶĒg 顶多以十万计量（平均水平），很难往上串了。

因为数据库存储层瓶颈的存在再多水平扩展的服务器都无法绕开，而从整个互联网的视角看，全世界电商的交易 ĶĒg 可以轻松上亿。

这个例子带给我们一些思考：为啥几家互联网公司的 ĶĒg 之和可以那么大，服务的用户数规模也极为吓人，而单个互联网公司的 ĶĒg 却很难提升？

究其本质，每家互联网公司都是一个独立的大型单元，他们各自服务自己的用户互不干扰。

这就是单元化的基本特性，任何一家互联网公司，其想要成倍的扩大自己系统的服务能力，都必然会走向单元化之路。

它的本质是分治，我们把广大的用户分为若干部分，同时把系统复制多份，每一份都独立部署，每一份系统都服务特定的一群用户。

以淘宝举例，这样之后，就会有很多个淘宝系统分别为不同的用户服务，每个淘宝系统都做到十万 ĶĒg 的话，Ô 个这样的系统就可以轻松做到 Ôŭ十万的 ĶĒg 了。

‡k b 实现的关键就在于单元化系统架构设计，所以在蚂蚁内部，‡k b 和单元化是不分家的，这也是很多同学比较困扰的地方，看似没啥关系，实则是单元化体系设计成就了 ‡k b 。

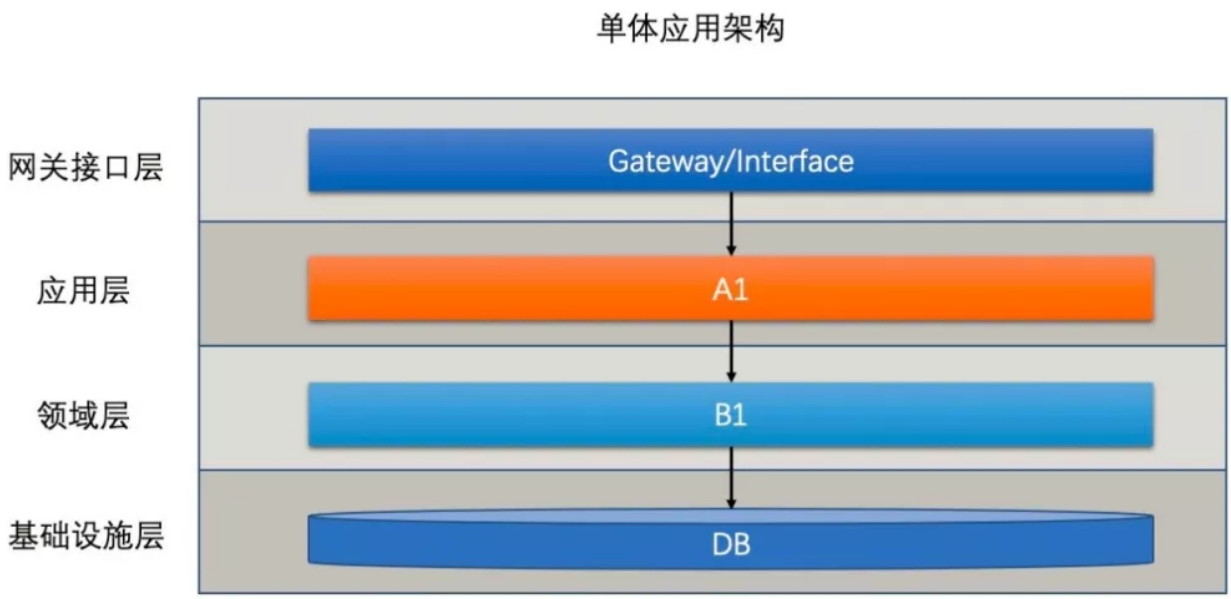
小结：分库分表解决的最大痛点是数据库单点瓶颈，这个瓶颈的产生是由现代二进制数据存储体系决定的（即 Æ 速度）。

单元化只是分库分表后系统部署的一种方式，这种部署模式在灾备方面也发挥了极大的优势。

系统架构演化史

几乎任何规模的互联网公司，都有自己的系统架构迭代和更新，大致的演化路径都大同小异。

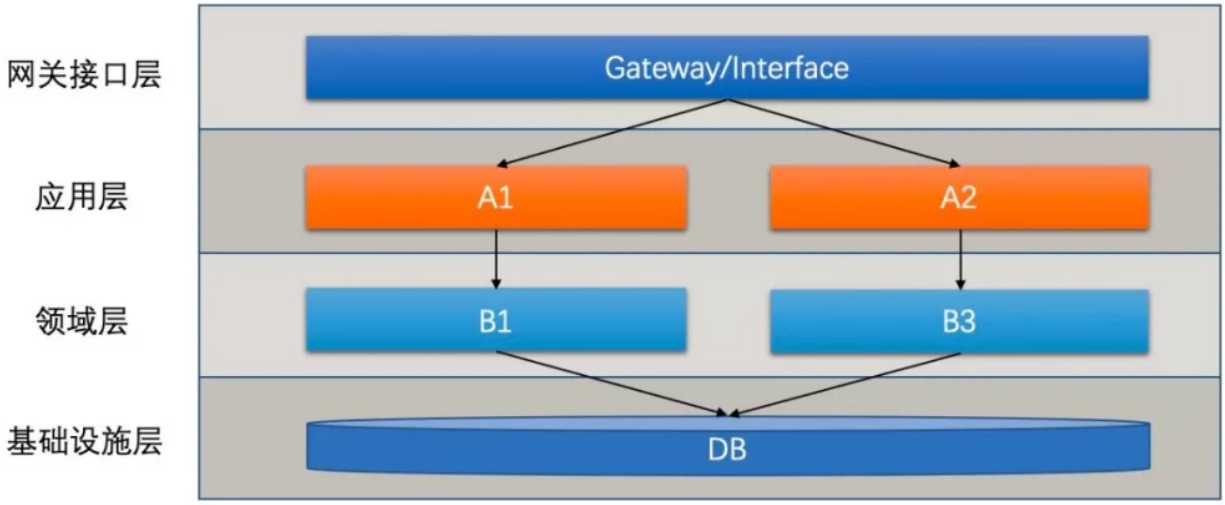
最早一般为了业务快速上线，所有功能都会放到一个应用里，系统架构如下图所示：



这样的架构显然是有问题的，单机有着明显的单点效应，单机的容量和性能都是很局限的，而使用中小型机会带来大量的浪费。

随着业务发展，这个矛盾逐渐转变为主要矛盾，因此工程师们采用了以下架构：

单数据库实例的分布式服务架构

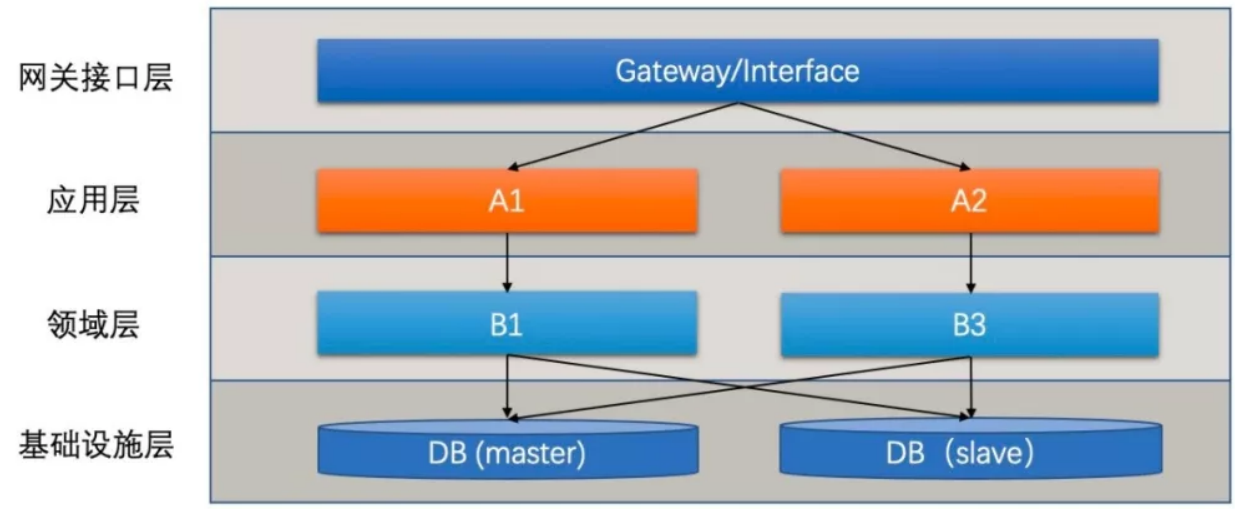


这是整个公司第一次触碰到分布式，也就是对某个应用进行了水平扩容，它将多个微机的计算能力团结了起来，可以完胜同等价格的中小型机器。

慢慢的，大家发现，应用服务器 bĒń 都很正常了，但是还是有很多慢请求，究其原因，是因为单点数据库带来了性能瓶颈。

于是程序员们决定使用主从结构的数据库集群，如下图所示：

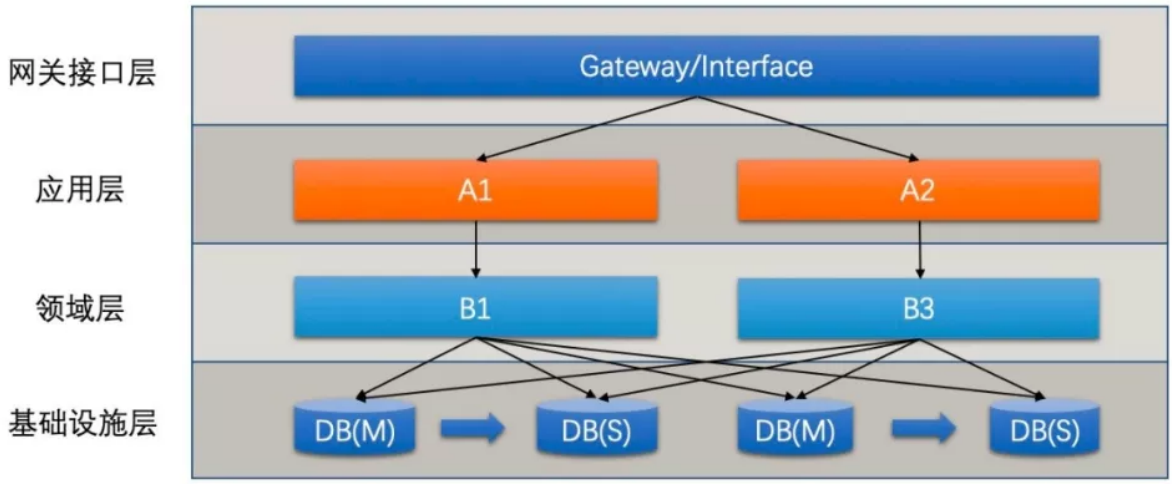
读写分离的主从DB分布式服务架构



其中大部分读操作可以直接访问从库，从而减轻主库的压力。然而这种方式还是无法解决写瓶颈，写依旧需要主库来处理，当业务量量级再次增高时，写已经变成刻不容缓的待处理瓶

这时候，分库分表方案出现了：

分库分表的分布式服务架构



分库分表不仅可以对相同的库进行拆分，还可以对相同的表进行拆分，对表进行拆分的方式叫做水平拆分。

不同功能的表放到不同的库里，一般对应的是垂直拆分（按照业务功能进行拆分），此时一般还对应了微服务化。

这种方法做到极致基本能支撑 ĶĒg 在万级甚至更高的访问量了。然而随着相同应用扩展的越多，每个数据库的链接数也巨量增长，这让数据库本身的资源成为了瓶颈。

这个问题产生的本质是全量数据无差别的分享了所有的应用资源，比如 2 用户的请求在负载均衡的分配下可能分配到任意一个应用服务器上，因而所有应用全部都要链接 2 用户所在的分库，数据库连接数就变成笛卡尔乘积了。

在本质点说，这种模式的资源隔离性还不够彻底。要解决这个问题，就需要把识别用户分库的逻辑往上层移动，从数据库层移动到路由网关层。

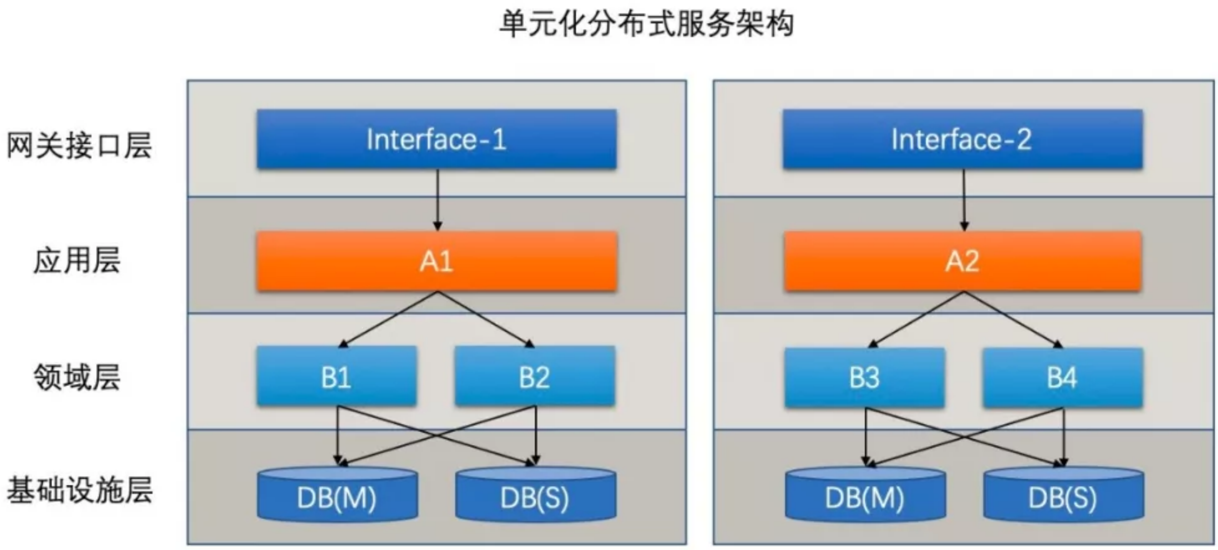
这样一来，从应用服务器　进来的来自 2 客户的所有请求必然落库到 k ^ ŏ2，因此　也不用链接其他的数据库实例了，这样一个单元化的雏形就诞生了。

思考一下，应用间其实也存在交互（比如 2 转账给 ^），也就意味着，应用不需要链接其他的数据库了，但是还需要链接其他应用。

如果是常见的 ĖĒb 框架如 k ő66”等，使用的是 ĶbĒıÆ 协议，那么等同于把之前与数据库建立的链接，换成与其他应用之间的链接了。

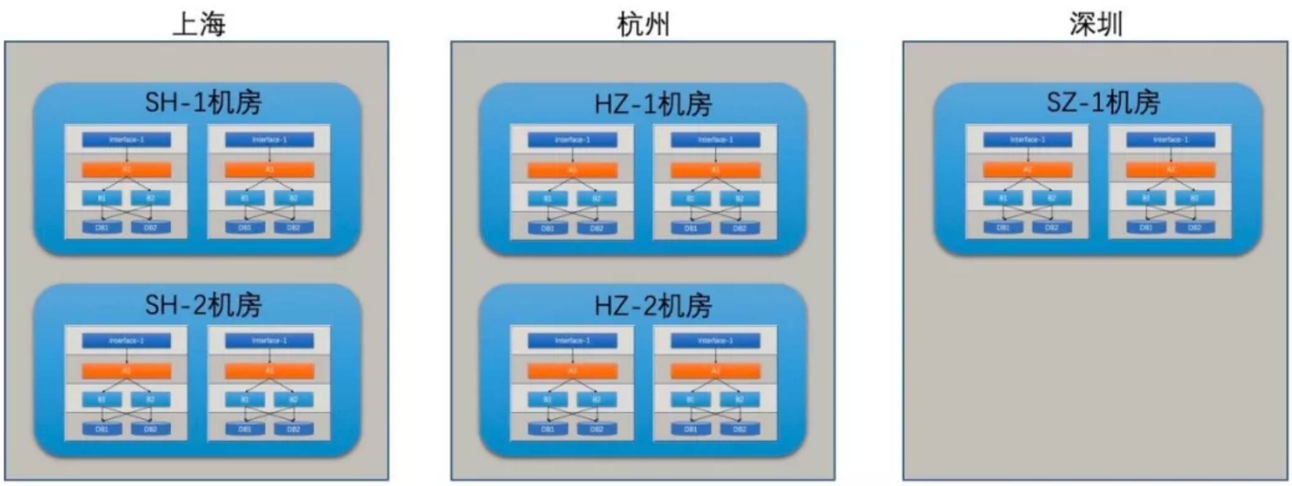
为啥这样就消除瓶颈了呢？首先由于合理的设计，应用间的数据交互并不巨量，其次应用间的交互可以共享 Ķb Ē 链接，比如 2ăï ^ 之间的 ģ” ˥ æXḿ链接可以被 2 中的多个线程复用。

而一般的数据库如 Î ŭģĕ ‡ 则不行，所以 Î ŭģĕ ‡ 才需要数据库链接池。



如上图所示，但我们把整套系统打包为单元化时，每一类的数据从进单元开始就注定在这个单元被消化，由于这种彻底的隔离性，整个单元可以轻松的部署到任意机房而依然能保证逻辑上的统一。

下图为一个三地五机房的部署方式：



蚂蚁单元化架构实践

蚂蚁支付宝应该是国内最大的支付工具，其在双 55 等活动日当日的支付 ĶĒģ 可达几十万级，未来这个数字可能会更大，这决定了蚂蚁单元化架构从容量要求上看必然从单机房走向多机房。

另一方面，异地灾备也决定了这些 ÆÈb 机房必须是异地部署的。整体上支付宝也采用了三地五中心（ÆÈb 机房）来保障系统的可用性。

跟上文中描述的有所不同的是，支付宝将单元分成了三类（也称 bÈõ 架构）：

- **Ė ”ęX（ĖXnjä’ę´ ”ęX）**：直译可能有点反而不好理解。实际上就是所有可以分库分表的业务系统整体部署的最小单元。每个 Ė ”ęX 连上数据库就可以撑起一片天空，把业务跑的溜溜的。
- **ő ”ęX（õË’6 È ”ęX）**：全局单元，意味着全局只有一份。部署了不可拆分的数据和服务，比如系统配置等。
  实际情况下，ő ”ęX 异地也会部署，不过仅是用于灾备，同一时刻，只有一地 ő ”ęX 进行全局服务。ő ”ęX 一般被 Ė ”ęX 依赖，提供的大部分是读取服务。
- **b ”ęX（bāmï ”ęX）**：顾名思义，这是以城市为单位部署的单元。同样部署了不可拆分的数据和服务，比如用户账号服务，客户信息服务等。理论上 b ”ęX 会被 Ė ”ęX 以比访问 ő ”ęX 高很多的频率进行访问。
  b ”ęX 是基于特定的 ő ”ęX 场景进行优化的一种单元，它把 ő ”ęX 中有些有着ή写读时间差现象ή的数据和服务进行了的单独部署，这样 Ė ”ęX 只需要访问本地的 b ”ęX 即可，而不是访问异地的 ő ”ęX。

ή写读时间差现象ή是蚂蚁架构师们根据实践统计总结的，他们发现大部分情况下，一个数据被写入后，都会过足够长的时间后才会被访问。

生活中这种例子很常见，我们办完银行卡后可能很久才会存第一笔钱；我们创建微博账号后，可能想半天才会发微博；我们下载创建淘宝账号后，可能得浏览好几分钟才会下单买东西。

当然了这些例子中的时间差远远超过了系统同步时间。一般来说异地的延时在 ⁵⁰⁰ ȟ ķ 以内，所以只要满足某地 b ”ęX 写入数据后 ⁵⁰⁰ ȟ ķ 以后才用这个数据，这样的数据和服务就适合放到 b ”ęX 中。

相信大家看到这都会问：为啥分这三种单元？其实其背后对应的是不同性质的数据，而服务不过是对数据的操作集。

下面我们来根据数据性质的不同来解释支付宝的 bÈõ 架构。当下几乎所有互联网公司的分库分表规则都是根据用户 ÆÈ 来制定的。

而围绕用户来看整个系统的数据可以分为以下两类：

用户流水型数据：典型的有用户的订单、用户发的评论、用户的行为记录等。

这些数据都是用户行为产生的流水型数据，具备天然的用户隔离性，比如 2 用户的 2ĻĻ 上绝对看不到 ^ 用户的订单列表。所以此类数据非常适合分库分表后独立部署服务。

用户间共享型数据：这种类型的数据又分两类。一类共享型数据是像账号、个人博客等可能会被所有用户请求访问的用户数据。

比如 2 向 ^ 转账，2 给 ^ 发消息，这时候需要确认 ^ 账号是否存在；又比如 2 想看 ^ 的个人博客之类的。

另外一类是用户无关型数据，像商品、系统配置（汇率、优惠政策）、财务统计等这些非用户纬度的数据，很难说跟具体的某一类用户挂钩，可能涉及到所有用户。

比如商品，假设按商品所在地来存放商品数据（这需要双维度分库分表），那么上海的用户仍然需要访问杭州的商品。

这就又构成跨地跨˝ ”ęX 访问了，还是达不到单元化的理想状态，而且双维度分库分表会给整个 ‡k b 运维带来复杂度提升。

注：网上和支付宝内部有另外一些分法，比如流水型和状态性，有时候还会分为三类：流水型、状态型和配置型。

个人觉得这些分法虽然尝试去更高层次的抽象数据分类，但实际上边界很模糊，适得其反。

直观的类比，我们可以很轻易的将上述两类数据对应的服务划分为 Ė ”ęX 和 ő ”ęX，Ė ”ęX 包含的就是分库分表后负责固定客户群体的服务，ő ”ęX 则包含了用户间共享的公共数据对应的服务。

到这里为止，一切都很完美，这也是主流的单元化话题了。对比支付宝的 b Ėõ 架构，我们一眼就发现少了 b（b ãűű˝ ”ęX），b˝ ”ęX 确实是蚂蚁在单元化实践领域的一个创新点。

再来分析下 ő ”ęX，ő ”ęX 之所以只能单地部署，是因为其数据要求被所有用户共享，无法分库分表，而多地部署会带来由异地延时引起的不一致。

比如实时风控系统，如果多地部署，某个 Ė ”ęX 直接读取本地的话，很容易读取到旧的风控状态，这是很危险的。

这时蚂蚁架构师们问了自己一个问题Ä Ä 难道所有数据受不了延时么？这个问题像是打开了新世界的大门，通过对 Ė ”ęX 已有业务的分析，架构师们发现) ⁰ ǎ!甚至更高的场景下，数据更新后都不要求立马被读取到。

也就是上文提到的ŋ写读时间差现象ŋ，那么这就好办了，对于这类数据，我们允许每个地区的 Ė ”ęX 服务直接访问本地，为了给这些 Ė ”ęX 提供这些数据的本地访问能力，蚂蚁架构师设计出了 ŏ ”ęX。

在 ŏ ”ęX 的场景下，写请求一般从 ő ”ęX 写入公共数据所在库，然后同步到整个 ˙ ^ 集群，然后由 ŏ ”ęX 提供读取服务。比如支付宝的会员服务就是如此。

即便架构师们设计了完美的 bĖõ，但即便在蚂蚁的实际应用中，各个系统仍然存在不合理的 bĖõ 分类，尤其是 bõ 不分的现象很常见。

支付宝单元化的异地多活和灾备

流量挑拨技术探秘简介

单元化后，异地多活只是多地部署而已。比如上海的两个单元为 Æ 范围为 ι⁰ ⁰ ȯ⁵ ⁿ ф, ι⁸ ⁰ ȯ⁺ ⁿ ф的用户服务。

而杭州的两个单元为 Æ 为 ι⁶ ⁰ ȯ⁷ ⁿ ф和ι˙ ⁰ ή ⁿ ф的用户服务，这样上海和杭州就是异地双活的。

支付宝对单元化的基本要求是每个单元都具备服务所有用户的能力，即Ä Ä 具体的那个单元服务哪些用户是可以动态配置的。所以异地双活的这些单元还充当了彼此的备份。

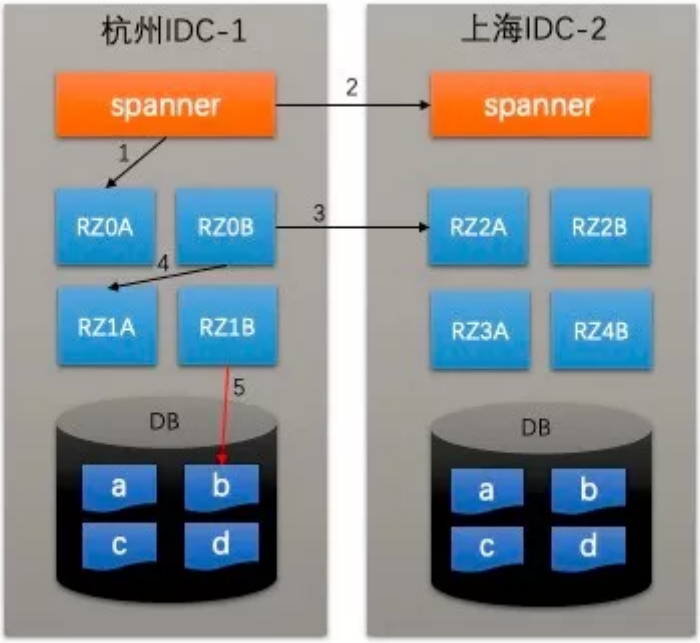
发现工作中冷备热备已经被用的很乱了。最早冷备是指数据库在备份数据时需要关闭后进行备份（也叫离线备份），防止数据备份过程中又修改了，不需要关闭即在运行过程中进行数据备份的方式叫做热备（也叫在线备份）。

也不知道从哪一天开始，冷备在主备系统里代表了这台备用机器是关闭状态的，只有主服务器挂了之后，备服务器才会被启动。

而相同的热备变成了备服务器也是启动的，只是没有流量而已，一旦主服务器挂了之后，流量自动打到备服务器上。本文不打算用第二种理解，因为感觉有点野。

为了做到每个单元访问哪些用户变成可配置，支付宝要求单元化管理系统具备流量到单元的可配置以及单元到 k ^ 的可配置能力。

如下图所示：



其中 gḤ ęęXĮH是蚂蚁基于 ÔnjāęŮ 自研的反向代理网关，也很好理解，有些请求我们希望在反向代理层就被转发至其他 ÆEb 的 gḤ ęęXĮH而无需进入后端服务，如图箭头 [6] 所示。

那么对于应该在本 ÆEb 处理的请求，就直接映射到对应的 Ė 即可，如图箭头 [5]。

进入后端服务后，理论上如果请求只是读取用户流水型数据，那么一般不会再进行路由了。

然而，对于有些场景来说，2 用户的一个请求可能关联了对 ^ 用户数据的访问，比如 2 转账给 ^，2 扣完钱后要调用账务系统去增加 ^ 的余额。

这时候就涉及到再次的路由，同样有两个结果：跳转到其他 ÆEb（如图箭头 [7]）或是跳转到本 ÆEb 的其他 Ė ”ęX（如图箭头 [8]）。

Ė ”ęX 到 k ^ 数据分区的访问这是事先配置好的，上图中 Ė 和 k ^ 数据分区的关系为：

```
RZ0* --> a
RZ1* --> b
```

```
RZ2* --> c
RZ3* --> d
```

下面我们举个例子来说明整个流量挑拨的过程，假设 b 用户所属的数据分区是 ɿ，而 b 用户在杭州访问了 ɿ ķŰăXļĥ ĕăĦ üŋɿ”ȟ（随便编的）。

目前支付宝默认会按照地域来路由流量，具体的实现承载者是自研的 õ‡ģ^（õȄ6 ȄģXĦ XĦ ‡” I ^ Ȅęɿăęnj）：

https://developer.alipay.com/article/1889

它会根据请求者的 ÆĒ，自动将 ɿ ķŰăXļĥ ĕăĦ üŋɿ”ȟ 解析为杭州 ÆEb 的 ÆĒ 地址（或者跳转到 ÆEb 所在的域名）。

大家自己搞过网站的化应该知道大部分 kÔģ 服务商的地址都是靠人去配置的，õ‡ģ^ 属于动态配置域名的系统，网上也有比较火的类似产品，比如花生壳之类（建过私站的同学应该很熟悉）的。

好了，到此为止，用户的请求来到了 ÆEbő 的 ģĦ ęęXĦ集群服务器上，ģĦ ęęXĦ从内存中读取到了路由配置，知道了这个请求的主体用户 b 所属的 Ė ⁷ ŭ不再本 ÆEb，于是直接转到了 ÆEbő 进行处理。

进入 ÆEbő 之后，根据流量配比规则，该请求被分配到了 Ė ⁷ ^ 进行处理。

Ė ⁷ ^ 得到请求后对数据分区 ɿ 进行访问。

处理完毕后原路返回。

大家应该发现问题所在了，如果再来一个这样的请求，岂不是每次都要跨地域进行调用和返回体传递?

确实是存在这样的问题的，对于这种问题，支付宝架构师们决定继续把决策逻辑往用户终端推移。

比如，每个 ÆEb 机房都会有自己的域名（真实情况可能不是这样命名的）ĦĦ

- ÆEbő !对应!ɿ ķŰăXļă ɿőŋ ĕăĦ üŋɿ”ȟ
- ÆEbő 对应!ɿ ķŰăXļă ɿőŋ ĕăĦ üŋɿ”ȟ

那么请求从 Æb 渊过一遍返回时会将前端请求跳转到 ķ 调用的接口域名），后面所有用户的行为都会在这个域名上发生，就避免了走一遍 Æb 带来的延时。

支付宝灾备机制

流量挑拨是灾备切换的基础和前提条件，发生灾难后的通用方法就是把陷入灾难的单元的流量重新打到正常的单元上去，这个流量切换的过程俗称切流。

支付宝 ‡k b 架构下的灾备有三个层次：

- **同机房单元间灾备**
- **同城机房间灾备**
- **异地机房间灾备**

同机房单元间灾备：灾难发生可能性相对最高（但其实也很小）。对 ‡k b 来说，最小的灾难就是某个单元由于一些原因（局部插座断开、线路老化、人为操作失误）宕机了。

从上节里的图中可以看到每组 都有 2，^ 两个单元，这就是用来做同机房灾备的，并且 2^ 之间也是双活双备的。

正常情况下 2^ 两个单元共同分担所有的请求，一旦 2 单元挂了，^ 单元将自动承担 2 单元的流量份额。这个灾备方案是默认的。

同城机房间灾备：灾难发生可能性相对更小。这种灾难发生的原因一般是机房电线网线被挖断，或者机房维护人员操作失误导致的。

在这种情况下，就需要人工的制定流量挑拨（切流）方案了。下面我们举例说明这个过程，如下图所示为上海的两个 Æb 机房。

整个切流配置过程分两步，首先需要将陷入灾难的机房中 Ė ”ęX 对应的数据分区的访问权配置进行修改。

假设我们的方案是由 Æbœ 机房的 Ė ⁶ 和 Ė ⁷ 分别接管 Æbœ 中的 Ė ⁰ 和 Ė ⁵。

那么首先要做的是把数据分区 ，6 对应的访问权从 Ė ⁰ 和 Ė ⁵ 收回，分配给 Ė ⁶ 和 Ė ⁷ 。

即将（如上图所示为初始映射）：

```
RZ0* --> a
RZ1* --> b
RZ2* --> c
RZ3* --> d
```

变为：

```
RZ0* --> /
RZ1* --> /
RZ2* --> a
RZ2* --> c
RZ3* --> b
RZ3* --> d
```

然后再修改用户 Æ 和 Ė 之间的映射配置。假设之前为：

```
[00-24] --> RZ0A(50%),RZOB(50%)
[25-49] --> RZ1A(50%),RZ1B(50%)
```

```
[50-74] --> RZ2A(50%),RZ2B(50%)
[75-99] --> RZ3A(50%),RZ3B(50%)
```

那么按照灾备方案的要求，这个映射配置将变为：

```
[00-24] --> RZ2A(50%),RZ2B(50%)
[25-49] --> RZ3A(50%),RZ3B(50%)
[50-74] --> RZ2A(50%),RZ2B(50%)
[75-99] --> RZ3A(50%),RZ3B(50%)
```

这样之后，所有流量将会被打到 ÆEb ő⁶ 中，期间部分已经向 ÆEb ő⁵ 发起请求的用户会收到失败并重试的提示。

实际情况中，整个过程并不是灾难发生后再去做的，整个切换的流程会以预案配置的形式事先准备好，推送给每个流量挑拨客户端（集成到了所有的服务和 gḤ ęęXḪ中）。

这里可以思考下，为何先切数据库映射，再切流量呢？这是因为如果先切流量，意味着大量注定失败的请求会被打到新的正常单元上去，从而影响系统的稳定性（数据库还没准备好）。

异地机房间灾备：这个基本上跟同城机房间灾备一致（这也是单元化的优点），不再赘述。

蚂蚁单元化架构的 b2Ē 分析

回顾 b2Ē

b2Ē 的定义

b2Ē 原则是指任意一个分布式系统，同时最多只能满足其中的两项，而无法同时满足三项。

所谓的分布式系统，说白了就是一件事一个人做的，现在分给好几个人一起干。

我们先简单回顾下 b2Ē 各个维度的含义：

b”ęķāķmXęl ü（一致性），这个理解起来很简单，就是每时每刻每个节点上的同一份数据都是一致的。

这就要求任何更新都是原子的，即要么全部成功，要么全部失败。想象一下使用分布式事务来保证所有系统的原子性是多么低效的一个操作。

2ţ æË 6ãẽñü（可用性），这个可用性看起来很容易理解，但真正说清楚的不多。我更愿意把可用性解释为：任意时刻系统都可以提供读写服务。

举个例子，当我们用事务将所有节点锁住来进行某种写操作时，如果某个节点发生不可用的情况，会让整个系统不可用。

对于分片式的 Ô" gě ‡ 中间件集群（ĖXI ą̃ķ，Î Xň l l ÜXI ）来说，一旦一个分片歇菜了，整个系统的数据也就不完整了，读取宕机分片的数据就会没响应，也就是不可用了。

需要说明一点，哪些选择 bĒ 的分布式系统，并不是代表可用性就完全没有了，只是可用性没有保障了。

为了增加可用性保障，这类中间件往往都提供了ή分片集群"I复制集ή的方案。

Ē ĻHãñã"ę m̈ ÈXĻ ęl X（分区容忍性），这个可能也是很多文章都没说清楚的。Ē 并不是像 b2 一样是一个独立的性质，它依托于 b2 来进行讨论。

参考文献中的解释：ή除非整个网络瘫痪，否则任何时刻系统都能正常工作ή，言下之意是小范围的网络瘫痪，节点宕机，都不会影响整个系统的 b2。

我感觉这个解释听着还是有点懵逼，所以个人更愿意解释为当节点之间网络不通时（出现网络分区），可用性和一致性仍然能得到保障。

从个人角度理解，分区容忍性又分为ή可用性分区容忍性ή和ή一致性分区容忍性ή。

出现分区时会不会影响可用性的关键在于需不需要所有节点互相沟通协作来完成一次事务，不需要的话是铁定不影响可用性的。

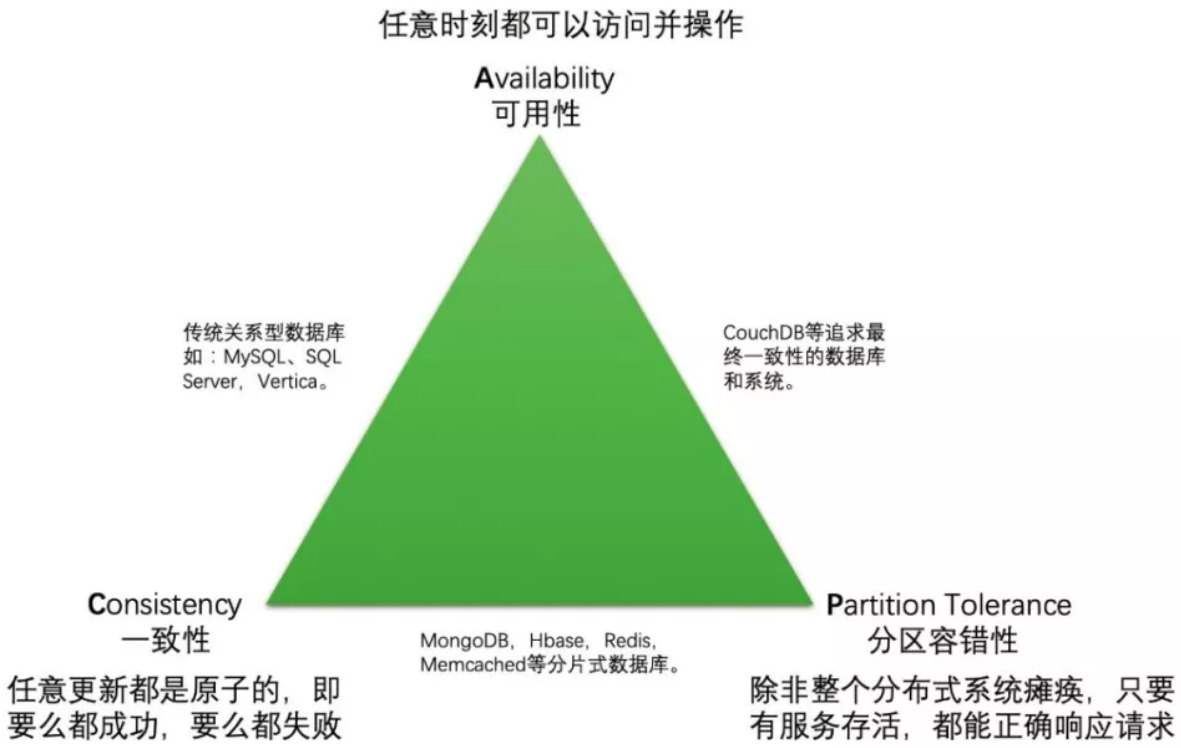
庆幸的是应该不太会有分布式系统会被设计成完成一次事务需要所有节点联动，一定要举个例子的话，全同步复制技术下的 Î ůgě ‡ 是一个典型案例。

出现分区时会不会影响一致性的关键则在于出现脑裂时有没有保证一致性的方案，这对主从同步型数据库（Î ůgě ‡、gě ‡ gXĻ XĻ）是致命的。

一旦网络出现分区，产生脑裂，系统会出现一份数据两个值的状态，谁都不觉得自己是错的。

需要说明的是，正常来说同一局域网内，网络分区的概率非常低，这也是为啥我们最熟悉的数据库（Î ŭğě‡、ğě‡ğXĦXH等）也是不考虑 Ē 的原因。

下图为 b2Ē 之间的经典关系图：



还有个需要说明的地方，其实分布式系统很难满足 b2Ē 的前提条件是这个系统一定是有读有写的，如果只考虑读，那么 b2Ē 很容易都满足。

比如一个计算器服务，接受表达式请求，返回计算结果，搞成水平扩展的分布式，显然这样的系统没有一致性问题，网络分区也不怕，可用性也是很稳的，所以可以满足 b2Ē。

b2Ē 分析方法

先说下 b2 和 Ē 的关系，如果不考虑 Ē 的话，系统是可以轻松实现 b2 的。

而 Ē 并不是一个单独的性质，它代表的是目标分布式系统有没有对网络分区的情况做容错处理。

如果做了处理，就一定是带有 Ē 的，接下来再考虑分区情况下到底选择了 2 还是 b。所以分析 b2Ē，建议先确定有没有对分区情况做容错处理。

以下是个人总结的分析一个分布式系统 b2Ē 满足情况的一般方法：

```
if( 不存在分区的可能性 || 分区后不影响可用性或一致性 || 有影响但考虑了分区情况-P){
    if(可用性分区容忍性-A under P))
      return "AP";
    else if(一致性分区容忍性-C under P)
      return "CP";
}
else{   //分区有影响但没考虑分区情况下的容错
     if(具备可用性-A && 具备一致性-C) {
         return AC;
     }
}
```

这里说明下，如果考虑了分区容忍性，就不需要考虑不分区情况下的可用性和一致性了（大多是满足的）。

水平扩展应用"I单数据库实例的 b2Ē!分析

让我们再来回顾下分布式应用系统的来由，早年每个应用都是单体的，跑在一个服务器上，服务器一挂，服务就不可用了。

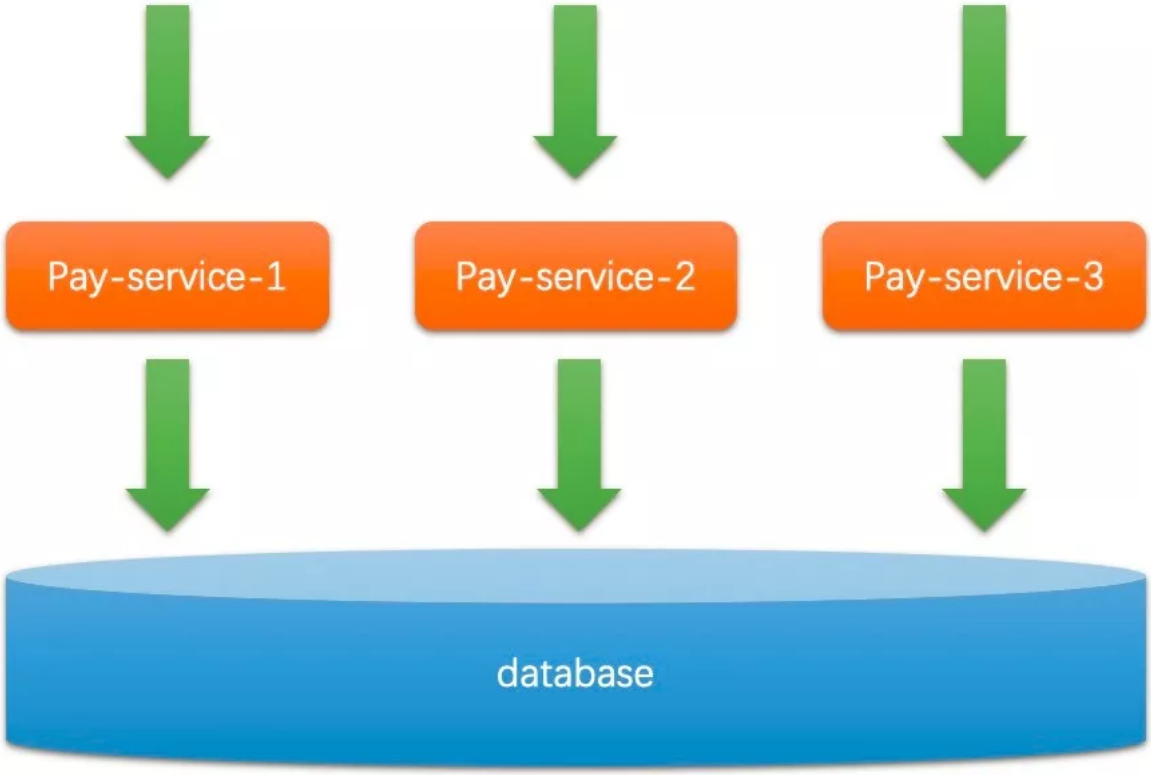
另外一方面，单体应用由于业务功能复杂，对机器的要求也逐渐变高，普通的微机无法满足这种性能和容量的要求。

所以要拆！还在 ÆÎ 大卖小型商用机的年代，阿里巴巴就提出要以分布式微机替代小型机。

所以我们发现，分布式系统解决的最大的痛点，就是单体单机系统的可用性问题。

要想高可用，必须分布式。一家互联网公司的发展之路上，第一次与分布式相遇应该都是在单体应用的水平扩展上。

也就是同一个应用启动了多个实例，连接着相同的数据库（为了简化问题，先不考虑数据库是否单点），如下图所示：



这样的系统天然具有的就是 2Ē（可用性和分区容忍性）：

- 一方面解决了单点导致的低可用性问题。
- 另一方面无论这些水平扩展的机器间网络是否出现分区，这些服务器都可以各自提供服务，因为他们之间不需要进行沟通。

然而，这样的系统是没有一致性可言的，想象一下每个实例都可以往数据库 ãęķXĮĦṅ和 ő Ḥl ṁ（注意这里还没讨论到事务），那还不乱了套。

于是我们转向了让 k ^ 去做这个事，这时候ή数据库事务ή就被用上了。用大部分公司会选择的 Î ügě ‡ 来举例，用了事务之后会发现数据库又变成了单点和瓶颈。

单点就像单机一样н本例子中不考虑从库模式ı，理论上就不叫分布式了，如果一定要分析其 b2Ē 的话，根据上面的步骤分析过程应该是这样的：

- **分区容忍性：**先看有没有考虑分区容忍性，或者分区后是否会有影响。单台 Î ügě ‡ 无法构成分区，要么整个系统挂了，要么就活着。
- **可用性分区容忍性：**分区情况下，假设恰好是该节点挂了，系统也就不可用了，所以可用性分区容忍性不满足。
- **一致性分区容忍性：**分区情况下，只要可用，单点单机的最大好处就是一致性可以得到保障。

因此这样的一个系统，个人认为只是满足了 bĒ。2 有但不出色，从这点可以看出，b2Ē 并不是非黑即白的。

包括常说的 ^2gt （最终一致性）方案，其实只是 b 不出色，但最终也是达到一致性的，^2gt 在一致性上选择了退让。

关于分布式应用"l单点数据库的模式算不算纯正的分布式系统，这个可能每个人看法有点差异，上述只是我个人的一种理解，是不是分布式系统不重要，重要的是分析过程。

其实我们讨论分布式，就是希望系统的可用性是多个系统多活的，一个挂了另外的也能顶上，显然单机单点的系统不具备这样的高可用特性。

所以在我看来，广义的说 b2Ē 也适用于单点单机系统，单机系统是 bĒ 的。

说到这里，大家似乎也发现了，水平扩展的服务应用"l数据库这样的系统的 b2Ē 魔咒主要发生在数据库层。

因为大部分这样的服务应用都只是承担了计算的任务（像计算器那样），本身不需要互相协作，所有写请求带来的数据的一致性问题下沉到了数据库层去解决。

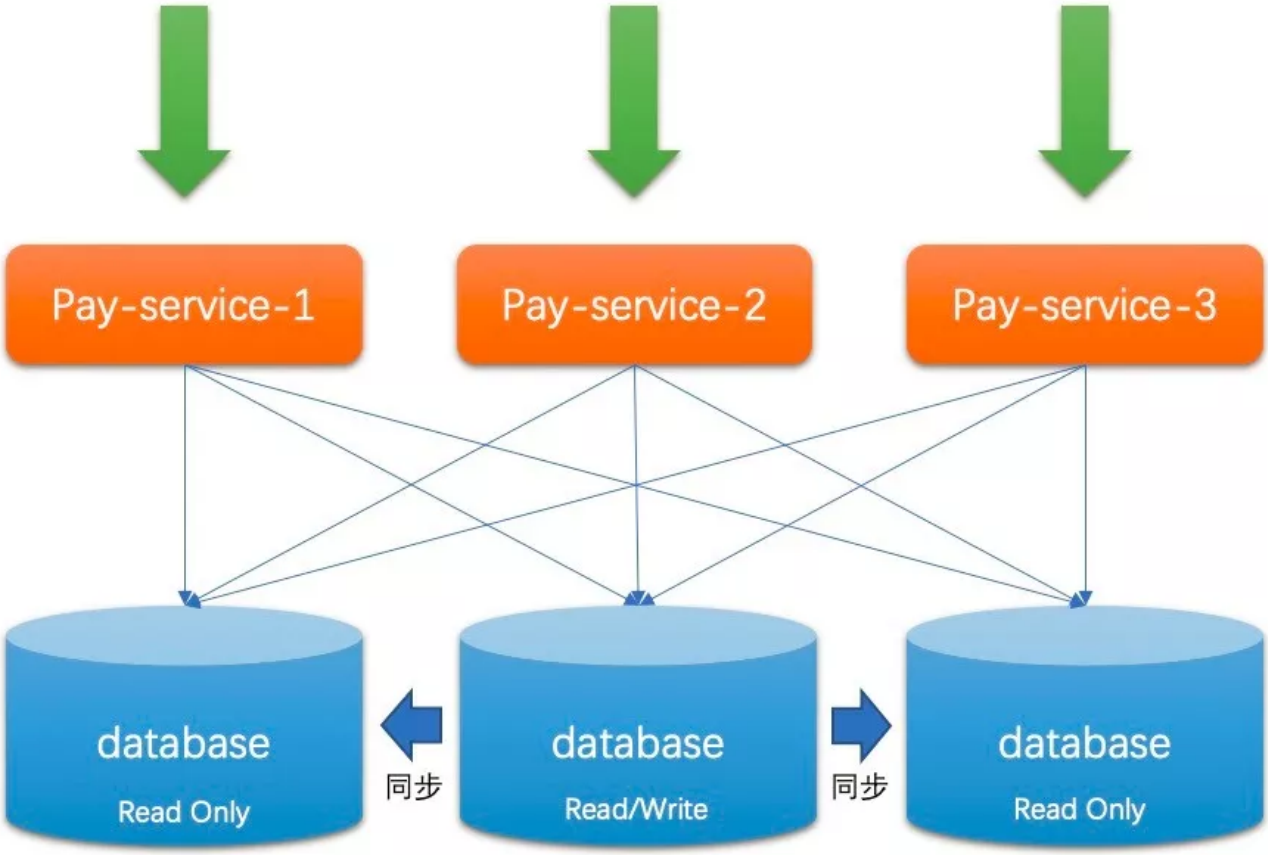
想象一下，如果没有数据库层，而是应用自己来保障数据一致性，那么这样的应用之间就涉及到状态的同步和交互了，˝ ”” ŸXXḤXĮH就是这么一个典型的例子。

水平扩展应用"l主从数据库集群的b2Ē分析

上一节我们讨论了多应用实例"l单数据库实例的模式，这种模式是分布式系统也好，不是分布式系统也罢，整体是偏 bĒ 的。

现实中，技术人员们也会很快发现这种架构的不合理性'A 'A 可用性太低了。

于是如下图所示的模式成为了当下大部分中小公司所使用的架构：



从上图我可以看到三个数据库实例中只有一个是主库，其他是从库。

一定程度上，这种架构极大的缓解了ή读可用性ή问题，而这样的架构一般会做读写分离来达到更高的ή读可用性ή，幸运的是大部分互联网场景中读都占了) 0 ǎ 以上，所以这样的架构能得到较长时间的广泛应用。

写可用性可以通过 ŸXXḤ ƀą XI 这种 §2（高可用）框架来保证主库是活着的，但仔细一想就可以明白，这种方式并没有带来性能上的可用性提升。还好，至少系统不会因为某个实例挂了就都不可用了。

可用性勉强达标了，这时候的 b2Ē 分析如下：

- **分区容忍性：**依旧先看分区容忍性，主从结构的数据库存在节点之间的通信，他们之间需要通过心跳来保证只有一个 Î ķnżĻ。
  然而一旦发生分区，每个分区会自己选取一个新的 Î ķnżĻ 这样就出现了脑裂，常见的主从数据库（Î ŭgě ‡，˙ Ḥ ⌉ƀX 等）并没有自带解决脑裂的方案。所以分区容忍性是没考虑的。
- **一致性：**不考虑分区，由于任意时刻只有一个主库，所以一致性是满足的。
- **可用性：**不考虑分区，§2 机制的存在可以保证可用性，所以可用性显然也是满足的。

所以这样的一个系统，我们认为它是 2b 的。我们再深入研究下，如果发生脑裂产生数据不一致后有一种方式可以仲裁一致性问题，是不是就可以满足 Ē 了呢。

还真有尝试通过预先设置规则来解决这种多主库带来的一致性问题的系统，比如 b”ő⌊Ūk^，它通过版本管理来支持多库写入，在其仲裁阶段会通过 k^2 配置的仲裁规则（也就是合并规则，比如谁的时间戳最晚谁的生效）进行自动仲裁（自动合并），从而保障最终一致性（^2gt），自动规则无法合并的情况则只能依赖人工决策了。

蚂蚁单元化 ‡kb 架构 b2Ē 分析

战胜分区容忍性

在讨论蚂蚁 ‡kb 架构的 b2Ē 之前，我们再来想想分区容忍性有啥值得一提的，为啥很多大名鼎鼎的 ^2gt （最终一致性）体系系统都选择损失实时一致性，而不是丢弃分区容忍性呢?

分区的产生一般有两种情况：

某台机器宕机了，过一会儿又重启了，看起来就像失联了一段时间，像是网络不可达一样。

异地部署情况下，异地多活意味着每一地都可能会产生数据写入，而异地之间偶尔的网络延时尖刺（网络延时曲线图陡增）、网络故障都会导致小范围的网络分区产生。

前文也提到过，如果一个分布式系统是部署在一个局域网内的（一个物理机房内），那么个人认为分区的概率极低，即便有复杂的拓扑，也很少会有在同一个机房里出现网络分区的情况。

而异地这个概率会大大增高，所以蚂蚁的三地五中心必须需要思考这样的问题，分区容忍不能丢！

同样的情况还会发生在不同 Æ₲Ē 的机房之间（想象一下你和朋友组队玩 k˙ Ķ2，他在电信，你在联通）。

为了应对某一时刻某个机房突发的网络延时尖刺活着间歇性失联，一个好的分布式系统一定能处理好这种情况下的一致性问题。

那么蚂蚁是怎么解决这个问题的呢？我们在上文讨论过，其实 ‡kb 机房的各个单元都由两部分组成：负责业务逻辑计算的应用服务器和负责数据持久化的数据库。

大部分应用服务器就像一个个计算器，自身是不对写一致性负责的，这个任务被下沉到了数据库。所以蚂蚁解决分布式一致性问题的关键就在于数据库！

想必蚂蚁的读者大概猜到下面的讨论重点了A A ˙ ɩX ȩ^ ḳX（下文简称˙ ^），中国第一款自主研发的分布式数据库，一时间也确实获得了很多光环。

在讨论˙ ^ 前，我们先来想想 Ŷ Ǖü ȩ" ṁÎ üģĕ ‡?

首先，就像 b2Ē 三角图中指出的，Î üģĕ ‡ 是一款满足 2b 但不满足 Ē 的分布式系统。

试想一下，一个 Î üģĕ ‡ 主从结构的数据库集群，当出现分区时，问题分区内的 ģȆ ṭ X 会认为主已经挂了，所以自己成为本分区的 Î ḳṁẊḶH（脑裂）。

等分区问题恢复后，会产生⁶ 个主库的数据，而无法确定谁是正确的，也就是分区导致了一致性被破坏。这样的结果是严重的，这也是蚂蚁宁愿自研˙ ɩX ȩ^ ḳX 的原动力之一。

那么如何才能让分布式系统具备分区容忍性呢？按照老惯例，我们从ŋ可用性分区容忍ŋ和ŋ一致性分区容忍ŋ两个方面来讨论：

可用性分区容忍性保障机制：可用性分区容忍的关键在于别让一个事务一来所有节点来完成，这个很简单，别要求所有节点共同同时参与某个事务即可。

一致性分区容忍性保障机制：老实说，都产生分区了，哪还可能获得实时一致性。

但要保证最终一致性也不简单，一旦产生分区，如何保证同一时刻只会产生一份提议呢？

换句话说，如何保障仍然只有一个脑呢？下面我们来看下 Ē2ș˙ ģ 算法是如何解决脑裂问题的。

这里可以发散下，所谓的ŋ脑ŋ其实就是具备写能力的系统，ŋ非脑ŋ就是只具备读能力的系统，对应了 Î üģĕ ‡ 集群中的从库。

下面是一段摘自维基百科的 Ē2ș˙ ģ 定义：

> *Ć' śüĞĂĞ' iiŀ Ê ĂŞ üiiiạèüÍŭ üŒiiüèĞüŖĂ vŕ üÎ ĞƩÎ ĞḺĞĂ ' ÎƸÍŗ üèæüiiiĻÎ èƸŒ' ǬạèüƸĞĞüèĞ ʾÍd' Í ĂĢạèüƸĞĞüèĞÍd' Í Ê 'Ş iiŀ ĂǬ*

大致意思就是说，Ē2ș˙ ģ 是在一群不是特别可靠的节点组成的集群中的一种共识机制。

Ē Ǘ' ḳ 要求任何一个提议，至少有 ĤÔı⁶ ɪ"l⁵ 的系统节点认可，才被认为是可信的，这背后的一个基础理论是少数服从多数。

想象一下，如果多数节点认可后，整个系统宕机了，重启后，仍然可以通过一次投票知道哪个值是合法的（多数节点保留的那个值）。

这样的设定也巧妙的解决了分区情况下的共识问题，因为一旦产生分区，势必最多只有一个分区内的节点数量会大于等于 ĤÔı⁶ ɪ˜l⁵。

通过这样的设计就可以巧妙的避开脑裂，当然 Î ŭgĕ ‡ 集群的脑裂问题也是可以通过其他方法来解决的，比如同时 Ēǟęnj 一个公共的 Æ，成功者继续为脑，显然这就又制造了另外一个单点。

如果你了解过比特币或者区块链，你就知道区块链的基础理论也是 Ē2ș˙ g。区块链借助 Ē2ș˙ g 对最终一致性的贡献来抵御恶意篡改。

而本文涉及的分布式应用系统则是通过 Ē2ș˙ g 来解决分区容忍性。再说本质一点，一个是抵御部分节点变坏，一个是防范部分节点失联。

大家一定听说过这样的描述：Ē2ș˙ g 是唯一能解决分布式一致性问题的解法。

这句话越是理解越发觉得诡异，这会让人以为 Ē2ș˙ g 逃离于 b2Ē 约束了，所以个人更愿意理解为：Ē2ș˙ g 是唯一一种保障分布式系统最终一致性的共识算法（所谓共识算法，就是大家都按照这个算法来操作，大家最后的结果一定相同）。

Ē2ș˙ g 并没有逃离 b2Ē 魔咒，毕竟达成共识是 ĤÔı⁶ ɪ˜l⁵ 的节点之间的事，剩下的 ĤÔı⁶ ɪά⁵ 的节点上的数据还是旧的，这时候仍然是不一致的。

所以 Ē2ș˙ g 对一致性的贡献在于经过一次事务后，这个集群里已经有部分节点保有了本次事务正确的结果（共识的结果），这个结果随后会被异步的同步到其他节点上，从而保证最终一致性。

以下摘自维基百科：

> *Ć'śüĜĂĜ' iiÌ Ê ĂĢ üiiiḍèüÍŭ üǬiiüè ĜüǬŖĂ vi üÎ ĜŽÎ ĜḺĜĂ 'ÎŽÍŗ üèæüiiiḺÎ èŽǬ Ǭ ḍèü ŽĜĜüèĜ ʾÍd'Í ĂĢ ḍèü ŽĜĜüèĜÍd'Í Ê'Ṣ iiÌ ĂǬ̥č ḺüèḺÊ ĜŽśḍèŽĜĜÍdŽ Ĝ'iiŽÍṢ ʾüè üÎ ĜĂĜŽÎ Ṣ ḍèüḍŽèÍĂŽĜüiii Ć'śüĜ ṢŽÎ ĜḺèĂ vi 'Í Ǭ'ĜĜüÊŽ ĜḺèŖĂŖĂ viḍèü ŽĜĜüèèŽÍ'Ă Ĝæ̂ üŗ Ǭ viŽ üiiiÍdŽ èŽĜḺǬĜ*

另外 Ē2ș˙ g 不要求对所有节点做实时同步，实质上是考虑到了分区情况下的可用性，通过减少完成一次事务需要的参与者个数，来保障系统的可用性。

˙ ʅX ę^ ķX 的 b2Ē 分析

上文提到过，单元化架构中的成千山万的应用就像是计算器，本身无 b2Ē 限制，其 b2Ē 限制下沉到了其数据库层，也就是蚂蚁自研的分布式数据库 ˙ ʅX ę^ ķX（本节简称 ˙ ^）。

在 ˙ ^ 体系中，每个数据库实例都具备读写能力，具体是读是写可以动态配置（参考第二部分）。

实际情况下大部分时候，对于某一类数据（固定用户号段的数据）任意时刻只有一个单元会负责写入某个节点，其他节点要么是实时库间同步，要么是异步数据同步。

˙ ^ 也采用了 Ē2ș˙ g 共识协议。实时库间同步的节点（包含自己）个数至少需要 ʜÔı⁶ ɪ‟l⁵ 个，这样就可以解决分区容忍性问题。

下面我们举个马老师改英文名的例子来说明 ˙ ^ 设计的精妙之处：

> *假设数据库按照用户 ŭZ 分库分表，马老师的用户 ŭZ 对应的数据段在˘ ˜ ˝ə˙，开始由单元！ 负责数据写入。*
>
> *假如马老师（用户 ŭZ 假设为˜˜˜ ）正在用支付宝！ḍḍ 修改自己的英文名，马老师一开始打错了，打成了 ≈‘ Ğüî ／ ‘，！ 单元收到了这个请求。*
>
> *这时候发生了分区（比如！ 网络断开了），我们将单元！ 对数据段˘ ˃ə˙ 的写入权限转交给单元 M（更改映射），马老师这次写对了，为 ≈ᵛ æ⁄ ‘。*
>
> *而在网络断开前请求已经进入了！，写权限转交给单元 M生效后，！ 和 M同时对˘ ˃ə˙ 数据段进行写入马老师的英文名。*
>
> *假如这时候都允许写入的话就会出现不一致，！ 单元说我看到马老师设置了 ≈‘ Ğüî ／ ‘，M单元说我看到马老师设置了 ≈ᵛ æ⁄ ‘。*
>
> *然而这种情况不会发生的，！ 提议说我建议把马老师的英文名设置为 ≈‘ Ğüî ／ ‘ 时，发现没人回应它。*
>
> *因为出现了分区，其他节点对它来说都是不可达的，所以这个提议被自动丢弃，！ 心里也明白是自己分区了，会有主分区替自己完成写入任务的。*
>
> *同样的，M提出了将马老师的英文名改成 ≈ᵛ æ⁄ ‘ 后，大部分节点都响应了，所以 M成功将 ≈ᵛ æ ／ ‘ 写入了马老师的账号记录。*
>
> *假如在写权限转交给单元 M后！ 突然恢复了，也没关系，两笔写请求同时要求获得 ˀfi¸Ə‘__ 个节点的事务锁，通过 Î ù˝ī ‘Ǻ 设计，在 M获得了锁之后，其他争抢该锁的事务都会因为失败而回滚。*

下面我们分析下 ˙ ^ 的 b2Ē：

- **分区容忍性：** ˙ ^ 节点之间是有互相通信的（需要相互同步数据），所以存在分区问题，˙ ^ 通过仅同步到部分节点来保证可用性。这一点就说明 ˙ ^ 做了分区容错。
- **可用性分区容忍性：** ˙ ^ 事务只需要同步到!（Ôı⁶ ɪ‟l⁵ 个节点，允许其余的一小半节点分区（宕机、断网等），只要 ʜÔı⁶ ɪ‟l⁵ 个节点活着就是可用的。

极端情况下，比如 + 个节点分成 7 份（6 'Ħ'Ħ），那就确实不可用了，只是这种情况概率比较低。

- **一致性分区容忍性：** 分区情况下意味着部分节点失联了，一致性显然是不满足的。但通过共识算法可以保证当下只有一个值是合法的，并且最终会通过节点间的同步达到最终一致性。

所以 ˙ ^ 仍然没有逃脱 b2Ē 魔咒，产生分区的时候它变成 2Ē˝l最终一致性（b）。整体来说，它是 2Ē 的，即高可用和分区容忍。

结语

个人感觉本文涉及到的知识面确实不少，每个点单独展开都可以讨论半天。回到我们紧扣的主旨来看，双十一海量支付背后技术上大快人心的设计到底是啥?

我想无非是以下几点：

- 基于用户分库分表的 Ė ”ęX 设计。每个用户群独占一个单元给整个系统的容量带来了爆发式增长。
- Ė ”ęX 在网络分区或灾备切换时 ˙ ^ 的防脑裂设计（Ē2ş˙ ģ）。我们知道 Ė ”ęX 是单脑的（读写都在一个单元对应的库），而网络分区或者灾备时热切换过程中可能会产生多个脑，˙ ^ 解决了脑裂情况下的共识问题（Ē2ş˙ ģ 算法）。
- 基于 b́ ”ęX 的本地读设计。这一点保证了很大一部分有着ŋ写读时间差ŋ现象的公共数据能被高速本地访问。
- 剩下的那一丢丢不能本地访问只能实时访问 ő ”ęX 的公共配置数据，也兴不起什么风，作不了什么浪。
  比如用户创建这种 ĶĒģ，不会高到哪里去。再比如对于实时库存数据，可以通过ŋ页面展示查询走应用层缓存ŋ˝lŋ实际下单时再校验ŋ的方式减少其 ő ”ęX 调用量。

而这就是蚂蚁 ‡k b 的 bĖő 架构，相信 + 8 ŋ̊ 万笔ı秒还远没到 ‡k b 的上限，这个数字可以做到更高。

当然双 5 5 海量支付的成功不单单是这么一套设计所决定的，还有预热削峰等运营˝l技术的手段，以及成百上千的兄弟姐妹共同奋战，特此在这向各位双 5 5 留守同学致敬。

感谢大家的阅读，文中可能存在不足或遗漏之处，欢迎批评指正。

参考文献：

- *Ćè˅ ĺĂ ʓ üiiiQǬĻ ĎȘĞ̋ʓÊ !ˆ Ê Ă̄ Ă̋Ğè' ĺĂ̄ıĬ ̧ Ģdʓ̧ ZʓŖĖȩdĜ ' Ĭ̂ ĎđcĆè˅ ĺĂʓĜiiüè t' ʓ ĎʓèŖĂ ʓĢ ŢüǬĒʓȫ ĢdüÊ ' Ğ ! ̧ œĔ ü Ĭ̂ ʓǪ̋Ă Ďĺè' ĺ' đ ̧ Qd ' Ǭ̧d ̧ QdèĂ̋Ğ̋Ă ' ≈ ̧ ì üvi' Ĭ 。*
- */ ȘĎč œ´ ̧Z'Ұ̈半同步复制技术*

dĬĭďĞ¸¸ŗ ŗ ŗ ̥̌ Ĭ̂ ǬviĢ ùÊ ¸ųʒǜ⁼vivi¸ď¸ә̆ ′ Ƶ ɵӛ̥ďĬÊ Ǭ

- *M! Ďc 理论分析*

  dĬĭďĞ¸¸ŗ ŗ ŗ ¸ǞÎ ĞdĻ̥̌ ùÊ ¸ď¸iii ′ Ƶ zʒ′ ɵ̈

- *»ʒʒď ' ǬR̈ʒ*

  dĬĭďĞ¸¸ ' Ăəʒ ' Ă Ļ̥̌ ùÊ ¸ĂʒÊ ¸»ʒʒď ' ǬR̈ʒ ö Ә́ Ƶ′ z ƀiiè ' Ǭ ˆ Ă

- *Ć! ŬÈĎ*

  dĬĭďĞ¸¸ʒÎ ¸ŗ ĂəĂŗʒ Ă ̥ùèvi¸ŗ ĂəĂĆ'śùĞ ̌ ùÊ ďĻĭʒè Ğ ĂʒĬ̌ ʒ́

- *È ʒ'Î M'Ğʒ 支撑 Ә ö′ 亿成交额背后的技术原理*

  dĬĭďĞ¸¸ŗ ŗ ŗ ̥̌ Ĭ̂ ǬviĢ ùÊ ¸'Î ĬiiĂ ¸'èĬĂ ǬĞ Әɵɵəɵ ̥ďĬÊ Ǭ

- *M̌ əĻď*

  dĬĭďĞ¸¸ʒÎ ¸ŗ ĂəĂŗʒ Ă ̥ùèvi¸ŗ ĂəĂM̌ əĻď

如果觉得文章对你有帮助，**请转发朋友圈、点在看**，让更多人获益，感谢您的支持！

t Ôk

关注我

公众号 **WŬăķŬXęnj** 里回复 面经、t g、ôĽęæ ！gHĻęnj、÷ ţ 、Ÿ Nəə、监控 等关键字可以查看更多关键字对应的文章。

Flink 实战

1、《从0到1学习Flink》—— Apache Flink 介绍

2、《从0到1学习Flink》—— Mac 上搭建 Flink 1.6.0 环境并构建运行简单程序入门

3、《从0到1学习Flink》—— Flink 配置文件详解

4、《从0到1学习Flink》—— Data Source 介绍

5、《从0到1学习Flink》—— 如何自定义 Data Source ？

6、《从0到1学习Flink》—— Data Sink 介绍

7、《从0到1学习Flink》—— 如何自定义 Data Sink ？

8、《从0到1学习Flink》—— Flink Data transformation(转换)

9、《从0到1学习Flink》—— 介绍 Flink 中的 Stream Windows

10、《从0到1学习Flink》—— Flink 中的几种 Time 详解

11、《从0到1学习Flink》—— Flink 读取 Kafka 数据写入到 ElasticSearch

12、《从0到1学习Flink》—— Flink 项目如何运行？

13、《从0到1学习Flink》—— Flink 读取 Kafka 数据写入到 Kafka

14、《从0到1学习Flink》—— Flink JobManager 高可用性配置

15、《从0到1学习Flink》—— Flink parallelism 和 Slot 介绍

16、《从0到1学习Flink》—— Flink 读取 Kafka 数据批量写入到 MySQL

17、《从0到1学习Flink》—— Flink 读取 Kafka 数据写入到 RabbitMQ

18、《从0到1学习Flink》—— 你上传的 jar 包藏到哪里去了

19、大数据“重磅炸弹”——实时计算框架 Flink

20、《Flink 源码解析》—— 源码编译运行

21、为什么说流处理即未来？

22、OPPO数据中台之基石：基于Flink SQL构建实时数据仓库

23、流计算框架 Flink 与 Storm 的性能对比

24、Flink状态管理和容错机制介绍

25、原理解析 | Apache Flink 结合 Kafka 构建端到端的 Exactly-Once 处理

26、Apache Flink 是如何管理好内存的？

27、《从0到1学习Flink》——Flink 中这样管理配置，你知道？

28、《从0到1学习Flink》——Flink 不可以连续 Split(分流)？

29、Flink 从0到1学习—— 分享四本 Flink 的书和二十多篇 Paper 论文

30、360深度实践：Flink与Storm协议级对比

31、Apache Flink 1.9 重大特性提前解读

32、如何基于Flink+TensorFlow打造实时智能异常检测平台？只看这一篇就够了

33、美团点评基于 Flink 的实时数仓建设实践

34、Flink 灵魂两百问，这谁顶得住？

35、一文搞懂 Flink 的 Exactly Once 和 At Least Once

36、你公司到底需不需要引入实时计算引擎？

37、Flink 从0到1学习 —— 如何使用 Side Output 来分流？

38、一文让你彻底了解大数据实时计算引擎 Flink

39、基于 Flink 实现的商品实时推荐系统(附源码)

40、如何使用 Flink 每天实时处理百亿条日志？

41、Flink 在趣头条的应用与实践

42、Flink Connector 深度解析

43、滴滴实时计算发展之路及平台架构实践

44、Flink Back Pressure(背压)是怎么实现的？有什么绝妙之处？

45、Flink 实战 | 贝壳找房基于Flink的实时平台建设

46、如何使用 Kubernetes 部署 Flink 应用

47、一文彻底搞懂 Flink 网络流控与反压机制

48、Flink中资源管理机制解读与展望

49、Flink 实时写入数据到 ElasticSearch 性能调优

50、深入理解 Flink 容错机制

51、吐血之作 | 流系统Spark/Flink/Kafka/DataFlow端到端一致性实现对比

ôbąę源码解析

知识星球里面可以看到下面文章

阅读原文