背景：为什么要做异地多活?

饿了么要做多活，是受业务发展的驱动，经过几年的高速发展，我们的业务已经扩大到单个数据中心撑不住了，主要机房已经不能再加机器，业务却不断的要求加扩容，所以我们需要一个方案能够把服务器部署到多个机房。

另外一个更重要的原因是，整个**机房级别的故障**时有发生，每次都会带来严重的后果，我们需要在发生故障时，能够把一个机房的业务全部迁移到别的机房，保证服务可用。

异地多活面临的主要挑战是网络延迟，**以北京到上海 1468 公里，即使是光速传输，一个来回也需要接近10ms，我们在实际测试的过程中，发现上海到北京的网络延迟，一般是 30 ms。**

这 30 ms可以和运算系统中其他的延迟时间做个比较：

```
L1 cache reference ......................... 0.5 ns

Branch mispredict ............................ 5 ns

L2 cache reference ........................... 7 ns

Mutex lock/unlock ........................... 25 ns

Main memory reference ...................... 100 ns

Compress 1K bytes with Zippy ............. 3,000 ns = 3 µs

Send 2K bytes over 1 Gbps network ....... 20,000 ns = 20 µs

SSD random read ........................ 150,000 ns = 150 µs

Read 1 MB sequentially from memory ..... 250,000 ns = 250 µs

Round trip within same datacenter ...... 500,000 ns = 0.5 ms

Read 1 MB sequentially from SSD* ..... 1,000,000 ns = 1 ms
```

北京上海两地的网络延迟时间，大致是内网网络访问速度的 **60 倍（30ms/0.5ms）**.

如果不做任何改造，一方直接访问另外一方的服务，那么我们的APP的反应会比原来**慢 60 倍**，其实考虑上多次往返，可能会**慢600倍。**

如果机房都在上海，那么网络延迟只有内网速度的2倍，**可以当成一个机房使用。**

所有有些公司的多活方案，会选择同城机房，把同城的几个机房当成一个机房部署，可以在不影响服务架构的情况下扩展出多个机房，不失为一个快速见效的方法。

我们在做多活的初期也讨论过**同城方案**，比如在北京周边建设一个新机房，迁移部分服务到新机房，**两个机房专线连接**，服务间做跨机房调用。

虽然这个方案比较容易，也解决了机房的扩展问题，**但是对高可用却没有好处**，相反还带来了更高的风险。

异地多活的关键

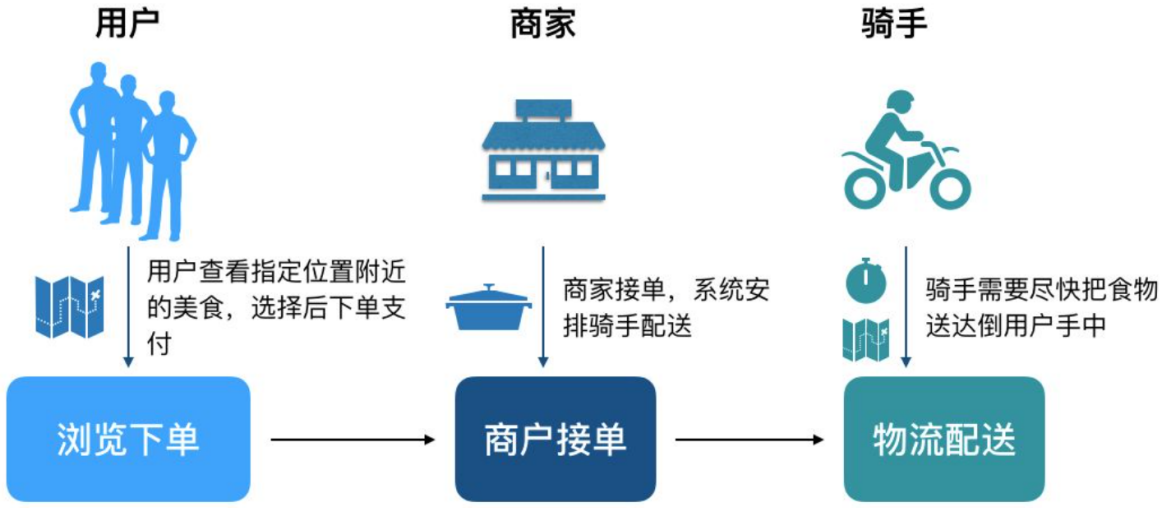
与同城多活的方案不同，异地多活的关键—— **限制机房间的相互调用**，需要对业务 进行单元化改造 —— 定义**清晰的服务边界**，**减少相互依赖**，**让每个机房都成为独立的单元**，不依赖于其他机房。

经过几番考量，我们最终选择了异地多活的方案，对这两个方案的比较和思考可以见下表，异地多活虽然更困难一点，但是能同时达到我们的两个核心目标，更为可行。

设计：异地多活的实现思路和方法

我们的异地多活方案的，有几条基本原则，整个多活方案都是这些原则的自然推导。但在介绍一下这些原则之前，先要说明一下饿了么的服务流程，才能让大家更好的理解这些原则的来由

下面这张简图是我们的主流程：



业务过程中包含3个最重要的角色，分别是用户、商家和骑手，一个订单包含3个步骤：

1. 用户打开我们的APP，系统会推荐出用户位置附近的各种美食，推荐顺序中结合了用户习惯，推荐排序，商户的推广等。用户找到中意的食物 ，下单并支付，订单会流转到商家。
2. 商家接单并开始制作食物，制作完成后，系统调度骑手赶到店面，取走食物
3. 骑手按照配送地址，把食物送到客户手中。

整个下单到配送完成，有严格的时间要求，必须在短短的几十分钟内完成，我们的服务和地理位置强相关，并且实时性要求高，服务的地域性和实时性是我们的核心特性，多活设计最重要的是满足这两个特性。

进过反复讨论，我们的多活架构通过遵循以下几条基本原则，来满足这两个核心特性：

业务内聚：

单个订单的旅单过程，要在一个机房中完成，**不允许跨机房调用。**这个原则是为了保证实时性，旅单过程中不依赖另外一个机房的服务，才能保证没有延迟。我们称**每个机房为一个 ezone**，一个 ezone 包含了饿了么需要的各种服务。一笔业务能够内聚在一个 ezone 中，那么一个定单涉及的**用户，商家，骑手**，**都会在相同的机房**，这样订单在各个角色之间流转速度最快，不会因为各种异常情况导致延时。恰好我们的业务是地域化的，通过合理的地域划分，也能够实现业务内聚。

> eureka提供了region和zone两个概念来进行分区，这两个概念均来自于亚马逊的AWS：
>
> - region：可以简单理解为地理上的分区，比如亚洲地区，或者华北地区，再或者北京等等，没有具体大小的限制。根据项目具体的情况，可以自行合理划分region。
> - zone：可以简单理解为region内的具体机房，比如说region划分为北京，然后北京有两个机房，就可以在此region之下划分出zone1,zone2两个zone。

可用性优先：

当发生故障切换机房时，优先保证系统可用，首先让用户可以下单吃饭，容忍有限时间段内的数据不一致，在事后修复。**每个 ezone 都会有全量的业务数据**，当一个 ezone 失效后，其他的 ezone 可以接管用户。用户在一个ezone的下单数据，会实时的复制到其他ezone。

保证数据正确：

在确保可用的情况下，需要对数据做保护以避免错误，在切换和故障时，如果发现某些订单的状态在两个机房不一致，会锁定该笔订单，阻止对它进行更改，保证数据的正确。

> 简单的说，当数据不一致的时候，进行锁定。 通过锁定不一致数据的方式，保障数据一致性。

业务可感多活区域：

因为基础设施还没有强大到可以抹去跨机房的差异，**需要让业务感知多活逻辑**，业务代码要做一些改造，包括：需要业务代码能够**识别出业务数据的归属**，只处理本 ezone 的数据，过滤掉无关的数据。

完善业务状态机，能够在数据出现不一致的时候，通过状态机发现和纠正。

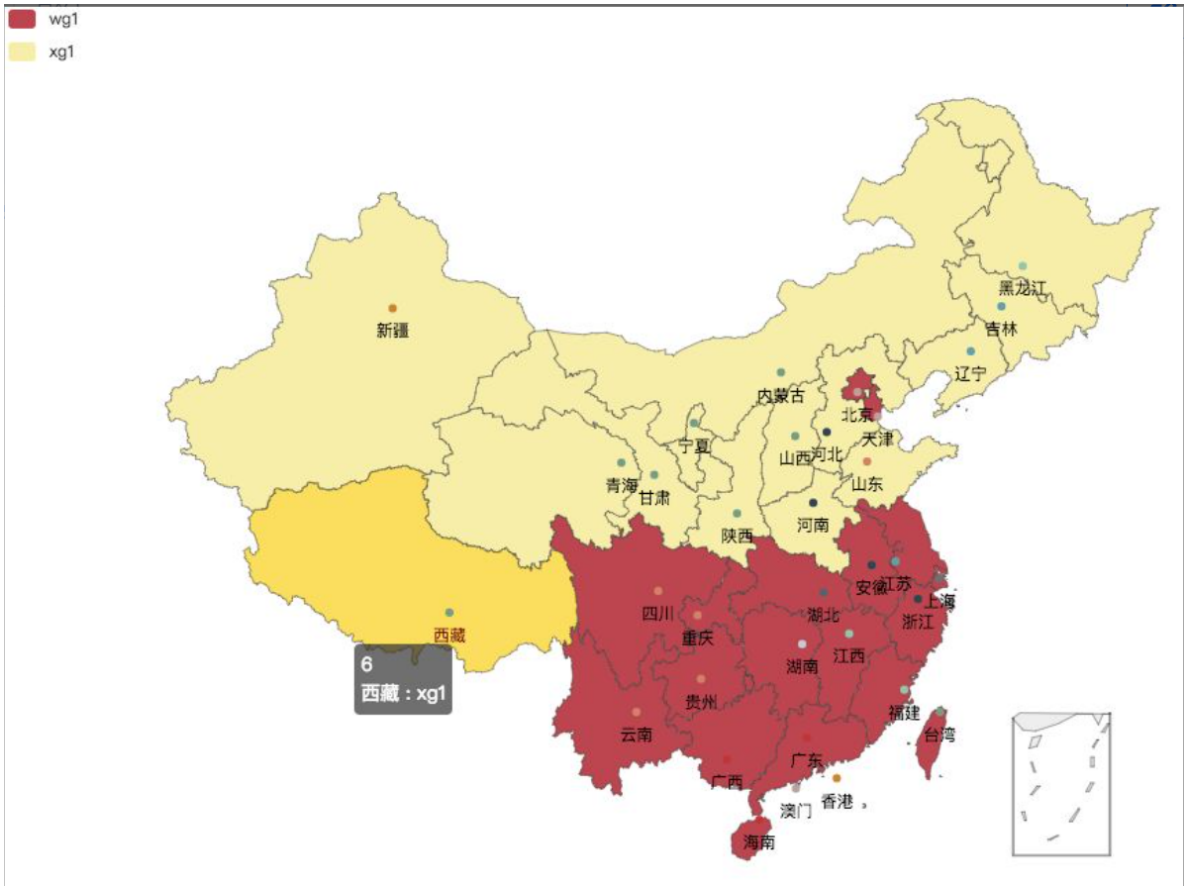
分区方案（Sharding）

为了实现业务内聚，我们首先要选择一个划分方法（Sharding Key），对服务进行分区，让用户，商户，骑手能够正确的内聚到同一个 ezone 中。

> 分区方案是整个多活的基础，它决定了之后的所有逻辑。

根据饿了么的业务特点，我们自然的**选择地理位置作为划分业务的单元**，把地理位置上接近的用户，商户，骑手划分到同一个ezone，这样**一个订单的履单流程就会在一个机房完成**，能够保证最小的延时，在某个机房出现问题的时候，也可以按照地理位置把用户，商户，骑手打包迁移到别的机房即可。

所以我们最终选择的方案如下图，自定义地理划分围栏，用围栏把全国分为多个 shard，围栏的边界尽量按照行政省界，必要的时候做一些调整，避免围栏穿过市区。一个ezone可以包含多个 shard，某个 ezone 的 shard ，可以随时切换到另外一个 ezone ，灵活的调度资源和failover。



这样的划分方案，基本解决了垮城市下单的问题，**线上没有观察到有跨 ezone 下单的情况。**围栏的划分是灵活的，可以随着以后业务的拓展进行修改，因为每个机房都是全量数据，所以调整围栏不会导致问题。

对这种划分方法，有一些常见的疑问，比如：

1.**如果两个城市是接壤的**，会出现商家和用户处于不同 ezone 的情况，岂不是破坏了内聚性原则？

这种情况确实会出现，为了尽量避免，我们在划分shard的时候没有简单的用城市名称，而是用了复杂的地理围栏实现，地理围栏主体按照省界划分，再加上局部微调，我们最大限度的避免了跨ezone下单的情况。但如果真的出现了，用户下单也不受影响，最多只是状态有1s左右的延迟。

2.用户是会动的，如果用户从北京到了上海，那么划分规则应该怎么应对?

用户在北京下单，数据落在北京shard，到上海下单，数据则落在上海的 shard，借助于**底层的数据同步工具**，用户无论在什么地方，都能看到自己的数据，但是**有1s左右的延时**，对于大部分的业务场景，这个延迟是可以承受的。当然也有些业务场景不能接受这 1s 的延时，我们也提供了另外的方案来应对，参考下文介绍Globa Zone的章节。

3.为什么不简单点，按照用户的ID来切分?

阿里是按照用户ID**的取模来划分单元的**，比较简洁。

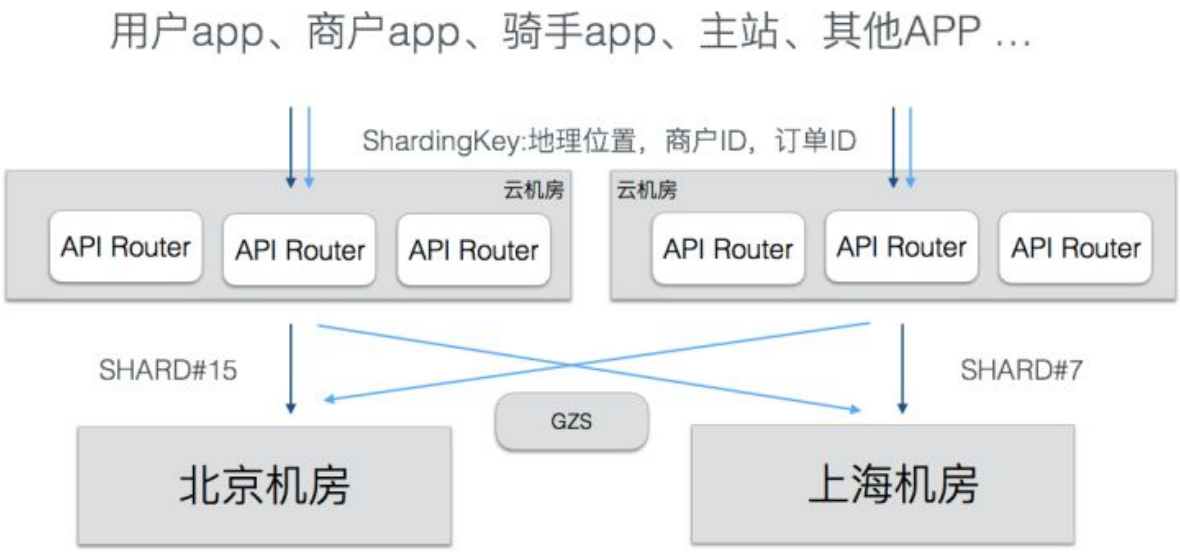
我们如果也用ID做切分，同一地方的用户，商户，骑手可能被划分到不同 ezone，就会出现比较多的跨机房调用，这样就更可能出现延迟，难以保证实时性。所以，我们本地配送的业务模式，决定了需要用地理位置来划分服务。

流量路由:

基于地理位置**划分规则**，我们开发了统一的流量路由层（API Router）。

流量路由层（API Router）的职责:

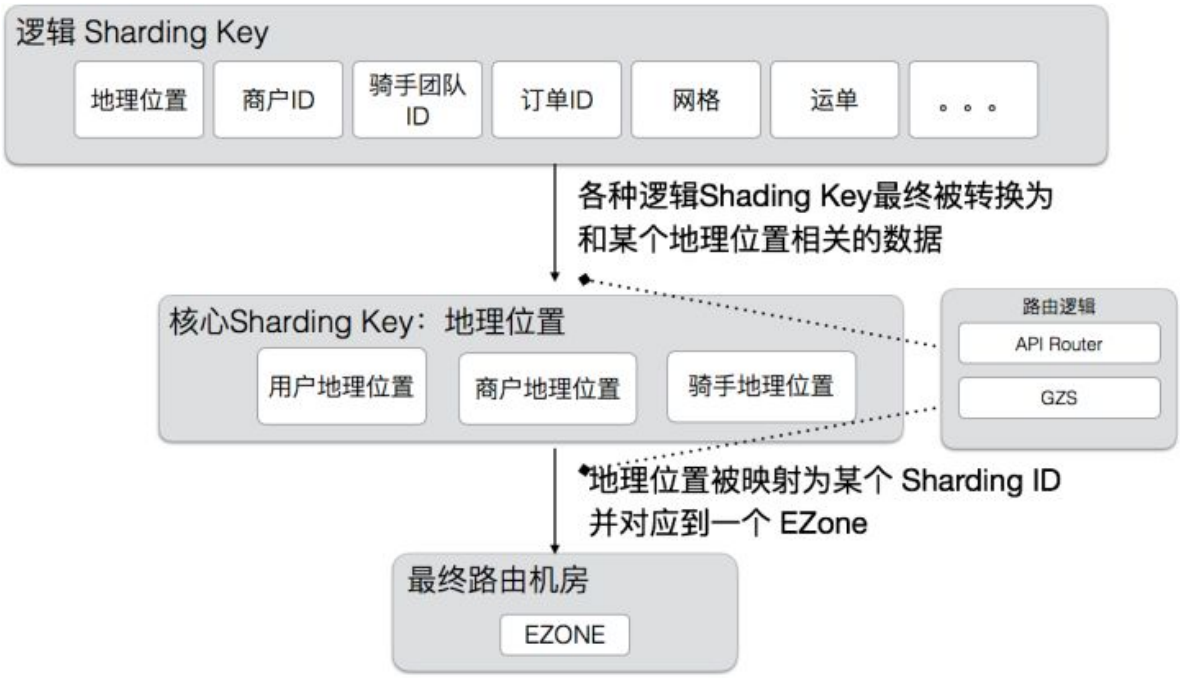
> 这一层负责对客户端过来的 API 调用进行路由，把流量导向到正确的 ezone。
>
> API Router 部署在多个公有云机房中，用户就近接入到公有云的API Router，还可以提升接入质量。



前端 APP 做了改造，为每个请求都带上了分流标签，API Router 会检查流量上自带的分流标签，把分流标签转换为对应的 Shard ID，再查询 Shard ID 对应的 eZone，最终决定把流量路由到哪个 ezone。

最基础的分流标签是地理位置，有了地理位置，AR 就能计算出正确的 shard 归属。

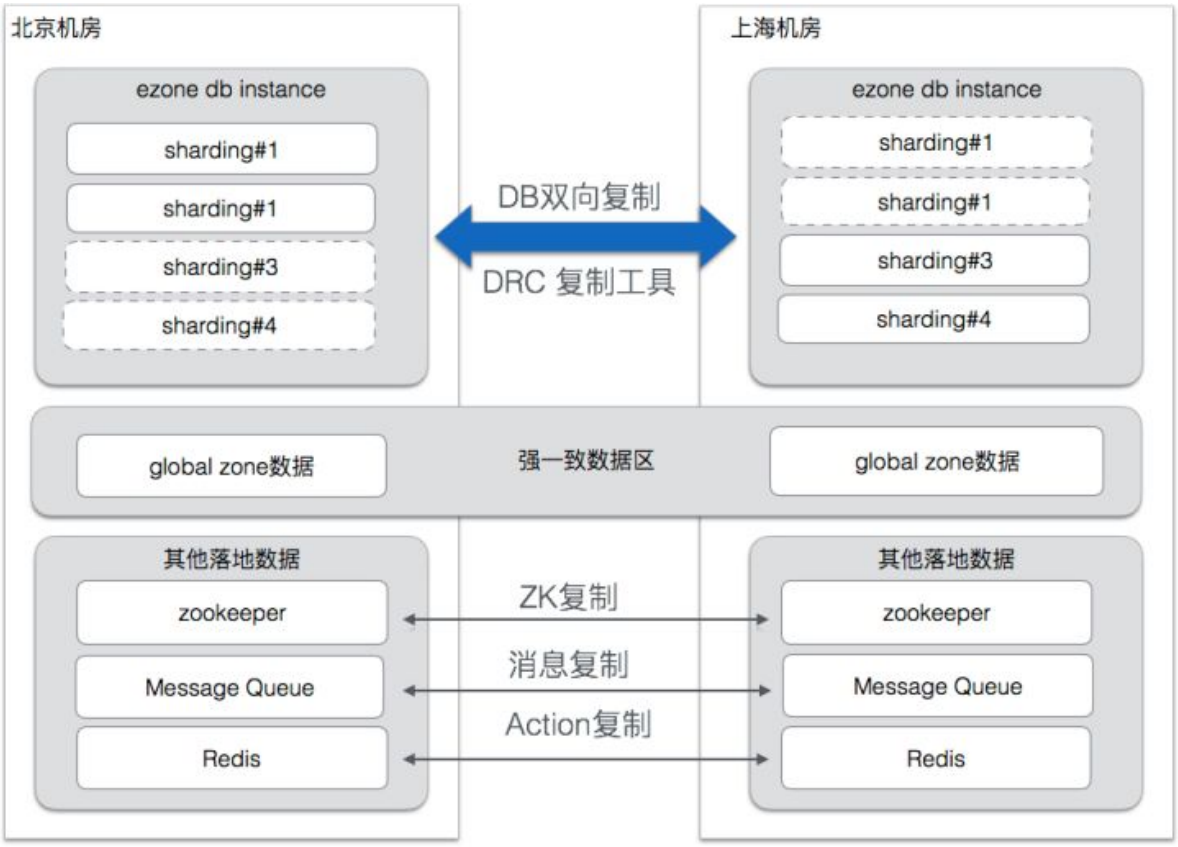
但业务是很复杂的，并不是所有的调用都能直接关联到某个地理位置上，我们使用了一种分层的路由方案，核心的路由逻辑是地理位置，但是也支持其他的一些 High Level Sharding Key，这些 Sharding Key 由 APIRouter 转换为核心的 Sharding Key，具体如下图。这样既减少了业务的改造工作量，也可以扩展出更多的分区方法。



除了入口处的路由，我们还开发了 SOA Proxy，用于路由SOA调用的，和API Router基于相同的路由规则。APIRouter 和 SOAProxy 构成了流量的路由通道，让我们可以灵活的控制各种调用在多活环境下的走向。

数据复制：

为了实现可用优先原则，所有机房都会有全量数据，这样用户可以随时切换到其他机房，全量数据就需要对数据进行实时复制，我们开发了相应的中间件，对 mysql，zookeeper ，消息队列和 redis 的数据进行复制。



实现：多活的基础中间件

下面简要介绍一下支持以上功能的中间件，我们归纳为多活 5 大基础组件，之后会有系列文章，介绍每个基础组件的具体实现。

APIRouter： 路由分发服务

API Router是一个HTTP反向代理和负载均衡器，部署在公有云中作为HTTP API流量的入口，它能识别出流量的归属 shard ，并根据 shard 将流量转发到对应的 ezone。API Router 支持多种路由键，可以是地理位置，也可以是商户ID，订单ID等等，最终由 API Router 映射为统一的 Sharding ID。

Global Zone Service：全局状态协调器

GZS 维护着整个多活的路由表，其他所有的服务都从 GZS 订阅路由信息。切换机房的操作也在 GZS 控制台中完成。路由表包括：地理围栏信息，shard 到 ezone 的归属信息，商铺ID / 订单ID 等路由逻辑层到 shard id 的映射关系等。GZS 通过在 SDK 端建立 Cache，来保证shard 逻辑能够最快速度执行，基本不需要和 GZS 交互，同时也有实时推送机制，确保在数据变更后能够快速通知到其他的服务。

SOA Proxy：内部网关

SOA Proxy 实现了对 SOA 调用的路由，执行和 API Router 相似的逻辑，但只用在机房之间进行通信的场景。业务使用 SOA Proxy 需要对代码做一些修改，把路由信息加入到调用的上下文中。

Data Replication Center：数据复制

DRC 负责 Mysql 数据的实时双向复制，保证跨机房延时在 1s 以内。提供了基于时间的冲突解决方案，确保各个机房的数据一致。DRC 除了复制数据，还对外提供了数据变更的通知，让业务能够感知到其他机房的数据变化，做相应的处理，例如清除Cache等。

除了DRC，我们还有 ZK复制工具，RMQ 复制工具，Redis复制工具，基本每个数据层次，都有对应的复制方案。

Data Access Layer：数据访问

数据访问层支撑了 Globa Zone 的逻辑，还提供了最后一道保护，拒绝路由错误的数据写入，是多活最底层的支撑。

Mysql 数据复制工具 DRC:

Mysql 的数据量最大，每个机房产生的数据，都通过 DRC 复制到其他 ezone，每个ezone的主键取值空间是ezoneid + 固定步长，所以产生的 id 各不相同，数据复制到一起后不会发生主键冲突。按照分区规则，正常情况下，每个 ezone 只会写入自己的数据，但万一出现异常，2个 ezone 同时更新了同一笔数据，就会产生冲突。DRC 支持基于时间戳的冲突解决方案，当一笔数据在两个机房同时被修改时，最后修改的数据会被保留，老的数据会被覆盖。

ZooKeeper 复制:

有些全局的配置信息，需要在所有机房都完全一致，我们开发了 zookeeper 复制工具，用于在多个机房中同步 ZK 信息。

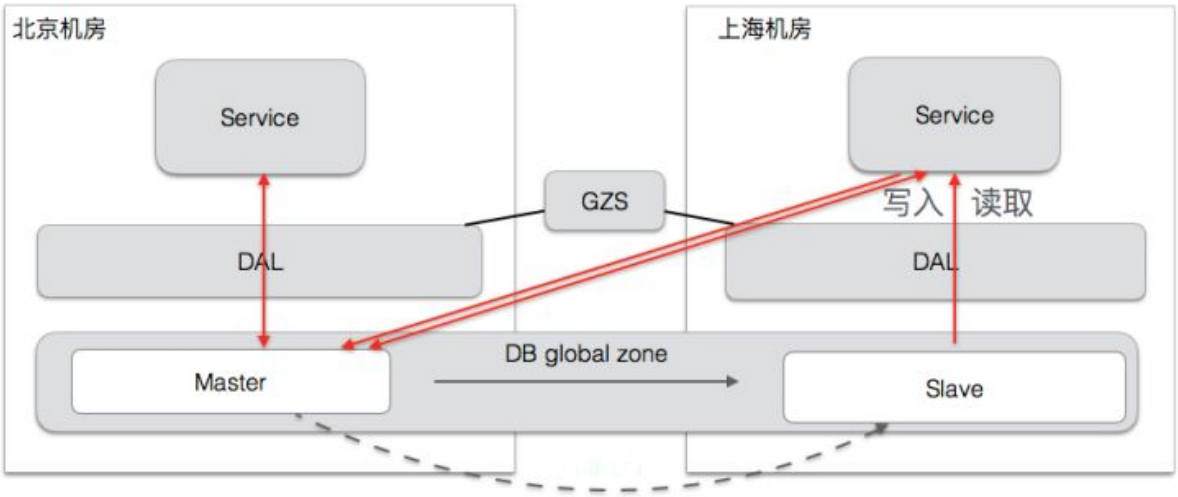
消息队列和Redis复制:

MQ,Redis 的复制与 ZK 复制类似，也开发了 相应的复制工具。

强一致保证:

对于个别一致性要求很高的应用，我们提供了一种强一致的方案（Global Zone），

Globa Zone是一种跨机房的读写分离机制，所有的写操作被定向到一个 Master 机房进行，以保证一致性，读操作可以在每个机房的 Slave库执行，也可以 bind 到 Master 机房进行，这一切都基于我们的数据库访问层（DAL）完成，业务基本无感知。



切换过程和各种异常保护:

避免数据错误非常重要，在网络断开，或者是切换过程中，特别容易产生错误数据。比如由于复制延时，订单状态不一致，用户有可能会重复支付。为了避免我们采取了一些保护措施，避免在切换时发生错误。

1. 在网络中断时，如果不是必要，不做切换，因为任意单个机房能够提供完整服务。
2. 如果需要切换，**对锁定切换过程中的订单，直到切换完成，数据复制正常，才开放锁定。**这个过程也通过 DAL 来实现
3. 对于标记为其他机房的写入数据，**DAL 会进行保护，拒绝写入。**
4. DRC 会检查并报告错误的写入操作，方便检查隐藏问题。

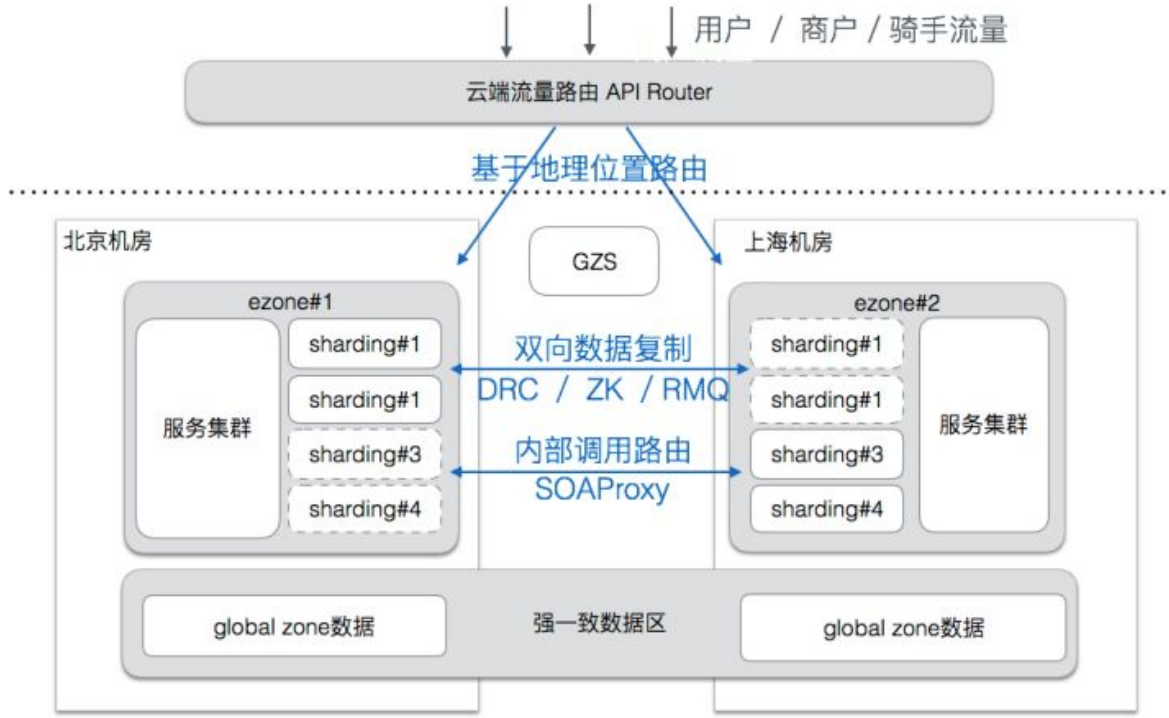
通过以上4条的保护，我们保证了数据的正确性，频繁的切换也不会出现异常的业务数据。

多个机房的Cache刷新：

数据的变更信息，通过 DRC 广播到多个机房，实现缓存的刷新，保证各个机房的缓存一致性。

整体结构

以上介绍了各个考虑的方面，现在可以综合起来看，饿了么多活的整体结构如下图：



业务改造：

业务可感知是一条基本原则，通过中间件提供的服务，多活逻辑会暴露给业务方，例如：当前服务所属的 ezone，路由策略，数据的归属 shard 等，基于这些信息，业务可以执行很多的逻辑。包括：

1. 后台任务可以过滤掉非本 ezone 的数据。
2. 可以在发生切换时，执行特定的逻辑，触发特定动作。
3. 业务需要准备一些数据修复逻辑，在万一发生不一致时，手工或者自动纠正数据。

未来：下一步多活的计划

目前饿了么的服务已经部署到2个异地机房，下一步我们会扩展到3-4个机房，并且在公有云上建立一个新的ezone，充分利用公有云的强大的扩展能力，未来我们将能够快速的在全世界各地搭建数据中心，也能够快速的利用各种公有云基础设施，实现全球规模的高可用和扩展性。