云计算设计模式

极客学院出版

前言

云带来的改变是显而易见的，云计算是一种按使用量付费的模式，这种模式提供可用的、便捷的、按需的网络访问， 进入可配置的计算资源共享池（资源包括网络，服务器，存储，应用软件，服务），这些资源能够被快速提供，只需投入很少的管理工作，或与服务供应商进行很少的交互。本文提供了 24 种云计算设计模式，能够让读者学习如何设计高可用性、高弹性、低运维、可监控与自动化的云计算平台。

适用人群

本文适合云计算爱好者，及目前从事和云计算相关工作的开发者或运维人员。

学习前提

本文以设计思想为主，你可以零基础学习。但是想要了解到具体是怎么实现的，你需要具备云计算的基础、了解分布式计算、虚拟化、数据存储、数据管理等技术。

鸣谢：

译文出处 http://blog.csdn.net/column/details/clouddesignpattern.html?&page=2

原文出处 MSDN：http://msdn.microsoft.com/en-us/library/dn589799.aspx

目录

1 缓存预留模式

缓存预留模式是根据需求从数据存储缓存加载数据。这种模式可以提高性能，并有助于维持在基础数据存储在高速缓存中保持的数据和数据之间的一致性。

背景和问题

应用程序使用的高速缓存来优化重复访问的数据存储中保持的信息。然而，它通常是不切实际的期望缓存的数据将始终与在数据存储器中的数据完全一致。应用程序要实现一种策略，有助于确保在高速缓存中的数据是最新的，只要有可能，但也可以检测和处理的过程中出现，当在高速缓存中的数据已经变得陈旧的情况。

解决方案

许多商业缓存系统提供通读和直写式/后写操作。在这些系统中，应用程序通过引用高速缓存中检索数据。如果数据不在缓存中，它被透明地从数据存储中检索并添加到高速缓存。任何修改在高速缓存中保持的数据被自动地写入到数据存储区以及。

为缓存不提供此功能，则使用该缓存保持在高速缓存中的数据的应用程序的责任。

一个应用程序可以通过实现高速缓存预留战略模拟的读式高速缓存的功能。这种策略有效地将数据加载需求的高速缓存。图 1 总结了在该过程中的步骤。

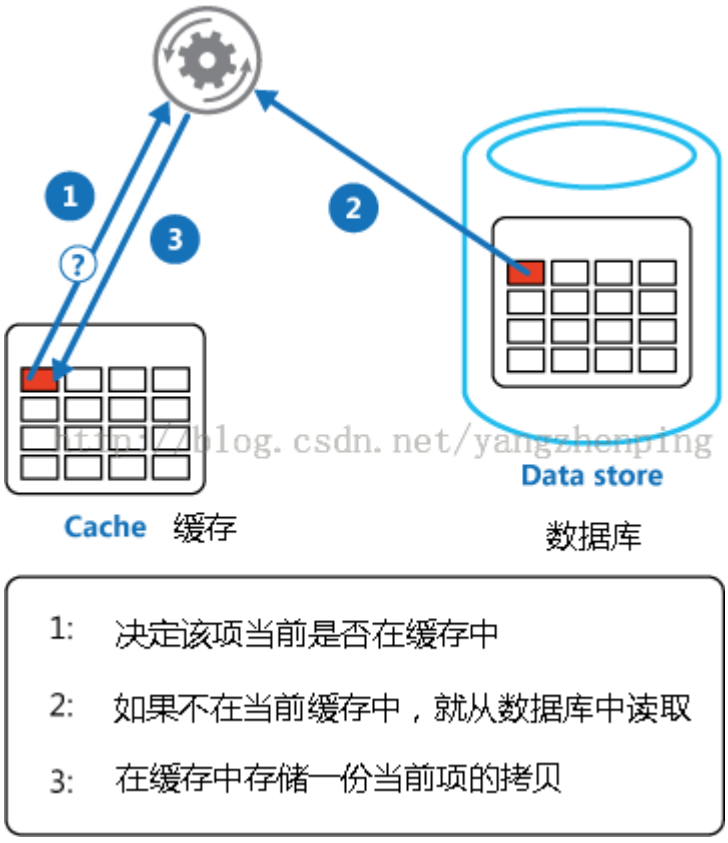


图1 – 使用Cache–除了图案来将数据存储在高速缓冲存储器

如果一个应用程序将更新的信息，它可以模拟通写策略如下：

- 根据修改到数据存储
- 作废对应的项在缓存中。

当该项目被下一个需要，可使用高速缓存预留策略将导致从数据存储中检索和重新添加到高速缓存中的更新数据。

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 缓存数据的生命周期。很多缓存实现一个过期策略，导致数据无效，并从缓存中移除如果它不是在指定时间内访问。对于缓存一边是有效的，确保了过期策略相匹配的访问用于使用数据的应用程序的模式。不要使有效期限太短，因为这会导致应用程序不断地从数据存储中检索数据，并将其添加到缓存中。同样，不要使保质期这么久，缓存的数据很可能会变得陈旧。记住，缓存是最有效的相对静态的数据，或者数据被频繁地读出。
- 驱逐数据。最高速缓存具有比从其中数据源自数据存储区只有有限的大小，并在必要时它们将收回的数据。大多数缓存采用最近最少使用的政策选择项目驱逐，但是这可能是定制的。配置全局到期属性和高速缓存的其它性能，并且每个高速缓存项的到期属性，以帮助确保缓存成本效益。它可能并不总是适合于高速缓存中的应用全球驱逐政策，每一个项目。例如，如果缓存项是非常昂贵的，从数据存储中检索，也可能是有益的，保留在更频繁地访问但不昂贵的物品的费用此产品的高速缓存中。
- 灌注缓存。许多解决方案，预填充用的应用程序可能需要作为启动处理的一部分的数据的高速缓存。如果某些数据已到期，被驱逐的缓存，除了图案可能仍然是有用的。
- 一致性。执行缓存除了图案不保证数据存储和高速缓存之间的一致性。在数据存储中的项目可以在任何时候被改变由外部的过程中，这种变化可能不反映在高速缓存中的项目被装载到高速缓存，直到下一次。在一个系统，整个数据存储复制数据，如果同步发生非常频繁这个问题可能会变得尤为突出。
- 本地（内存）缓存。缓存可以是本地的应用程序实例，并存储在内存中。缓存预留如果应用程序多次访问相同的数据可以在该环境中是有用的。然而，本地高速缓存是私有的，因此不同的应用程序实例可各自具有相同的缓存数据的副本。此数据可能很快变成高速缓存之间不一致，所以它可能有必要在到期专用高速缓存中保存的数据和更经常地刷新。在这些场景中它可能是适当的，调查使用了共享或分布式缓存机制。

当使用这个模式

使用这种模式时：

- 缓存不提供原生读通过，并通过写操作。
- 资源的需求是不可预测的。这种模式使应用程序能够按需加载数据。它使任何假设有关的数据的应用程序将需要提前。

这种模式可能不适合：

- 当缓存的数据集是静态的。如果数据将适合可用的高速缓存空间，首要的高速缓存中的数据在启动和应用，防止数据从止政策。
- 对于托管在 Web 场中的 Web 应用程序缓存会话状态信息。在这种环境下，你应该避免引入基于客户端 – 服务器关系的依赖。

例子

在微软的 Azure，您可以使用 Azure 的缓存来创建一个分布式缓存，可以通过一个应用程序的多个实例可以共享。下面的代码示例中的 GetMyEntityAsync 方法给出了基于 Azure 的缓存 Cache 后备模式的实现。此方法从利用读虽然方法缓存中的对象。

一个目的是确定用一个整数ID作为键。该 GetMyEntityAsync 方法生成基于此键（在 Azure 缓存 API 使用的键值字符串）的字符串值，并尝试检索与从缓存中这一关键的项目。如果匹配的项目被发现，它被返回。如果在缓存中没有匹配，则 GetMyEntityAsync 方法从一个数据存储中的对象时，把它添加到缓存中，然后将其返回（即实际上获得从数据存储中的数据的代码已经被省略，因为它是数据存储依赖）。注意，缓存项被配置以防止其成为陈旧如果是在别处更新过期。

```
private DataCache cache;
...

public async Task<MyEntity> GetMyEntityAsync(int id)
{
  // Define a unique key for this method and its parameters.
  var key = string.Format("StoreWithCache_GetAsync_{0}", id);
  var expiration = TimeSpan.FromMinutes(3);
  bool cacheException = false;

  try
```

```
{
  // Try to get the entity from the cache.
  var cacheItem = cache.GetCacheItem(key);
  if (cacheItem != null)
  {
    return cacheItem.Value as MyEntity;
  }
}
catch (DataCacheException)
{
  // If there is a cache related issue, raise an exception
  // and avoid using the cache for the rest of the call.
  cacheException = true;
}

// If there is a cache miss, get the entity from the original store and cache it.
// Code has been omitted because it is data store dependent.
var entity = ...;

if (!cacheException)
{
  try
  {
    // Avoid caching a null value.
    if (entity != null)
    {
      // Put the item in the cache with a custom expiration time that
      // depends on how critical it might be to have stale data.
      cache.Put(key, entity, timeout: expiration);
    }
  }
  catch (DataCacheException)
  {
    // If there is a cache related issue, ignore it
    // and just return the entity.
  }
}

return entity;
}
```

> 注意：
>
> 该示例使用了 Azure 的缓存 API 来访问存储和检索的缓存信息。有关 Azure 的缓存 API 的更多信息，请参阅 MSDN 上使用微软的 Azure 缓存。

> 下面所示的 UpdateEntityAsync 方法说明如何在高速缓存中的对象无效，当该值是由应用程序改变。这是一个写通方法的实例。该代码更新原始数据存储，然后通过调用 Remove 方法，指定键（这部分功能的代码已经被省略了，因为这将是数据存储相关）从缓存中删除缓存项。

注意

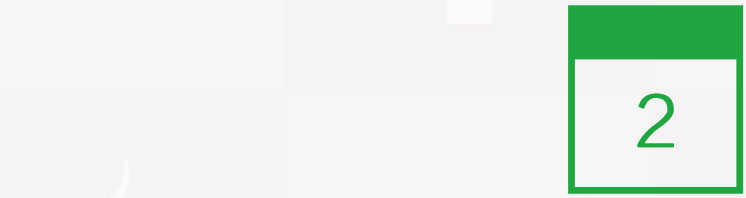
在这个序列中的步骤的次序是重要的。如果之前的缓存更新的项被删除，对于客户端应用程序中的数据存储中的项目之前获取的数据（因为它没有在高速缓存中发现的）的机会已经改变一个小窗口，从而在缓存包含过期数据。

```
public async Task UpdateEntityAsync(MyEntity entity)
{
  // Update the object in the original data store
  await this.store.UpdateEntityAsync(entity).ConfigureAwait(false);

  // Get the correct key for the cached object.
  var key = this.GetAsyncCacheKey(entity.Id);

  // Then, invalidate the current cache object
  this.cache.Remove(key);
}

private string GetAsyncCacheKey(int objectId)
{
  return string.Format("StoreWithCache_GetAsync_{0}", objectId);
}
```



2

断路器模式

处理故障连接到远程服务或资源时，可能需要耗费大量的时间。这种模式可以提高应用程序的稳定性和灵活性。

背景和问题

在分布式环境中，如在云，其中，应用程序执行访问远程资源和服务的操作，有可能对这些操作的失败是由于瞬时故障，如慢的网络连接，超时，或者被过度使用的资源或暂时不可用。这些故障一般之后的短时间内纠正自己，和一个强大的云应用应该准备使用的策略来处理它们，例如，通过重试模式进行说明。

但是，也可以是其中的故障是由于那些不容易预见的突发事件的情况下，这可能需要更长的时间来纠正。这些故障从连接的部分损失到服务的完整的故障范围的严重程度。在这种情况下，它可能是毫无意义的应用，不断重试执行的操作是不太可能成功，而不是应用程序应该很快接受该操作已失败，并相应地处理这个故障。

此外，如果一个服务是非常繁忙的，在系统中的一个部分出现故障可能会导致连锁故障。例如，调用一个服务的操作可被配置成实现一个超时，如果该服务无法在这段时间内响应一个失败消息答复。然而，这一策略可能导致许多并发请求的相同的操作，直到超时时段期满被阻止。这些被禁止的请求可能会持有关键系统资源，如内存，线程，数据库连接等。因此，这些资源可能会耗尽，从而导致该系统的其他可能无关的部件，需要使用相同的资源的失败。在这些情况下，这将是优选的操作立即失败，并且只尝试调用服务，如果它是可能成功。注意，设置一个较短的超时可能有助于解决此问题，但在超时不应该如此之短，以致操作失败的大部分时间，即使该请求到服务最终会成功。

解决方案

该断路器图案可以防止一个应用程序多次试图执行一个操作，即很可能失败，允许它继续而不等待故障恢复或者浪费 CPU 周期，而它确定该故障是持久的。断路器模式也使应用程序能够检测故障是否已经解决。如果问题似乎已经得到纠正，应用程序可以尝试调用操作。

注意

断路器图案的目的是从该重试模式的不同。重试模式使应用程序可以重试操作以期望它会成功。断路器图案防止应用程序执行一个操作，即很可能失败。一个应用程序可以通过使用重试模式，通过一个断路器调用操作结合这两种模式。然而，在重试逻辑应该是由断路器返回任何异常敏感和放弃重试次数，如果断路器指示故障不是瞬时的。

断路器充当可能失败操作的代理。代理应监测最近发生的故障数量，然后使用这个信息来决定是否允许该操作继续进行，或简单地立即返回一个异常。

代理可以被实现为状态机与模拟的电路断路器的功能如下状态：

关闭：从应用程序的请求是通过对操作进行路由。代理保持最近的失败次数的计数，并且如果该呼叫到操作不成功，则代理递增该计数。如果最近的失败次数超过了一个给定时间周期内的规定的阈值时，该代理将被置于打开状态。在这一点上的代理启动一个超时定时器，当该定时器期满的代理放置到半开放状态。

注意

超时定时器的目的是为了给系统时间，纠正允许应用程序尝试再次执行该操作之前导致失败的问题。

- 打开：从应用程序请求立即失败和异常返回给应用程序。
- 半开放：从应用程序请求的数量有限允许通过并调用运行。如果这些请求是成功的，则假定先前导致故障的故障已修复和断路器切换到闭合状态（故障计数器被复位）。如果任何请求失败，断路器假设故障仍然存在，因此恢复到打开状态，并重新启动超时定时器，系统的时间再延长，从故障中恢复。

注意

半开的状态是很有用的，以防止恢复服务，从突然被淹没的请求。作为服务恢复，也可能是能够支持请求的限制音量，直到恢复完成，但在恢复过程中，海量的工作可能会导致服务超时或再次失败。

下图展示出了用于一个可能的实现的电路断路器的状态。

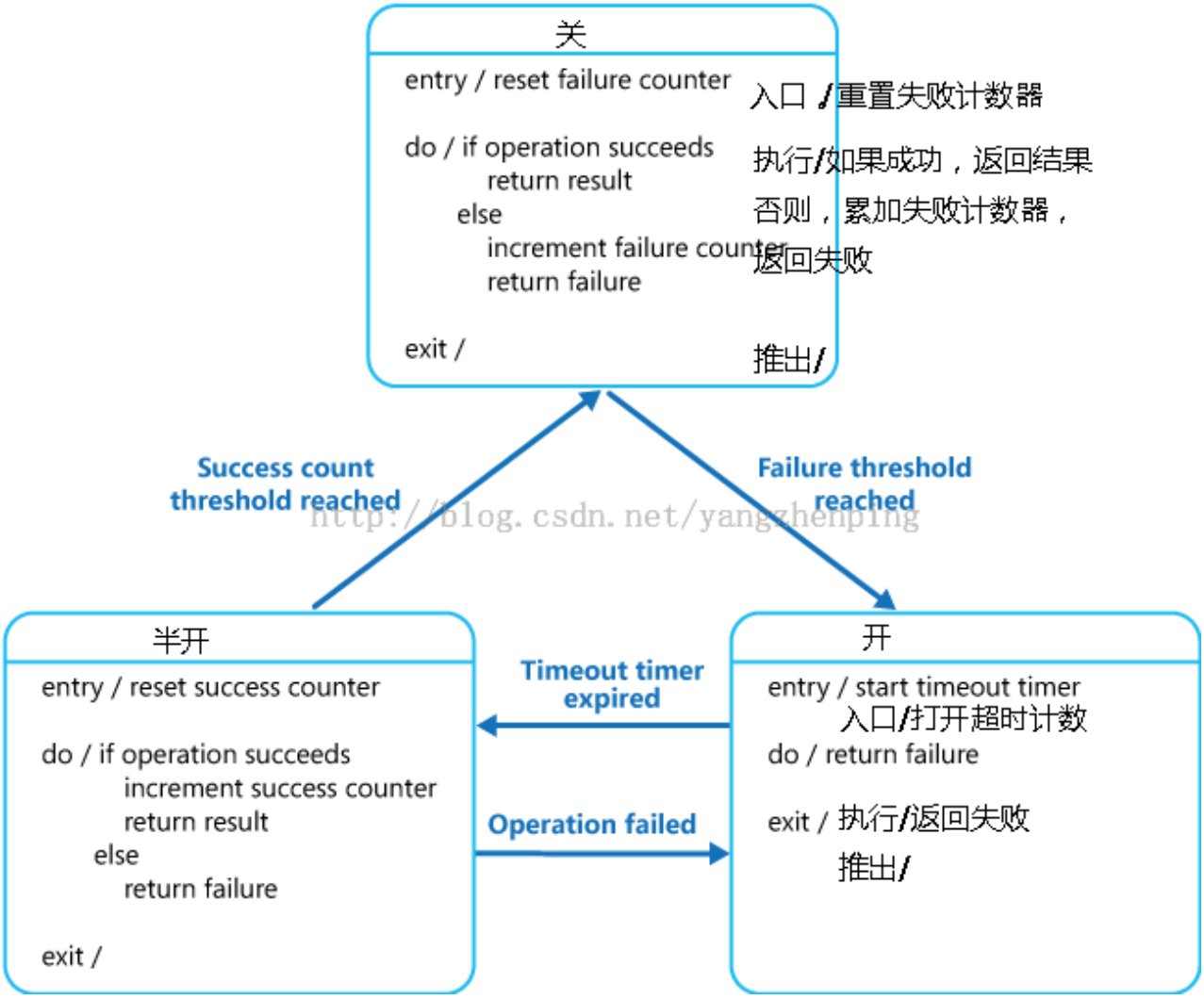


图 1 – 断路器状态

需要注意的是，在图 1 中，所用的封闭状态下的失败计数器是基于时间的。它以定期自动复位。这有助于防止断路器进入打开状态，如果它经受偶然的失败;这使断路器跳闸进入打开状态的故障阈值时，故障的指定数量的指定的时间间隔期间发生的仅达到。所使用的半开状态下的成功计数器记录成功尝试调用的操作的数量。断路器恢复到封闭状态后的连续操作调用中指定数量的已成功。如果任何调用失败时，断路器立即进入打开状态，并且成功的计数器将其进入半开状态下一次复位。

Note

如何将系统恢复从外部处理，可能通过恢复或重新启动故障部件或修理的网络连接。

执行断路器图案增加了稳定性和灵活性，以一个系统，提供稳定性，而系统从故障中恢复，并尽量减少此故障的对性能的影响。它可以帮助快速地拒绝对一个操作，即很可能失败，而不是等待操作超时（或者不返回）的请

求，以保持系统的响应时间。如果断路器提高每次改变状态的时间的事件，该信息可以被用来监测由断路器保护系统的部件的健康状况，或以提醒管理员当断路器跳闸，以在打开状态。

模式是可定制的，并且可以根据可能的故障的性质进行调整。例如，您可以申请增加的超时时间为一个断路器。可以放置在打开状态的断路器的几秒钟开始，然后，如果故障一直没有解决增加超时到几分钟的时间，等等。在某些情况下，而不是打开状态返回故障并提高了异常，也可能是有用的，返回一个缺省值，该值是有意义的应用。

问题和注意事项

在决定如何实现这个模式时，您应考虑以下几点：

- 异常处理。通过断路器调用操作的应用程序必须准备好处理，如果该操作是不可用的，可以被抛出的异常。在这样的异常处理将特定应用程序的方式。例如，一个应用程序可以暂时降低其功能，调用替换操作来尝试执行相同的任务或获得相同的数据，或者报告该异常给用户，并要求他们稍后再试。
- 例外的类型。一个请求可能失败的原因有多种，其中有一些可能指示更严重的类型的失效比其他。例如，一个请求可能会失败，因为远程服务已经崩溃了，可能需要几分钟才能恢复，或失败可能是由于该服务被暂时超载造成的超时时间。一种断路器可能能够检查发生的异常的类型，并根据这些异常的性质调整策略。例如，它可能需要的超时异常更大数目的断路器的开状态相比失败次数跳闸由于服务是完全不可用。
- 日志记录。一个断路器应记录所有失败的请求（也可能是成功的请求），以使管理员能够监视它封装了操作的健康。
- 可恢复性。您应该配置断路器与之相匹配的是保护的操作可能恢复模式。例如，如果断路器保持在打开状态下很长一段时间，也可能产生异常，即使对于失败的原因早已得到了解决。类似地，一个断路器可以振荡并降低应用程序的响应时间，如果它从打开状态到半开状态太快切换。
- 测试失败的操作。在打开状态下，而不是使用一个计时器来确定何时切换到半开放状态下，断路器可代替周期性地查验远程服务或资源，以确定它是否已经再次变得可用。这个平可以采取的企图的形式援引了以前失败的操作，也可以使用由远程服务提供的特殊操作专门用于测试服务的健康状况，所描述的卫生端点监测图案。
- 手动覆盖。在一个系统中，如果恢复时间为一个失败的操作是非常可变的，它可能是有利的，以提供一个手动复位选项，使管理员能够强行关闭断路器（和复位的故障计数器）。同样，管理员可以强制断路器进入开放状态（并重新启动超时定时器），如果由断路器保护动作暂时不可用。
- 并发。相同的电路断路器可以通过大量的应用程序的并行实例来访问。实施不应该阻塞并发请求或添加过多的开销，以每次调用操作。

- 资源分化。使用单个断路器时，一个类型的资源，如果可能有多个潜在的独立供应商要小心。例如，在数据存储器，其包括多个碎片，1分片可以是完全可访问的，而另一个是经历一个暂时的问题。如果在这些情况下的错误响应被合二为一，应用程序可能试图访问一些碎片，即使发生故障的可能性高，同时获得其他碎片，即使它是可能成功的可能被堵塞。
- 加速断路。有时失败响应可以包含足够的信息用于断路器的实施知道它应当立即跳闸并保持处于跳闸状态的最小时间量。例如，从该过载的共享资源的错误响应可以指示立即重试时不推荐使用，并且该应用程序应代替再次尝试在几分钟时间。

Note

HTTP 协议定义的“HTTP503 服务不可用”，它可以如所请求的服务是当前不可用的特定的 Web 服务器上的被返回的响应。此响应可以包括附加信息，例如延迟的预期持续时间。

- 重播失败的请求。在打开状态下，而不是简单的故障很快，断路器也可以记录每个请求的详细信息，以轴颈和安排这些请求时，远程资源或服务变得可用重放。
- 对外部服务不当超时。电路断路器可能无法充分保护的应用程序，从失败中配置有一个漫长的超时时间对外服务业务。如果超时太长，运行一个断路器的螺纹可能被堵塞长时间之前断路器指示操作已失败。在这个时候，许多其他的应用程序实例也可以尝试通过断路器来调用服务，并占用一个显著的线程数之前，他们都失败。

当使用这个模式

使用这种模式：

- 为了防止一个应用程序试图调用一个远程服务或访问共享资源，如果该操作是极有可能失败。

这种模式可能不适合：

- 对于处理中的应用程序访问本地专用资源，例如在存储器内数据结构。在这种环境下，使用断路器只会增加开销到您的系统。
- 作为一个替代品来处理异常在应用程序的业务逻辑。

例子

在 Web 应用中，几个页面的已填充了从外部服务中检索数据。如果该系统实现了最小的缓存，点击率最高的为每个页面都会导致往返服务。从 Web 应用程序到服务的连接可以用一个超时时间段（通常为 60 秒）进行配置，并且如果该服务没有在这个时间响应在每个网页的逻辑将假设该服务不可用，并且抛出异常。

但是，如果服务失败，系统非常繁忙，用户可能会被迫等待异常发生时长达60秒前。最终的资源，如内存，连接和线程可能被耗尽，以防止其他用户连接到系统，即使它们没有访问检索业务数据的页面。

通过添加更多的 Web 服务器和执行负载均衡扩展，系统可能会延误的点资源趋于枯竭，但它不会解决问题，因为用户请求仍然会反应迟钝，所有的 Web 服务器仍然可以最终耗尽资源。

包裹连接到服务，并检索数据中的断路器的逻辑可以帮助缓解这个问题的影响，并且更优雅处理服务故障。用户请求仍然会失败的，但它们将更加迅速地失败，并且资源不会被阻塞。

该 `CircuitBreaker` 类维护有关的对象，它实现下面的代码所示 `ICircuitBreakerStateStore` 接口电路断路器的状态信息。

```
interface ICircuitBreakerStateStore
{
  CircuitBreakerStateEnum State { get; }

  Exception LastException { get; }

  DateTime LastStateChangedDateUtc { get; }

  void Trip(Exception ex);

  void Reset();

  void HalfOpen();

  bool IsClosed { get; }
}
```

状态属性指示断路器的当前状态，以及由 CircuitBreakerStateEnum 枚举所定义的将是这些值中的一个程序，HalfOpen，或者已关闭。如果电路断路器闭合，但如果其打开或半开的 IsClosed 属性应该是真实的。跳闸方法切换断路器为打开状态的状态，并记录该引起状态变化的异常，与所发生的异常的日期和时间一起。该 LastException 和 LastStateChangedDateUtc 属性返回此信息。复位方法关闭断路器和 HalfOpen 方法将断路器半开。

在该实例中 InMemoryCircuitBreakerStateStore 类包含 ICircuitBreakerStateStore 接口的实现。该 CircuitBreaker 类创建这个类的一个实例来保存断路器的状态。

在 CircuitBreaker 类的 ExecuteAction 方法包装的操作（在 Action 委托的形式）可能会失败。当该方法运行时，它首先检查断路器的状态。如果它被关闭（当地 IsOpen 属性，如果断路器处于打开状态或半开，返回真，是假的）的 ExecuteAction 方法试图调用 Action 委托。如果此操作失败，异常处理程序执行 TrackException 方法，用于设置该电路断路器的状态通过调用 InMemoryCircuitBreakerStateStore 物体的行程的方法打开。下面的代码示例强调了这一流程。

```
public class CircuitBreaker
{
  private readonly ICircuitBreakerStateStore stateStore =
    CircuitBreakerStateStoreFactory.GetCircuitBreakerStateStore();

  private readonly object halfOpenSyncObject = new object ();
  ...
  public bool IsClosed { get { return stateStore.IsClosed; } }

  public bool IsOpen { get { return !IsClosed; } }

  public void ExecuteAction(Action action)
  {
    ...
    if (IsOpen)
    {
      // The circuit breaker is Open.
      ... (see code sample below for details)
    }

    // The circuit breaker is Closed, execute the action.
    try
    {
      action();
    }
    catch (Exception ex)
    {
      // If an exception still occurs here, simply
      // re-trip the breaker immediately.
      this.TrackException(ex);

      // Throw the exception so that the caller can tell
      // the type of exception that was thrown.
      throw;
    }
```

```
  }

  private void TrackException(Exception ex)
  {
    // For simplicity in this example, open the circuit breaker on the first exception.
    // In reality this would be more complex. A certain type of exception, such as one
    // that indicates a service is offline, might trip the circuit breaker immediately.
    // Alternatively it may count exceptions locally or across multiple instances and
    // use this value over time, or the exception/success ratio based on the exception
    // types, to open the circuit breaker.
    this.stateStore.Trip(ex);
  }
}
```

下面的例子显示了执行，如果断路器没有关闭的代码（从前面的例子中省略）。它如果断路器已经开了一段时间长于当地 OpenToHalfOpenWaitTime 字段中 CircuitBreaker 类中指定的时间首先检查。如果是这种情况，则 ExecuteAction 方法设置断路器半开，然后尝试执行该行动代表所指定的操作。

如果操作成功，则断路器复位到闭合状态。如果操作失败，则跳闸恢复到打开状态，并且在发生被更新，以使断路器将等待进一步期间再次尝试执行该操作之前的异常所需的时间。

如果断路器至今只有开放的时间很短，小于 OpenToHalfOpenWaitTime 值时，ExecuteAction 方法简单地抛出 CircuitBreakerOpenException 异常和返回引发的断路器转换到打开状态的误差。

此外，为了防止断路器试图执行并发呼叫的操作，同时它是半开的，它使用一个锁。兼试图调用该操作会如果断路器是公开进行处理，如后所述，它会失败并异常。

```
...
  if (IsOpen)
  {
    // The circuit breaker is Open. Check if the Open timeout has expired.
    // If it has, set the state to HalfOpen. Another approach may be to simply
    // check for the HalfOpen state that had be set by some other operation.
    if (stateStore.LastStateChangedDateUtc + OpenToHalfOpenWaitTime < DateTime.UtcNow)
    {
      // The Open timeout has expired. Allow one operation to execute. Note that, in
      // this example, the circuit breaker is simply set to HalfOpen after being
      // in the Open state for some period of time. An alternative would be to set
      // this using some other approach such as a timer, test method, manually, and
      // so on, and simply check the state here to determine how to handle execution
      // of the action.
      // Limit the number of threads to be executed when the breaker is HalfOpen.
      // An alternative would be to use a more complex approach to determine which
      // threads or how many are allowed to execute, or to execute a simple test
      // method instead.
```

```
    bool lockTaken = false;
    try
    {
      Monitor.TryEnter(halfOpenSyncObject, ref lockTaken)
      if (lockTaken)
      {
        // Set the circuit breaker state to HalfOpen.
        stateStore.HalfOpen();

        // Attempt the operation.
        action();

        // If this action succeeds, reset the state and allow other operations.
        // In reality, instead of immediately returning to the Open state, a counter
        // here would record the number of successful operations and return the
        // circuit breaker to the Open state only after a specified number succeed.
        this.stateStore.Reset();
        return;
      }
      catch (Exception ex)
      {
        // If there is still an exception, trip the breaker again immediately.
        this.stateStore.Trip(ex);

        // Throw the exception so that the caller knows which exception occurred.
        throw;
      }
      finally
      {
        if (lockTaken)
        {
          Monitor.Exit(halfOpenSyncObject);
        }
      }
    }
  }
  // The Open timeout has not yet expired. Throw a CircuitBreakerOpen exception to
  // inform the caller that the caller that the call was not actually attempted,
  // and return the most recent exception received.
  throw new CircuitBreakerOpenException(stateStore.LastException);
}
...
```

使用 CircuitBreaker 对象，以保护操作时，应用程序创建的 CircuitBreaker 类的一个实例，并调用 ExecuteAction 方法，指定的操作被作为参数来执行。该应用程序应该准备，如果操作失败，因为断路器处于打开状态，以赶上 CircuitBreakerOpenException 例外。下面的代码显示了一个示例：

```
var breaker = new CircuitBreaker();

try
{
  breaker.ExecuteAction(() =>
  {
    // Operation protected by the circuit breaker.
    ...
  });
}
catch (CircuitBreakerOpenException ex)
{
  // Perform some different action when the breaker is open.
  // Last exception details are in the inner exception.
  ...
}
catch (Exception ex)
{
  ...
}
```

3

补偿交易模式

撤消由一系列步骤，它们共同限定了最终一致性操作中，如果一个或多个步骤失败执行的工作。按照最终一致性模型，业务实现复杂的业务流程和工作流的云托管的应用程序中很常见。

背景和问题

在云中运行的应用程序频繁修改数据。此数据可跨在各种地理位置的所保持的数据源的一个品种传播。为了避免争用，并提高在分布式环境中，例如这样的性能，应用程序不应该试图提供强事务一致性。相反，应用程序应该实现最终一致性。在该模型中，一个典型的业务操作由一系列的独立的步骤。而正在执行这些步骤的系统状态的整体图可能是不一致的，但是，当操作完成并且所有步骤都被执行，系统应该重新变得一致。

注意

数据的一致性提供了入门为什么分布式事务不能很好地扩展更多的信息，并且巩固了最终一致性模型的原则。

在最终一致性模型的一个显著的挑战是如何处理失败无可挽回的一步。在这种情况下，可能需要撤消所有通过的操作中的前面的步骤完成的工作。然而，数据不能简单地被回滚，因为应用程序的其它并发实例可能已经改变，因为它。即使在数据没有被通过一并发实例变更的情况下，撤消一个步骤可能不是简单地恢复原始状态的问题。可能需要应用不同的业务特定的规则（参见实施例部分中描述的旅行网站）。

如果实现最终一致性操作跨越多个异构数据存储，解开在这样的操作中的步骤将需要访问的每个数据存储区中的转弯。在每一个数据存储区执行的工作必须可靠地复原到防止系统其余不一致。

不受实现最终一致性的操作的所有数据可能会在数据库中进行。在面向服务的架构（SOA）环境中的操作可能会调用一个服务动作，并导致由该服务保持状态的变化。要撤消的操作，这种状态的改变也必须是百废待兴。这可能涉及再次调用服务并执行该反转第一的影响另一个动作。

解决方案

落实补偿事务。在一个补偿事务的步骤必须撤消的原始操作的步骤的影响。补偿事务可能无法简单地与国家的制度在运行，因为这种方法可能会覆盖由应用程序的其他并发实例所做的更改开始取代目前的状态。相反，它必须是一个聪明的过程中，考虑到并发情况下进行的任何工作。这个过程通常是应用程序特定的，由原始操作所执行的工作的性质来驱动。

一种常见的方法来实现的，最终一致的操作，需要补偿的是使用的工作流。由于原来的动作的进行，系统记录每个步骤，以及如何通过该步骤完成的工作可以撤消信息。如果操作失败，在任何时候，在工作流倒卷回通过它已经完成的步骤，并执行反转每个步骤的工作。注意，补偿事务可能没有撤消的原始操作的精确镜面相反的顺序工作，并且它可能会执行一些并行撤销步骤。

注意

这种方法类似于英雄传奇策略。这一战略的描述是克莱门斯 Vasters 的博客在网上提供。

补偿事务本身是一个最终一致的操作，它也可能会失败。该系统应能够恢复补偿事务在故障点并继续。可能有必要重复发生故障的步骤，所以在补偿事务的步骤应该被定义为幂等的命令。有关幂等的详细信息，请参阅乔纳森 · 奥利弗的博客幂等模式。

在某些情况下，可能无法从该已失败，除非通过人工干预的步骤中恢复。在这种情况下，系统应发出警报，并提供尽可能多的信息尽可能了解失败的原因。

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 它可能不容易确定何时在实现最终一致性的动作的步骤已经失败。一个步骤可能不会立即失败，而是它可以阻止。可能有必要实现某种形式的超时机制。
- 补偿逻辑不容易推广。补偿事务是特定于应用程序;它依赖于具有足够的信息，以便能够撤消在一个失败的操作的每个步骤的效果的应用。
- 您应该定义的步骤在补偿事务的幂等命令。这使得，如果补偿事务本身不能被重复的步骤。
- 处理中原始操作的步骤，以及所述补偿事务的基础设施，必须是有弹性的。它一定不能失去，以补偿发生故障的步骤所需要的信息，而且它必须能够可靠地监视补偿逻辑的进度。
- 一个补偿事务并不一定在系统中返回数据的状态是在原操作的开始。相反，它补偿了由该成功完成操作失败之前的步骤中执行的工作。
- 在补偿事务中的步骤的顺序并不一定是反射镜相反的，在原来的操作的步骤。例如，一个数据存储可以是不一致比另一个更敏感，从而撤消更改到该商店中的补偿事务的步骤应首先发生。
- 在完成操作所需的每个资源放置一个短期的基于超时的锁，并提前获得这些资源，可以帮助增加的可能性，整体活动将取得成功。这项工作应执行的所有资源都被收购之后。所有操作必须完成的锁到期之前。
- 考虑使用重试逻辑比平常更多的宽容，尽量减少触发补偿事务失败。如果一个操作步骤，实现最终一致性失败，请尝试处理故障为一过性异常，并重复上述步骤。只有放弃操作，如果一个步骤反复或无可挽回地失败，启动补偿事务。

> 注意： 很多的挑战和实施补偿事务的问题是一样关心实现最终一致性。请参见注意事项实现了数据的一致性入门最终一致性的更多信息。

当使用这个模式

使用此模式仅适用于如果他们失败，必须撤销的操作。如果可能的话，设计解决方案，避免了需要补偿事务的复杂性（有关详细信息，请参阅数据一致性底漆）。

例子

一个旅游网站，使客户预订行程。一个单一的行程可包括一系列航班和酒店的。一位顾客旅行从西雅图到伦敦及巴黎可以创建一个行程时，请执行以下步骤：

1.预订一个座位上的 F1 航班从西雅图飞往伦敦。

2.预订一个座位上的 F2 航班从伦敦到巴黎。

3.书本占座 F3 航班从巴黎飞往西雅图。

4.预订的房间在伦敦酒店 H1。

5.预订在巴黎一间客房的酒店 H2。

这些步骤构成了最终一致的操作，虽然每一步基本上是在自己的权利单独的原子操作。因此，以及在执行这些步骤时，系统还必须记录必要撤消各以防客户决定取消行程步骤计数器的操作。必要执行计数器操作步骤，然后可以作为一个补偿事务如有必要运行。

请注意，在补偿事务中的步骤可能不是原来的步骤完全相反，并且在补偿事务的每个步骤必须考虑到任何特定于业务的逻辑规则。例如，“unbooking 取消预订”座位上的飞行可能不是客户有权向支付任何款项完成退款。

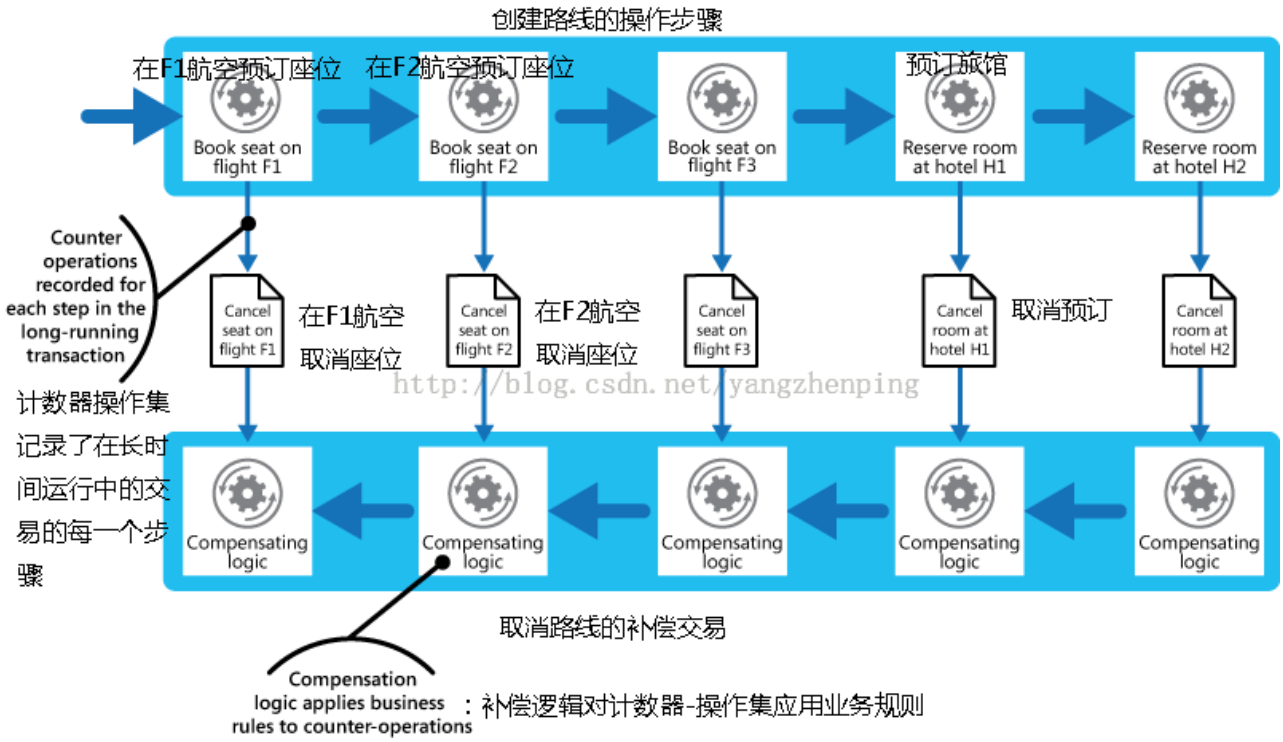


图1 – 生成一个补偿事务撤消一个长时间运行的事务预订旅游行程

Note

它可能会在并行执行的补偿事务的步骤，这取决于你如何设计每一步的补偿逻辑。

在许多商业解决方案，在单步的故障不总是必要轧制系统背面用补偿事务。例如，具有在旅游网站的情况，客户是无法预订到酒店H1预订航班 F1，F2 和 F3 的话，以后，最好是提供客户在同一个城市的房间在不同的酒店而不是取消航班。客户仍然可以选择取消（在这种情况下，补偿事务运行，并撤消作出关于航班 F1，F2 和 F3中的预订），但这个决定应该由客户而不是由系统进行。

4

消费者的竞争模式

允许多个并发用户处理在同一个通讯通道接收的消息。这种模式使系统能够同时处理多个邮件，以优化吞吐量，提高可扩展性和可用性，以及平衡工作负载。

背景和问题

在云中运行的应用程序，可以预计，以处理大量的请求。而不是过程的每个请求同步地，一个常用的方法是通过一个消息传送系统到该异步地处理它们的另一服务（消费者服务），以通过他们的应用程序。这种策略有助于确保在应用程序的业务逻辑没有被阻塞，而正在处理的请求。

请求的数量可以随着时间的原因有很多显著变化。突然一阵在用户活动或聚集的请求，来自多个租户未来可能会导致不可预测的工作负载。在高峰时间的系统可能需要处理许多每秒数百个请求，而在其他时间的数量可能是非常小的。此外，该工作的性质进行的处理这些请求可能是高度可变的。使用消费者服务的单个实例，可能会导致该实例成为充斥请求或消息传送系统可通过消息从应用程序来的流入被重载。为了处理这种波动的负载，该系统可以运行消费者服务的多个实例。然而这些消费者必须协调，以确保每个消息只传送给一个单个消费者。工作量也需要跨消费者被负载平衡，以防止一个实例成为瓶颈。

解决方案

使用消息队列来实现应用和消费者服务的实例之间的通信信道。在消息队列中的形式应用帖请求，以及消费者的服务实例从队列中接收消息并对其进行处理。这种方法使消费者的服务实例的同一池中从应用程序的任何实例处理消息。图 1 示出了该架构。

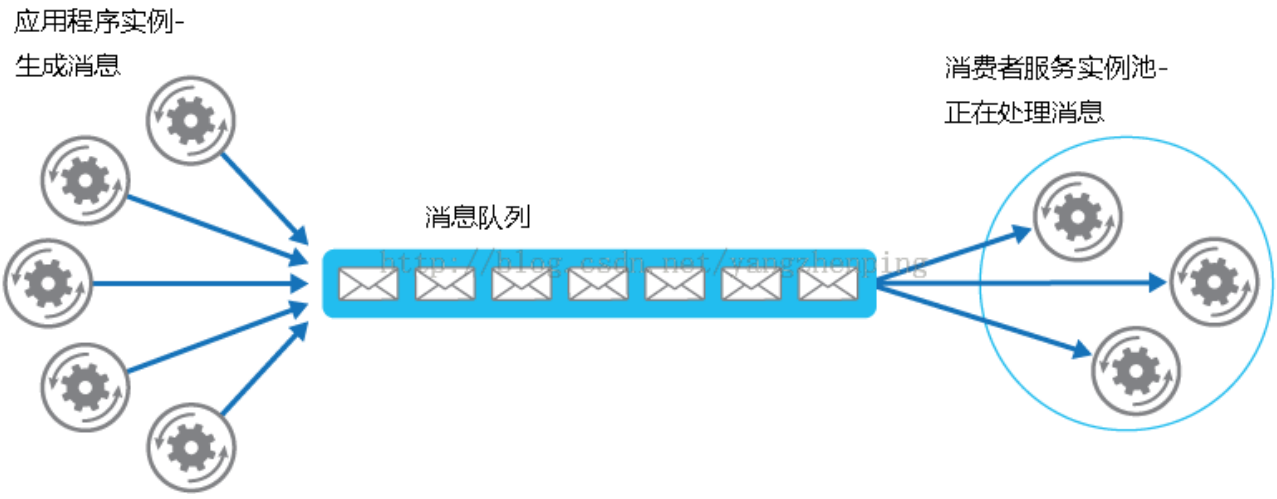


图1 – 使用消息队列分发工作提高到一个服务的实例

该解决方案具有以下优点：

- 它使固有的负载调平系统，可以处理由应用程序实例发送请求量很大的变化。队列充当应用程序实例和消费者服务实例，这有助于最大限度地减少对应用程序和服务实例（所描述的基于队列的负载调平模式）的可用性和响应性的影响之间的缓冲区。处理的消息，需要一些将被执行时，不会妨碍同时由消费者服务的其他实例所处理的其它消息长期运行的处理。
- 它提高了可靠性。如果一个生产者直接与消费者，而不是使用这种模式进行通信，但不监视消费者，有一个高概率的消息可能丢失或失败，如果消费者无法进行处理。在这种模式的消息不被发送给一个特定的服务实例，一个失败服务实例不会阻塞一个生产者和消息可以通过任何加工服务实例进行处理。
- 它不需要复杂的协调的消费者之间，或在生产者和消费者的实例。消息队列确保每个消息传递至少一次。
- 它是可扩展的。该系统能够动态地增加或减少消费者服务的实例的数目的消息量是波动的。
- 它可以提高弹性，如果消息队列提供事务读取操作。如果消费者服务实例能够读取和处理该消息作为一个事务操作的一部分，并且如果这种消费服务实例随后发生故障时，这种模式可以确保该消息将被返回到队列中被拾起并处理通过的另一个实例消费者服务。

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 留言订购。其中消费者服务实例接收消息的顺序是无法保证的，并且不一定反映了所创建的消息中的顺序。设计系统，以确保信息的处理是幂等的，因为这将有助于消除该消息的处理顺序上的任何依赖。有关幂等的详细信息，请参阅乔纳森 · 奥利弗的博客幂等模式。

注意

微软 Azure 服务总线队列可以通过使用消息会先入先出消息的顺序工具保证。欲了解更多信息，请参阅消息传递模式 MSDN 上使用会话。

- 设计服务的永续性。如果系统被设计为检测和重新启动失败的服务实例中，可能有必要执行由服务实例执行作为幂等操作，以最小化被检索和处理一次以上的单个消息的影响的处理。
- 检测有害消息。格式不正确的消息，或者需要访问不可用的资源的任务，可能会造成服务实例失败。该系统应避免这样的消息被返回到队列，而是捕获和别处存储这些信息的详细信息，以便可以在需要进行分析。
- 处理结果。服务实例处理一个消息从生成该消息的应用程序逻辑完全分离，并且它们未必能够直接进行通信。如果服务实例生成必须传回给应用程序逻辑结果，该信息必须被存储在一个位置，都可以访问两个和系统必须提供某种指示时的处理已经完成，以防止应用逻辑从检索数据不全。

注意

如果您正在使用 Azure 的工作进程可能能够通过使用专用的邮件回复队列回传结果的应用程序逻辑。应用逻辑必须能够将这些结果与原来的消息关联起来。这种情况下进行了更详细的异步消息的引物进行说明。

- 扩展的信息系统。在一个大型的解决方案，一个消息队列可以是不堪重负的消息的数量，并成为系统中的瓶颈。在这种情况下，考虑分割该消息系统直接从特定制造商的信息到一个特定的队列，或使用负载平衡，以跨多个消息队列分发消息。
- 邮件系统的可靠性保障。一个可靠的消息传送系统，需要保证的是，一旦应用程序放入队列的消息，它也不会丢失。这是确保所有邮件传递至少一次重要的。

当使用这个模式

使用这种模式时：

- 工作量为一个应用程序被分成可异步运行任务。
- 任务是独立的，可以并行地运行。
- 工作容积变化很大，需要一个可扩展的解决方案。
- 该解决方案必须提供高可用性，并且如果处理一个任务失败必须是有弹性的。

这种模式可能不适合时：

- 它是不容易的应用程序的工作负荷分离成离散的任务，或有任务之间的依赖程度高。
- 任务必须同步进行，而应用逻辑必须等待任务完成后再继续。
- 任务必须以特定的顺序来执行。

Note

有些邮件系统支持会话，使生产者对消息进行分组在一起，并确保它们都被同一个接收者处理。这个机制可以与优先消息使用（如果它们支持）来实现消息排序的一种形式，在顺序从生产者传送消息到单个消费者。

例子

Azure 提供存储队列和服务总线队列，可作为一个合适的机制来实现这种模式。应用逻辑可以发布消息到一个队列，而消费者实现为在一个或多个角色的任务可以检索从这个队列中的消息并进行处理。对于弹性，一个服务总线队列使得消费者使用 PeekLock 模式，当它从队列检索消息。这种模式实际上不是删除消息，而只是从其他消

费者隐藏它。当处理完它原来的用户可以删除该邮件。如果消费者要失败，偷看锁将超时，消息将再次变得可见，让消费者又找回它。

Note

有关使用 Azure 的服务总线队列的详细信息，请参阅服务总线队列，主题和 MSDN 上的订阅。有关使用 Azure 存储队列的信息，请参阅如何 MSDN 上使用队列存储服务。

从可供下载的例子 CompetingConsumers 解决方案的 QueueManager 类下面的代码显示了本指南说明了如何通过在网络或辅助角色开始的事件处理程序使用 QueueClient 实例中创建一个队列。

```
private string queueName = ...;
private string connectionString = ...;
...

public async Task Start()
{
  // Check if the queue already exists.
  var manager = NamespaceManager.CreateFromConnectionString(this.connectionString);
  if (!manager.QueueExists(this.queueName))
  {
    var queueDescription = new QueueDescription(this.queueName);

    // Set the maximum delivery count for messages in the queue. A message
    // is automatically dead-lettered after this number of deliveries. The
    // default value for dead letter count is 10.
    queueDescription.MaxDeliveryCount = 3;

    await manager.CreateQueueAsync(queueDescription);
  }
  ...

  // Create the queue client. By default the PeekLock method is used.
  this.client = QueueClient.CreateFromConnectionString(
    this.connectionString, this.queueName);
}
```

下面的代码片段显示了一个应用程序如何创建和发送一批消息队列。

```
public async Task SendMessagesAsync()
{
  // Simulate sending a batch of messages to the queue.
  var messages = new List<BrokeredMessage>();
```

```
  for (int i = 0; i < 10; i++)
  {
    var message = new BrokeredMessage() { MessageId = Guid.NewGuid().ToString() };
    messages.Add(message);
  }
  await this.client.SendBatchAsync(messages);
}
```

下面的代码显示了如何消费服务实例可以从队列中下一个事件驱动的方式接收消息。该 processMessageTask 参数的 ReceiveMessages 法为代表，它引用在收到消息时运行的代码。此代码是异步运行。

```
private ManualResetEvent pauseProcessingEvent;
...

public void ReceiveMessages(Func<BrokeredMessage, Task> processMessageTask)
{
  // Set up the options for the message pump.
  var options = new OnMessageOptions();

  // When AutoComplete is disabled it is necessary to manually
  // complete or abandon the messages and handle any errors.
  options.AutoComplete = false;
  options.MaxConcurrentCalls = 10;
  options.ExceptionReceived += this.OptionsOnExceptionReceived;

  // Use of the Service Bus OnMessage message pump.
  // The OnMessage method must be called once, otherwise an exception will occur.
  this.client.OnMessageAsync(
    async (msg) =>
    {
      // Will block the current thread if Stop is called.
      this.pauseProcessingEvent.WaitOne();

      // Execute processing task here.
      await processMessageTask(msg);
    },
    options);
}
...

private void OptionsOnExceptionReceived(object sender,
  ExceptionReceivedEventArgs exceptionReceivedEventArgs)
{
  ...
}
```

需要注意的是自动缩放的功能，例如可在天青，可用于启动和停止的角色实例的队列长度的波动。欲了解更多信息，请参阅自动缩放指导。另外，没有必要维持角色实例和工人之间的一对一的对应过程，单个角色实例可以实现多个工作进程。欲了解更多信息，请参阅计算资源整合模式。

5

计算资源整合模式

合并多个任务或操作成一个单一的计算单元。这种模式可以提高计算资源的利用率，并降低与云托管的应用程序进行计算处理相关的成本和管理开销。

背景和问题

云应用程序频繁执行各种操作。在某些解决方案也可能是有意义的最初遵循的关注点分离的设计原则，并把这些操作成托管和独立部署（例如，如在微软的 Azure 云服务，独立 Azure 网站不同的角色独立计算单元或单独的虚拟机）。然而，尽管这种策略可以帮助简化溶液的逻辑设计，部署大量计算单元作为同一应用可以提高运行时的托管成本，使系统的管理更加复杂的一部分。

作为一个例子，图1示出了使用多个计算单元被实现的一个云托管解决方案的简化的结构。每个计算单元在其自己的虚拟环境中运行。每个功能已被实现为一个单独的任务（通过任务Ë标任务A）在自己的计算设备上运行。

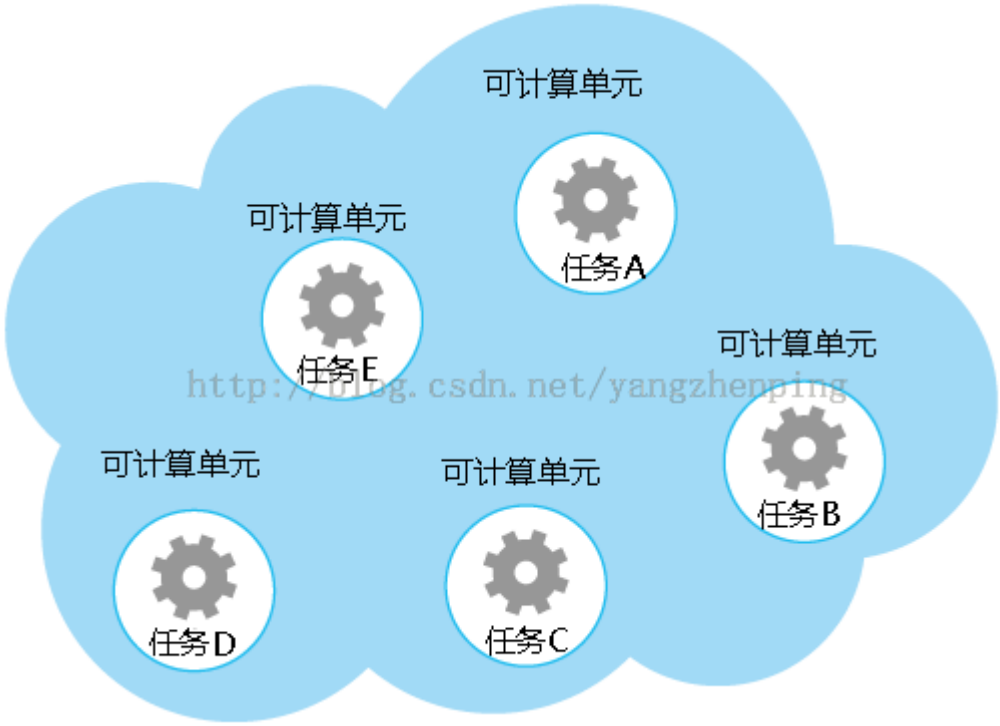


图1 – 通过使用一组专用计算单元运行在云环境中的任务

每个计算单元消耗的资源收费，即使是闲置或不常使用。因此，这种方法可能不总是最有成本效益的解决方案。

在 Azure 中，这一问题适用于云服务的角色，网站和虚拟机。这些产品在他们自己的虚拟环境中执行。运行的单独作用，网站，或者被设计为执行一组良好定义的操作的虚拟机的集合，但是需要进行通信和协作，作为一个单一的解决方案的一部分，可以是一个资源利用效率低。

解决方案

为了帮助降低成本，提高利用率，提高通信速度，减轻了管理工作有可能将多个任务或操作成一个单一的计算单元。

任务可以根据各种基于由环境提供的功能，以及与这些功能相关的成本的标准进行分组。一种常见的方法是寻找具有关于它们的可扩展性，寿命和加工要求具有相似的任务。分组这些产品一起使它们能够扩展为一个单元。由许多云环境所提供的弹性使一个计算单元的其他实例，以根据业务负载被启动和停止。例如，Azure提供自动缩放，可以适用于云服务的角色，网站和虚拟机。欲了解更多信息，请参阅自动缩放指导。

作为一个计数器的例子来说明如何扩展可以被用于确定哪些操作可能不应该被分组到一起，考虑以下两个任务：

- 任务 1 轮询发送到队列罕见的，对时间不敏感的信息。
- 任务 2 处理网络流量的高容量阵阵。

第二任务要求的弹性可能涉及起动和停止的大量的计算单元的实例。应用相同的缩放到第一任务只会导致更多的任务上监听同一队列不频繁的消息，并且是一种资源的浪费。

在许多云环境中，它可以指定在CPU内核，存储器，磁盘空间等的数量而言，以一个计算单元的可用资源。通常，指定的资源越多，就越有成本。对于金融效率，最大限度地工作的一个昂贵的计算单元执行的数量，而不是让它变成无活性在较长时间内是很重要的。

如果存在需要大量的 CPU 功率的短脉冲串的任务，考虑合并这些成一个单一的计算单元，其提供所需的电源。然而，重要的是平衡这种需要保持昂贵资源忙对它们是否过分强调指出可能发生了争用是重要的。长时间运行，计算密集型任务可能不应该共享相同的计算单位，例如。

问题和注意事项

实施该模式时请考虑以下几点：

- 可扩展性和弹性。许多云解决方案实现的可扩展性和弹性，在运算部的通过启动和停止的情况下，单位的水平。避免了分组在同一计算单元相互矛盾的可扩展性要求的任务。
- 一生。云计算基础架构可以定期回收托管的计算单元的虚拟环境。当执行一个计算单元内许多长期运行的任务，可能需要对设备进行配置，以防止它被回收，直到这些任务已经完成。可替换地，通过使用一个检查指向的方法，使他们停止干净，并继续在在其中，当所述计算单元被重新启动他们被中断的点的设计的任务。

- 释放节奏。如果一个任务的执行或配置变化频繁，则可能需要停止计算单位主办的更新的代码，重新配置和重新部署的单元，然后重新启动它。此过程也将需要相同的计算单元中的所有其他任务被停止，重新部署，并重新启动。
- 安全性。在相同的计算单元的任务可以共享相同的安全上下文，并能够访问相同的资源。必须有高度的任务之间的信任，而且相信，一个任务是不会损坏或其他不利的影响。此外，增加了在一个计算单元可以增加计算单元的攻击面运行的任务的数目;每个任务是否安全的一个最脆弱性。
- 容错。如果在一个计算单元中的一个任务失败或异常情况，它可能会影响在同一单元内运行的其他任务。例如，如果有一个任务无法正常启动它可能会导致对计算单元失败，整个启动逻辑，并且防止在同一单元的其他任务的运行。
- 争。避免这种情况，在相同的计算单元争夺资源的任务之间引入的争用。理想情况下，共享相同的计算单元的任务应该表现出不同的资源利用率的特征。例如，两个计算密集型任务可能不应该驻留在同一个计算单元，而且消耗大量的内存也不应该两个任务。然而，混合使用需要大量的存储器可以是一个可行的组合任务中计算密集型的任务。

注意

你应该考虑整合计算资源只对已在生产用于在一段时间内，使得操作人员和开发者能够监控系统，并创建热图，它标识了每个任务利用别共资源的系统。此图可以用于确定哪些任务是很好的候选共享计算资源。

- 复杂性。组合多个任务到一个单一的计算单元增加了复杂性中的代码单元，可能使得更难以进行测试，调试和维护。
- 稳定的逻辑架构。设计和实施中的代码中的每个任务，以便它不应该需要改变，即使物理环境中任务运行不会改变。
- 其他策略。整合计算资源的方法只有一个，以帮助减少同时运行多个任务相关的成本。这需要仔细的规划和监测，以确保它仍然是一个有效的办法。其他策略可能更合适，这取决于所执行的工作的性质和所代表这些任务正在运行的用户的位置。例如，工作负荷（如所描述的计算分区指南）的功能分解可能是一个更好的选择。

当使用这个模式

使用这种模式的任务，如果他们在自己的计算单元运行不符合成本效益。如果一个任务花费大量的时间闲置，运行此任务的专用设备可以是昂贵的。

这种模式可能不适合于执行关键容错操作处理高度敏感的或私有数据，并且需要其自身的安全上下文的任务或任务。这些任务应该在他们自己的独立的环境中运行，在一个单独的计算单元。

例子

在 Azure 上构建一个云服务，它可以巩固多任务的处理成一个单一的角色。通常，这是执行的背景或异步处理任务的辅助角色。

注意

在某些情况下它可能会包括在 Web 角色的背景或异步处理任务。这种技术可以有助于降低成本和简化部署，虽然它可以影响由 web 角色所提供的面向公众的接口的可扩展性和响应性。该文章合并多个天青工作者角色成天青 Web 角色包含执行背景或异步处理任务在 Web 角色的详细描述。

的作用是负责启动和停止的任务。当 Azure 结构控制器加载的作用，它引发的启动事件中的作用。您可以覆盖 WebRole 或 WorkerRole 类的 OnStart 方法来处理这个事件，也许是为了初始化数据和其他资源，在这种方法中，任务依赖。

当 OnStart 方法完成后，角色就可以开始响应请求。您可以找到有关使用的 OnStart 和运行方式的作用，在 the Application 启动进程中的模式与实践指南移动应用程序到云部分的更多信息和指导。

注意

请 OnStart 方法尽量精简的代码。 Azure 不上采取这种方法，完成时间强加任何限制，但作用不能够启动响应发送给它，直到此方法完成的网络请求。

当 OnStart 方法完成后，执行任务的运行方式。在这一点上，该织物控制器能够开始发送请求的作用。 将实际的运行方法创建任务的代码。注意，执行命令的方法可以有效地定义角色实例的生命周期。此方法完成，结构控制器将安排的作用被关闭。

当一个角色关机或再循环，结构控制器可以防止从负载平衡器接收任何更多的传入请求，并提高了停止事件。您可以通过覆盖作用的 onStop 方法捕获这个事件和角色终止前需要进行任何整理起来。

注意

在的 onStop 方法执行的任何操作须在 5 分钟（或者，如果您使用的是本地计算机上的天青模拟器 30 秒）内完成;否则 Azure 结构控制器假定的角色已经停止，并会迫使它停下来。

图 2 示出了一个角色的生命周期，任务和资源，它承载。该任务由运行方法，该方法然后等待任务来完成启动。任务本身，这实现云服务的业务逻辑，可以响应通过天青负载平衡器发布到角色的消息。

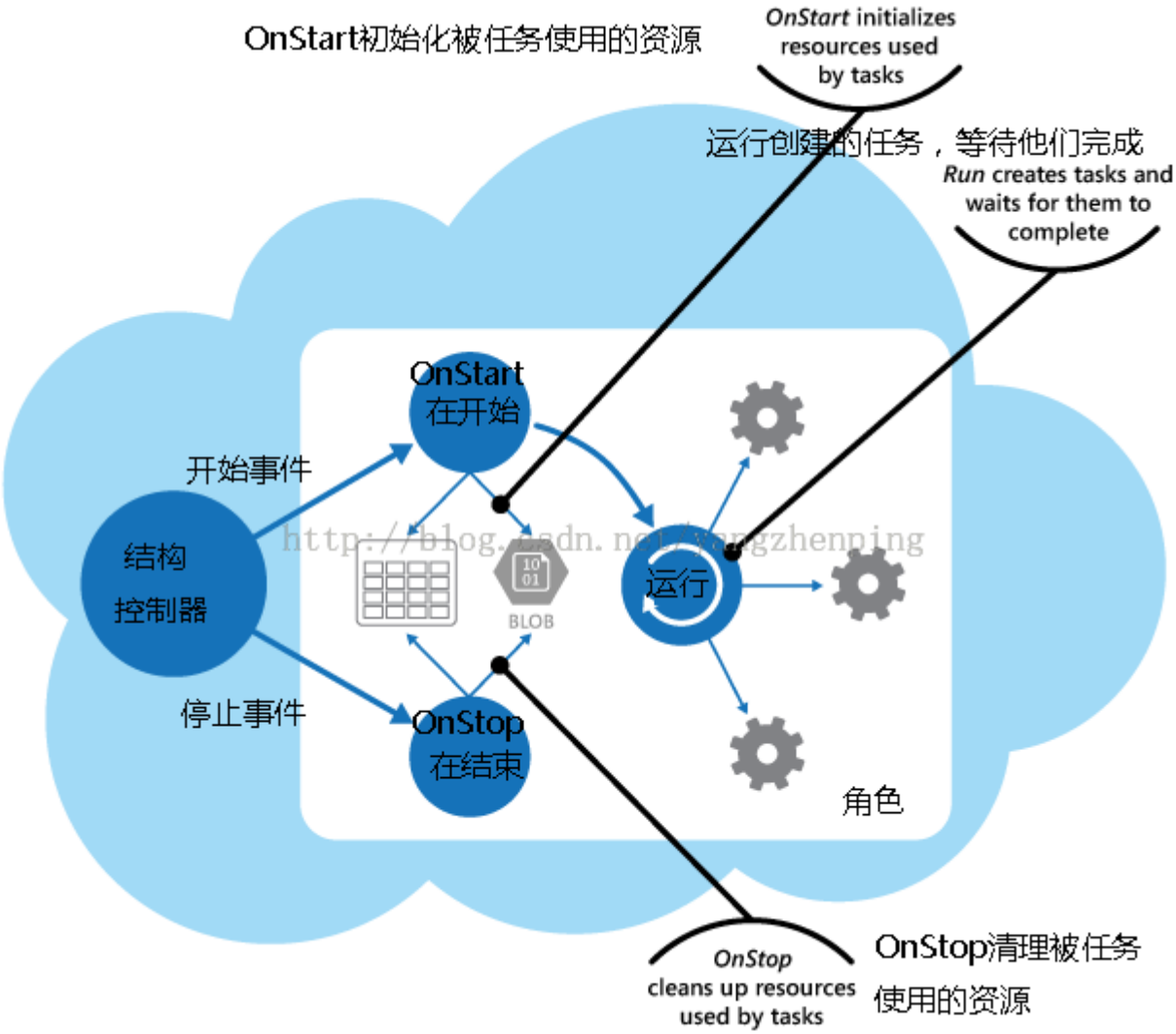


图2 – 任务和资源的作用，在Azure云服务的生命周期

在 ComputeResourceConsolidation.Worker 项目 WorkerRole.cs 文件显示了一个如何实现这个模式在 Azure 云服务的例子。

> 注意： 该 ComputeResourceConsolidation.Worker 项目是 ComputeResourceConsolidation 解决方案，可用于下载本指导意见的一部分。 在运行时被初始化的角色创建所需的取消标记和任务来运行的一个列表中的辅助角色，代码。

```
public class WorkerRole: RoleEntryPoint
{
 // The cancellation token source used to cooperatively cancel running tasks.
 private readonly CancellationTokenSource cts = new CancellationTokenSource ();
```

```
  // List of tasks running on the role instance.
  private readonly List<Task> tasks = new List<Task>();

  // List of worker tasks to run on this role.
  private readonly List<Func<CancellationToken, Task>> workerTasks
                    = new List<Func<CancellationToken, Task>>
   {
     MyWorkerTask1,
     MyWorkerTask2
   };

  ...
}
```

设置在 MyWorkerTask1 和 MyWorkerTask2 方法来说明如何在同一辅助角色执行不同的任务。下面的代码显示 MyWorkerTask1。这是休眠 30 秒，然后输出一个跟踪消息的简单任务。重复这个过程，直到无限期的任务被取消。在 MyWorkerTask2 代码非常相似。

```
// A sample worker role task.
private static async Task MyWorkerTask1(CancellationToken ct)
{
  // Fixed interval to wake up and check for work and/or do work.
  var interval = TimeSpan.FromSeconds(30);

  try
  {
    while (!ct.IsCancellationRequested)
    {
      // Wake up and do some background processing if not canceled.
      // TASK PROCESSING CODE HERE
      Trace.TraceInformation("Doing Worker Task 1 Work");

      // Go back to sleep for a period of time unless asked to cancel.
      // Task.Delay will throw an OperationCanceledException when canceled.
      await Task.Delay(interval, ct);
    }
  }
  catch (OperationCanceledException)
  {
    // Expect this exception to be thrown in normal circumstances or check
    // the cancellation token. If the role instances are shutting down, a
    // cancellation request will be signaled.
    Trace.TraceInformation("Stopping service, cancellation requested");

    // Re-throw the exception.
```

```
    throw;
  }
}
```

> 注意： 通过示例代码示出的方法是一个后台进程的一个常见的实现。在现实世界的应用程序，你可以按照这个结构相同，不同之处在于，你应该把自己的处理逻辑在等待取消请求的循环体。

经过工人的角色已初始化它使用的资源，Run 方法启动两个任务同时，如下图所示。

```
...
// RoleEntry Run() is called after OnStart().
// Returning from Run() will cause a role instance to recycle.
public override void Run()
{
  // Start worker tasks and add them to the task list.
  foreach (var worker in workerTasks)
    tasks.Add(worker(cts.Token));

  Trace.TraceInformation("Worker host tasks started");
  // The assumption is that all tasks should remain running and not return,
  // similar to role entry Run() behavior.
  try
  {
    Task.WaitAny(tasks.ToArray());
  }
  catch (AggregateException ex)
  {
    Trace.TraceError(ex.Message);

    // If any of the inner exceptions in the aggregate exception
    // are not cancellation exceptions then re-throw the exception.
    ex.Handle(innerEx => (innerEx is OperationCanceledException));
  }

  // If there was not a cancellation request, stop all tasks and return from Run()
  // An alternative to cancelling and returning when a task exits would be to
  // restart the task.
  if (!cts.IsCancellationRequested)
  {
    Trace.TraceInformation("Task returned without cancellation request");
    Stop(TimeSpan.FromMinutes(5));
  }
}
...
```

在该示例中，执行命令方法等待要完成的任务。如果任务被取消，运行方法假定的角色正在关闭，并等待剩下的任务完成（这在终止前等待最多 5 分钟）之前被取消。如果任务失败，由于预期异常，Run 方法将取消该任务。

> 注意： 需要注意的是，你可以实现 Run 方法更全面的监测和异常处理策略，如重新启动已失败的任务，或者包括代码，使角色停止和启动单个任务。 在以下代码中所示的停止方法时，网络控制器将关闭角色实例（它是从的 onStop 方法调用）被调用。该代码通过取消它优雅地停止每项任务。如果有任何的工作时间超过五分钟就能完成，在 Stop 方法取消处理真正地停止等待和作用被终止。

```
// Stop running tasks and wait for tasks to complete before returning
// unless the timeout expires.
private void Stop(TimeSpan timeout)
{
  Trace.TraceInformation("Stop called. Canceling tasks.");
  // Cancel running tasks.
  cts.Cancel();

  Trace.TraceInformation("Waiting for canceled tasks to finish and return");

  // Wait for all the tasks to complete before returning. Note that the
  // emulator currently allows 30 seconds and Azure allows five
  // minutes for processing to complete.
  try
  {
    Task.WaitAll(tasks.ToArray(), timeout);
  }
  catch (AggregateException ex)
  {
    Trace.TraceError(ex.Message);

    // If any of the inner exceptions in the aggregate exception
    // are not cancellation exceptions then re-throw the exception.
    ex.Handle(innerEx => (innerEx is OperationCanceledException));
  }
}
```

6

命令和查询职责分离（CQRS）模式

隔离，通过使用不同的接口，从操作读取数据更新数据的操作。这种模式可以最大限度地提高性能，可扩展性和安全性;支持系统在通过较高的灵活性，时间的演变;防止更新命令，从造成合并在域级别上的冲突。

背景和问题

在传统的数据管理系统中，这两个命令（更新数据）和查询（请求数据），针对在一个单一的数据存储库中的相同的一组实体的执行。这些实体可以是在关系数据库中的一个或多个表，如 SQL Server 的行的子集。

典型地，在这些系统中，所有的创建，读取，更新和删除（CRUD）操作被施加到该实体的相同的表示。例如，一个数据传输对象（DTO）的代表顾客从数据存储中检索由数据访问层（DAL）并显示在屏幕上。用户更新 DTO 的某些领域（也许是通过数据绑定）和 DTO，然后保存回数据存储在 DAL。相同的 DTO 同时用于读取和写入操作，如图1所示。

图1 – 一个传统的 CRUD 架构

传统的 CRUD 设计工作良好时，只有施加到数据操作有限的业务逻辑。由开发工具提供可以非常快速地创建数据访问代码的支架机构，根据需要，可再进行定制。

然而，传统的 CRUD 方法有一些缺点：

- 它往往意味着存在所述读取和写入的数据，如额外的列或属性，即使它们不是必需的作为操作的一部分，必须正确地更新的表示之间的不匹配。
- 它遇到风险的数据争用一个协作领域（在多个参与者并行运行在相同的数据集）时，记录被锁定在数据存储，或者更新冲突所造成的并发更新时，乐观锁使用。这些风险增加的复杂性和系统的吞吐量增加。此外，传统的方法也可以对性能有负面影响，由于加载的数据存储和数据访问层上，并在检索信息需要查询的复杂度。
- 它可以使安全管理和权限比较繁琐，因为每一个实体是受读取和写入操作，这可能会在不经意间暴露的数据在错误的情况下。

> 注意： 对于的 CRUD 方法的局限性有了更深的了解请参见“CRUD，只有当你能负担得起”MSDN 上。

解决方案

命令和查询职责分离（CQRS）是偏析，通过使用独立的接口读取操作的更新数据（命令）的数据（查询）的操作模式。这意味着，用于查询和更新的数据模型是不同的。该模型可随后被分离，如在图 2 中，虽然这不是绝对的要求。

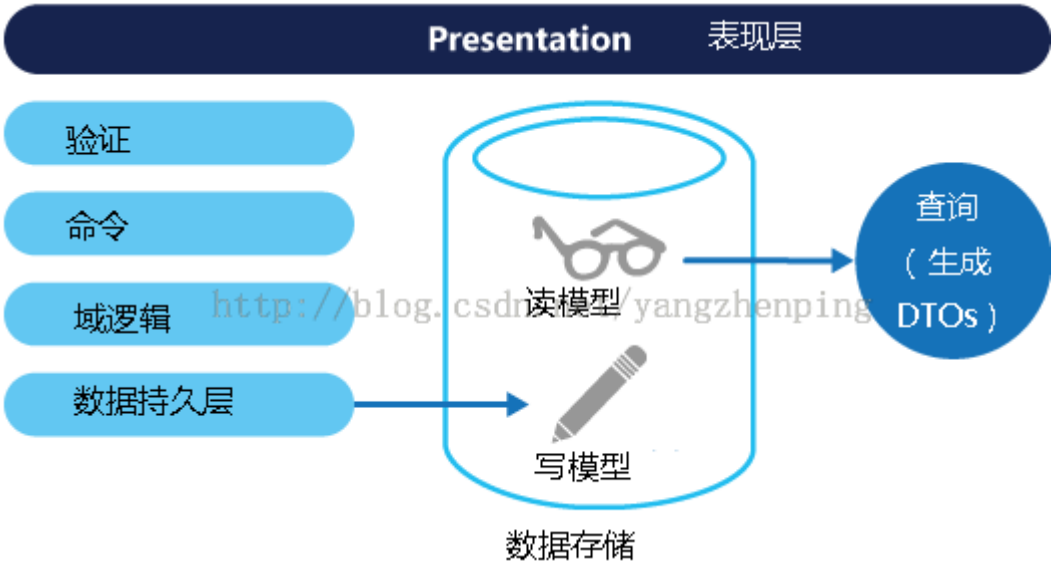


图2 – 一个基本的 CQRS 架构

相比于数据（从该开发商建立自己的概念模式）的单个模型中固有的 CRUD 为基础的系统中，使用单独的查询和更新模型中 CQRS 为基础的系统中的数据显着地简化设计和实施。然而，一个缺点是，不像 CRUD 的设计，CQRS 代码不能自动用支架的机制产生。

查询模型读取数据和写入数据可以访问相同的实体店，也许是通过使用 SQL 视图的更新模型，或产生对飞预测。但是，它是常见的数据分成不同的物理存储来提高性能，可扩展性和安全性;如图3。

图3 – 一个 CQRS 架构，具有独立读写店

所读取的存储可以是只读副本写入存储区，或读取和写入存储可以具有不同的结构完全。使用 read 店的多个只读副本可以大大提高查询性能和应用程序的UI响应速度，尤其是在分布式场景下的只读副本靠近应用程序实例。一些数据库系统，如 SQL Server，提供额外的功能，如故障转移副本，以最大限度地提高可用性。

读的分离和写入存储还允许每个到会适当缩放以匹配负载。例如，读取存储通常会遇到一个更高的负载写入存储。

当查询/读取模型中包含的非规范化的信息（见物化视图模式），性能正在读取数据的每一个视图时在应用程序中或在查询系统中的数据时最大化。

有关CQRS模式及其实现的详细信息，请参阅以下资源：

- 该模式与实践指导 CQRS 之旅 MSDN 上。尤其是你应该阅读的章节介绍了命令查询职责分离方式进行全面的探索模式，当它是有用的，这一章尾声：经验教训，了解一些，可以使用这种模式时出现的问题。
- 该职位 CQRS 由马丁 · 福勒，这也解释了该模式的基本知识，并链接到其他一些有用的资源。
- 代码更好的网站，它探讨的 CQRS 模式的许多方面对 Greg Young 的帖子。

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 分割数据存储到单独的物理存储用于读操作和写操作可以提高系统的性能和安全性，但它可以在弹性和最终一致性方面增加了相当大的复杂性。所读取的模型存储必须被更新以反映变化的写入模型存储，并且它可以是难以检测用户何时发出基于读取过时数据意味着该操作不能完成的请求。

注意

对于最终一致性的说明，请参阅数据一致性底漆。

- 考虑 CQRS 应用到你的系统中的限制部分地方将是最有价值的，并从经验中学习。
- 的典型方法拥抱最终一致性是使用事件采购与 CQRS 结合使写模式是由执行命令的驱动事件的追加只流。这些事件被用来更新充当读取模型化视图。欲了解更多信息，请参阅事件获取和 CQRS。

当使用这个模式

这种模式非常适合于：

- 其中并行地对相同的数据进行多项操作协同域。 CQRS 允许你有足够的粒度定义的命令，以尽量减少在域级别（或者不出现可以通过在命令合并的冲突）的合并冲突，更新这似乎是同一类型的数据时也是如此。
- 使用与基于任务的用户界面（其中用户通过一个复杂的过程引导作为一系列步骤），具有复杂的领域模型，以及用于团队已经熟悉领域驱动设计（DDD）技术。在写入模式有一个完整的命令处理栈与业务逻辑，输入验证和业务验证，以确保一切总是为每个聚集体（被视为一个单元进行数据变更的目的相关联的对象的每个集群相一致）中的写入模式。读出的模型没有业务逻辑或验证的堆栈，只是返回一个 DTO 在一个视图模型的使用。读出的模型与模型写入最终一致。
- 方案，其中数据的读出性能，必须分别从数据的性能进行微调写入，尤其是当读/写比是非常高的，并且当水平扩展是必要的。例如，在许多系统中的读取操作的数目是几个数量级更大的写入操作的数目。为了适应这种情况，考虑向外扩展的读取模式，但只在一个或几个实例中运行的写模式。少数写入模型实例也有助于减少合并冲突的发生。
- 场景的开发者之一的团队可以专注于复杂的领域模型，它是写模型的一部分，而另一个经验不足的团队可以专注于读模型和用户界面。
- 场景中，预计随着时间的推移，系统，并且可以包含多个版本的模型，或者业务规则经常改变。
- 与其他系统，特别是与事件采购，其中一个子系统的瞬时故障不会影响到其它的可用性的组合一体化。

这种模式可能不适合于下列情况：

- 凡域或业务规则很简单。
- 凡一个简单的 CRUD 风格的用户界面和相关的数据访问操作就足够了。
- 对于在整个系统中的实现。有一个整体的数据管理方案，其中 CQRS 可以是有用的特定组件，但是它可以增加它实际上并不需要相当大的和往往是不必要的复杂性。

事件获取和 CQRS

CQRS 模式常用于与事件获取图案一起使用。 CQRS 为基础的系统使用分离的读取和写入的数据模型，每个针对有关任务和通常位于物理上分离的存储区。当与采购活动时，事件的存储是写模式，这是信息的权威来源。一个 CQRS 为基础的系统的读取模型提供数据的物化视图，通常是高度非规范化的意见。这些视图量身定做的接口和应用程序，这有助于最大程度地显示和查询性能的显示要求。

使用事件作为写入存储区，而不是实际的数据的流，在一个时间点，避免了在单个聚合更新冲突并最大限度地提高性能和可扩展性。该事件可用于异步生成用于填充读取存储器中的数据的实体化视图。

由于事件存储是信息的权威来源，就可以删除物化视图和回放所有过去的事件来创建当前状态的一个新表示当系统升级时，或者当读取模式必须改变。物化视图是有效的数据的耐用只读缓存。

当使用 CQRS 结合事件获取模式，考虑以下几点：

- 与任何系统，其中写入和读出存储是分开的，在此基础上图案系统唯一最终一致。将有被生成的事件和数据存储器保存由这些事件被更新启动操作的结果之间有一些延迟。
- 该模式引入由于代码必须创建启动和处理事件，并组装或者更新查询或读取模型所需的适当的意见或物体额外的复杂性。在采购活动一起使用的 CQRS 模式固有的复杂性时，可以做一个成功的实现更加困难，需要重新学习的一些概念和不同的方法来设计系统。然而，事件采购可以更容易地对域进行建模，并且可以更容易地重建的观点或创建新的，因为变化的数据的意图将被保留。
- 生成物化视图中读取模型或数据通过重放和处理为特定的实体或实体的集合的事件突起的使用可能需要相当多的处理时间和资源的使用，尤其是如果它需要求和或值的数据在长时间内的，因为所有的相关联的事件可能需要被审查。这可以通过实现数据的快照在预定的时间间隔，如已经发生的特定操作的次数，或一个实体的当前状态的总计数被部分地解决。

> 注意： 欲了解更多信息，请参阅活动采购模式和物化视图模式，以及模式与实践指导 CQRS 之旅 MSDN 上。尤其是你应该阅读的章节介绍采购活动进行全面的探索模式，以及它如何与 CQRS 有用的，而章 CQRS 和 ES 深潜了解更多，包括如何聚集分区可以在微软的 Azure CQRS 使用。

例子

下面的代码显示了一个 CQRS 实现，它使用不同的定义读取和写入模型为例某些提取物。该模型的接口没有规定的基础数据存储的任何功能，并且可以发展和进行微调独立，因为这些接口是分开的。

下面的代码演示了读取的模型定义。

```
// Query interface
namespace ReadModel
{
  public interface ProductsDao
  {
    ProductDisplay FindById(int productId);
    IEnumerable<ProductDisplay> FindByName(string name);
    IEnumerable<ProductInventory> FindOutOfStockProducts();
    IEnumerable<ProductDisplay> FindRelatedProducts(int productId);
```

```
  }

  public class ProductDisplay
  {
    public int ID { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public decimal UnitPrice { get; set; }
    public bool IsOutOfStock { get; set; }
    public double UserRating { get; set; }
  }

  public class ProductInventory
  {
    public int ID { get; set; }
    public string Name { get; set; }
    public int CurrentStock { get; set; }
  }
}
```

该系统允许用户率的产品。应用程序代码通过使用在下面的代码中所示的 RateProduct 命令执行此操作。

```
<span></span><pre class="csharp" name="code">public interface Icommand
{
  Guid Id { get; }
}

public class RateProduct : Icommand
{
  public RateProduct()
  {
    this.Id = Guid.NewGuid();
  }
  public Guid Id { get; set; }
  public int ProductId { get; set; }
  public int rating { get; set; }
  public int UserId {get; set; }
}
```

本系统采用 ProductsCommandHandler 类来处理由应用程序发出的命令。客户端通常通过消息传送系统发送命令到域，如一个队列。命令处理程序接受这些命令，并调用域接口的方法。每个命令的粒度被设计成减轻冲突请求的机会。下面的代码显示了 ProductsCommandHandler 类的轮廓。

```
public class ProductsCommandHandler :
    ICommandHandler<AddNewProduct>,
```

```
    ICommandHandler<RateProduct>,
    ICommandHandler<AddToInventory>,
    ICommandHandler<ConfirmItemShipped>,
    ICommandHandler<UpdateStockFromInventoryRecount>
{
  private readonly IRepository<Product> repository;

  public ProductsCommandHandler (IRepository<Product> repository)
  {
    this.repository = repository;
  }

  void Handle (AddNewProduct command)
  {
    ...
  }

  void Handle (RateProduct command)
  {
    var product = repository.Find(command.ProductId);
    if (product != null)
    {
      product.RateProuct(command.UserId, command.rating);
      repository.Save(product);
    }
  }

  void Handle (AddToInventory command)
  {
    ...
  }

  void Handle (ConfirmItemsShipped command)
  {
    ...
  }

  void Handle (UpdateStockFromInventoryRecount command)
  {
    ...
  }
}
```

下面的代码显示了写模式 ProductsDoman 接口。

```
public interface ProductsDomain
{
  void AddNewProduct(int id, string name, string description, decimal price);
  void RateProduct(int userId int rating);
  void AddToInventory(int productId, int quantity);
  void ConfirmItemsShipped(int productId, int quantity);
  void UpdateStockFromInventoryRecount(int productId, int updatedQuantity);
}
```

还要注意如何 ProductsDomain 接口包含在域中的意义的方法。通常情况下，在一个 CRUD 环境中，这些方法将有通用名称，如保存或更新，并有一个 DTO 作为唯一的参数。该 CQRS 方法可以更好地定制，以满足该组织开展业务及库存管理的方式。

7

事件获取模式

使用仅追加存储到记录完整一系列描述在一个域上取数据，而不是存储仅仅是当前的状态，从而使存储区可以被用来实现该域对象的动作事件。该图案可以通过避免需要同步的数据模型和商业领域中简化复杂的结构域的任务;提高性能，可扩展性和响应能力;提供交易数据的一致性;并保持完整的审计跟踪和记录，可能使补偿措施。

背景和问题

大多数应用程序使用数据，并在典型的方法是应用到通过更新它作为用户使用的数据保持数据的当前状态。例如，在传统的创建，读取，更新和删除（ CRUD ）模型的典型数据处理将是从存贮器中读出的数据，进行一些修改，以使其和更新的数据的当前状态与新的值时—常常通过使用锁定数据的事务。

CRUD 方法有一定的局限性:

- 在 CRUD 系统直接执行更新操作对数据存储可能会影响性能和响应能力，并限制可扩展性，因为它需要处理开销的事实。
- 在具有许多并发用户的协作域，数据更新冲突更可能发生，因为在更新操作发生在数据的单个项目。
- 除非有另外的审核机制，它记录在一个单独的日志的每个操作的详细内容，历史记录丢失。

> 注意： 对于的 CRUD 方法的局限性有了更深的了解请参见“CRUD，只有当你能负担得起”MSDN 上。

解决方案

事件获取模式定义了一个方法来处理操作上是受一个事件序列，其中的每一个记录在仅追加存储驱动的数据。应用程序代码发送一系列命令性描述上发生的数据的情况下存储，在那里它们被持久保存的每个动作的事件。每个事件都表示一组数据更改（ 如 AddedItemToOrder ）的。

事件持久保存在一个事件存储在充当真理或记录的系统的源（ 权威数据源给定的数据元素或信息 ）有关的数据的当前状态。事件存储通常发布这些事件让消费者能够得到通知，如果需要，可以处理它们。消费者可以，例如，启动该应用中的事件的动作的其他系统的任务或执行完成操作所需的任何其他相关联的动作。注意，生成该事件的应用程序代码从订阅该事件的系统去耦。

在事件存储公布了事件的典型用途是保持实体化视图的应用程序中的行动改变他们，并与外部系统的集成。例如，系统可保持用于填充UI部分的客户订单的实体化视图。随着应用增加了新的订单，增加或订单上删除的项目，并增加了发货信息，描述这些变化可以被处理和使用的事件来更新物化视图。

此外，在任何时间点，可以对应用程序来读取事件的历史，并使用它通过有效地“回放”和消耗所有有关该实体的事件，以实现一个实体的当前状态。这可能发生在需求，以处理时的要求，或通过计划任务，使该实体的状态可以被保存为一个物化视图，以支持表示层来实现域对象。

图1示出的图案的逻辑的概述，包括一些使用事件流，例如，创建的物化视图，与外部应用程序和系统整合事件，并重放事件来创建特定实体的当前状态的突起的选项。

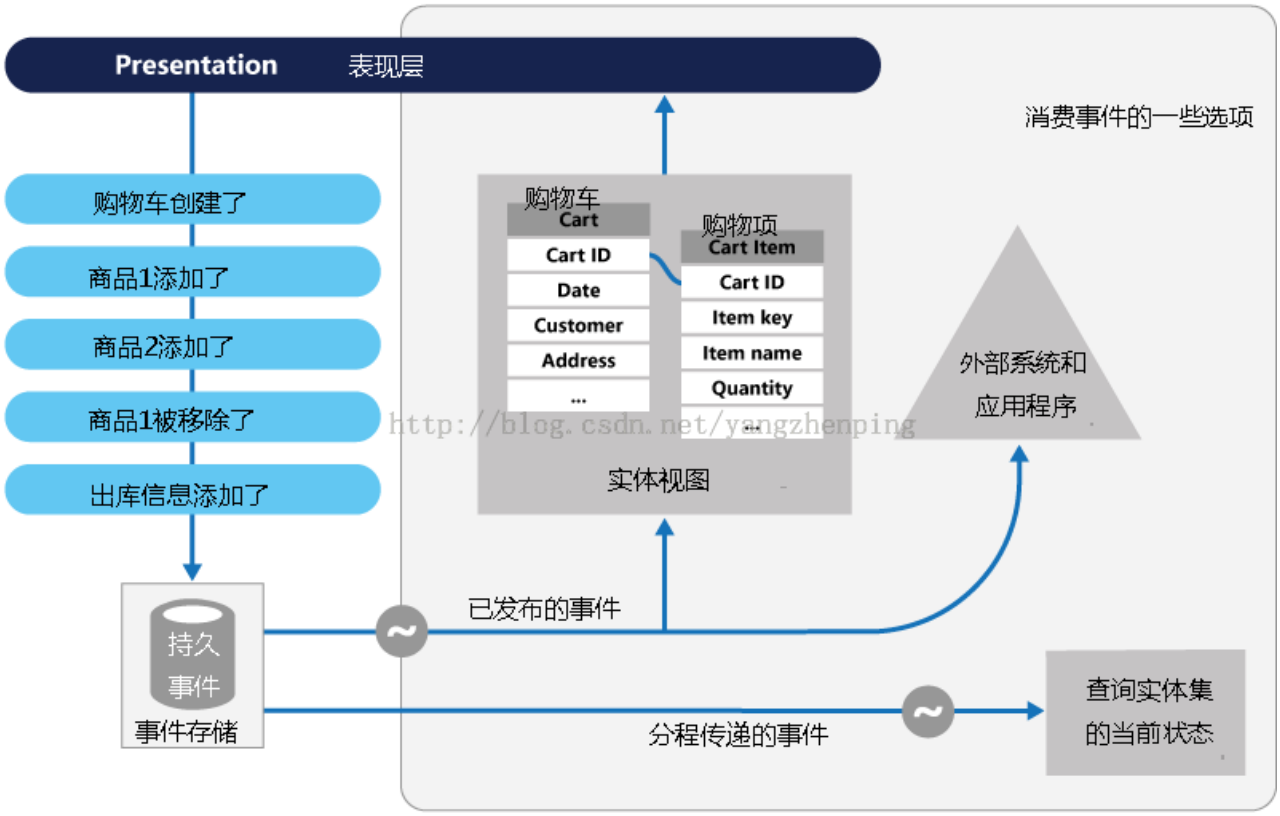


图1 – 的情况下获取模式的概述和示例

事件获取模式提供了许多优点，包括如下：

- 活动是不可变的，因此可以使用仅追加操作来保存。用户界面，工作流或过程发起产生该事件可以继续，并且处理这些事件可以在后台运行的任务的操作。此，结合的事实，有记录的执行过程中没有争用，可以极大地提高性能和可扩展性的应用，尤其是对于表示层或用户界面。
- 活动是描述所发生的一些动作，再加上描述的事件所代表的行动所需的任何相关数据的简单对象。事件不直接更新数据存储;它们被简单地记录用于处理在适当的时间。这些因素可以简化实施和管理。
- 活动通常意味着一个领域的专家，而对象关系的阻抗失配的复杂性可能意味着一个数据库表可能无法清楚地了解该领域的专家。表是表示该系统中，未发生的事件的当前状态，人工构建体。
- 事件的采购可以帮助防止引起冲突，因为它避免了要求直接更新在数据存储对象的并发更新。然而，领域模型仍然必须用来保护自己免受可能导致不一致的状态的请求。
- 事件的仅追加存储提供了可用于监测对一个数据存储所采取的行动的审核跟踪，再生的当前状态作为通过重播事件随时物化视图或预测，并协助测试和调试系统。此外，该规定使用补偿事件取消变化提供了被逆转的

变化，这不会是如果模型简单地存储在当前状态的情况下的历史记录。事件列表中，也可以用于分析应用程序的性能，并检测用户行为趋势，或获得其它有用的商业信息。

- 从响应进行操作的事件存储提出的每个事件的任何任务事件的解耦提供了灵活性和可扩展性。例如，用于处理由所述事件存储引发的事件的任务都知道只有事件的性质和它包含的数据。时所执行的任务的方式是从触发事件的动作去耦。此外，多个任务可以处理每个事件。这可能使得与其他服务和系统，只需要监听的事件存储提出了新的事件，易于集成。然而，该事件采购事件往往是非常低的水平，并且可能有必要以产生特异性整合事件来代替。

> 注意： 事件来源通常结合 CQRS 模式通过执行数据管理任务响应于所述事件，并通过物化从所存储的事件的意见。

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 创建物化视图或重放事件产生的数据的预测时，系统只会是最终一致。有一个应用程序添加事件，事件存储作为处理一个请求的结果之间有一些延迟，被公布事件，而消费者对事件的处理它们。在此期间，描述该进一步修改实体的新的事件可能已到达的情况下存储。

> 注意： 请参阅数据一致性底漆有关最终一致性的信息。

- 事件存储是信息不可变源，因此事件数据不应该被更新。要以撤销变更更新一个实体的唯一办法是补偿的事件添加到事件存储，就像你会用负数交易的会计核算。如果持久化事件的格式（而不是数据）需要改变，或者在迁移过程中，可能难以对现有的事件结合在商店的新版本。可能有必要通过所有更改的事件来循环以使它们符合新格式，或添加使用该新格式的新事件。考虑使用上的每个版本的事件模式的版本标记，以保持旧的和新的事件格式。
- 多线程应用程序和应用程序的多个实例可以被存储在事件存储事件。在事件存储事件的一致性是非常重要的，是影响到特定实体（的顺序改变为一个实体发生影响其当前状态）的事件的顺序。添加时间戳到每一个事件是一个选项，可以帮助避免出现问题。另一种常见的做法是将注释每一个事件所导致使用增量标识符的请求。如果两个动作试图为同一实体的同时添加的事件，该事件存储可以拒绝匹配现有实体标识符和事件标识符事件。
- 有没有标准的方法，还是准备建机制，如 SQL 查询，读取事件来获取信息。可以提取的唯一数据是使用事件标识符作为条件的事件流。事件ID通常映射到单个实体。一个实体的当前状态，可以仅通过重放所有涉及到它针对该实体的原始状态的事件的确定。
- 每个事件流的长度可以有上管理并更新该系统的后果。如果流很大，可以考虑创建以特定的间隔的快照，如活动指定数量。可以从该快照，通过重放该时间点之后发生的任何事件中得到的实体的当前状态。

> 注意： 有关创建数据快照的更多信息，请参见Martin Fowler的企业应用架构的网站和主从快照复制MSDN上的快照。

- 尽管采购活动减少冲突的更新数据的机会，应用程序必须仍然能够应付可能出现的通过最终一致性和缺乏交易的不一致。例如，一个事件，指示库存的库存的减少可能在数据存储到达，而对于该产品的订单被放置，从而导致需求调和这两种业务;可能是建议客户或创建后顺序。
- 事件发布可能是“至少一次”等消费者的事件，必须幂等。它们不能重新在一个事件描述，如果该事件被处理多于一次的更新。例如，如果一消费者的多个实例保持一些实体的一个属性的集合，如订单放置的总数中，只有一个必须在当一个“顺序放置”事件发生时，递增该聚合成功。虽然这不是事件来源的固有特性，它是通常的实现决策。

当使用这个模式

这种模式非常适合以下情况：

- 当你想捕捉“意图”，“目的”或“理”中的数据。例如，如动家，已关闭帐户，或已故变为一个客户实体可被捕捉为一系列特定的事件类型。
- 当它是至关重要的，尽量减少或完全避免更新冲突的发生数据。
- 当你想记录发生的事件，并能够重放它们来恢复系统的状态;用它们来回滚更改的系统;或者简单地作为一个历史和审计日志。例如，当一个任务涉及多个步骤，您可能需要执行恢复更新操作，然后重放一些步骤，使数据恢复到一致状态。
- 当使用事件是应用程序的动作的自然特征，并且需要很少的额外的开发或实现工作。
- 当您需要输入分离或应用这些行动所需的任务更新数据的过程。这可能是为了提高用户界面的性能，或分发事件到其它听众如其他应用程序或服务，必须采取某些行动的事件发生时。一个例子是有费用提交网站整合了工资制度，使响应的费用提交网站做数据的更新提出的事件存储事件由两个网站和工资系统消耗。
- 当您想要的灵活性，能够改变实体化模型和实体数据的格式，如果需求发生变化，或者，当结合使用 CQRS，你需要适应的读模式或公开数据的意见。
- 与 CQRS 一起使用时，和最终一致性是可以接受的，而读出的模型被更新，或者，在从事件流中再水化的实体和数据发生的性能的影响是可以接受的。

这种模式可能不适合于下列情况：

- 小型或简单的领域，很少或没有业务逻辑，或者非域系统，自然与传统的 CRUD 的数据管理机制，运作良好的系统。
- 哪里的一致性和实时更新数据的意见是必需的系统。

- 不要求系统中的审计跟踪，历史，功能，回滚和回放操作。
- 系统中只有非常低的冲突更新到基础数据的发生。例如，主要为添加数据而不是更新它的系统。

例子

的会议管理系统需要跟踪会议完成了预定的数量，以便它可以检查是否有座位仍然可用时，一个潜在的与会者试图做一个新的预订。该系统可以存储预定的总数为在至少两个方面的会议：

- 该系统可以存储关于预约的总数作为数据库中的该搁置预约信息的独立的实体的信息。作为预订制成或取消订单，则系统可以增加或减少该数量适当。这种方法在理论上是简单的，但可引起可扩展性问题，如果有大量的与会者试图进行预订时的短时间内座位。例如，在最后一天或预约期间闭合，以便之前。
- 该系统可以存储大约预订和取消如在事件存储举办的活动信息。然后，它可以计算出可用通过重播这些事件的席位数。这种方法可以更加扩展性由于事件的不变性。该系统只需要能够从事件存储读取数据，或将数据追加到该事件存储。关于预订和取消事件信息不会被修改。

图2显示了如何将会议管理系统的座位子系统可能通过采购活动来实现。

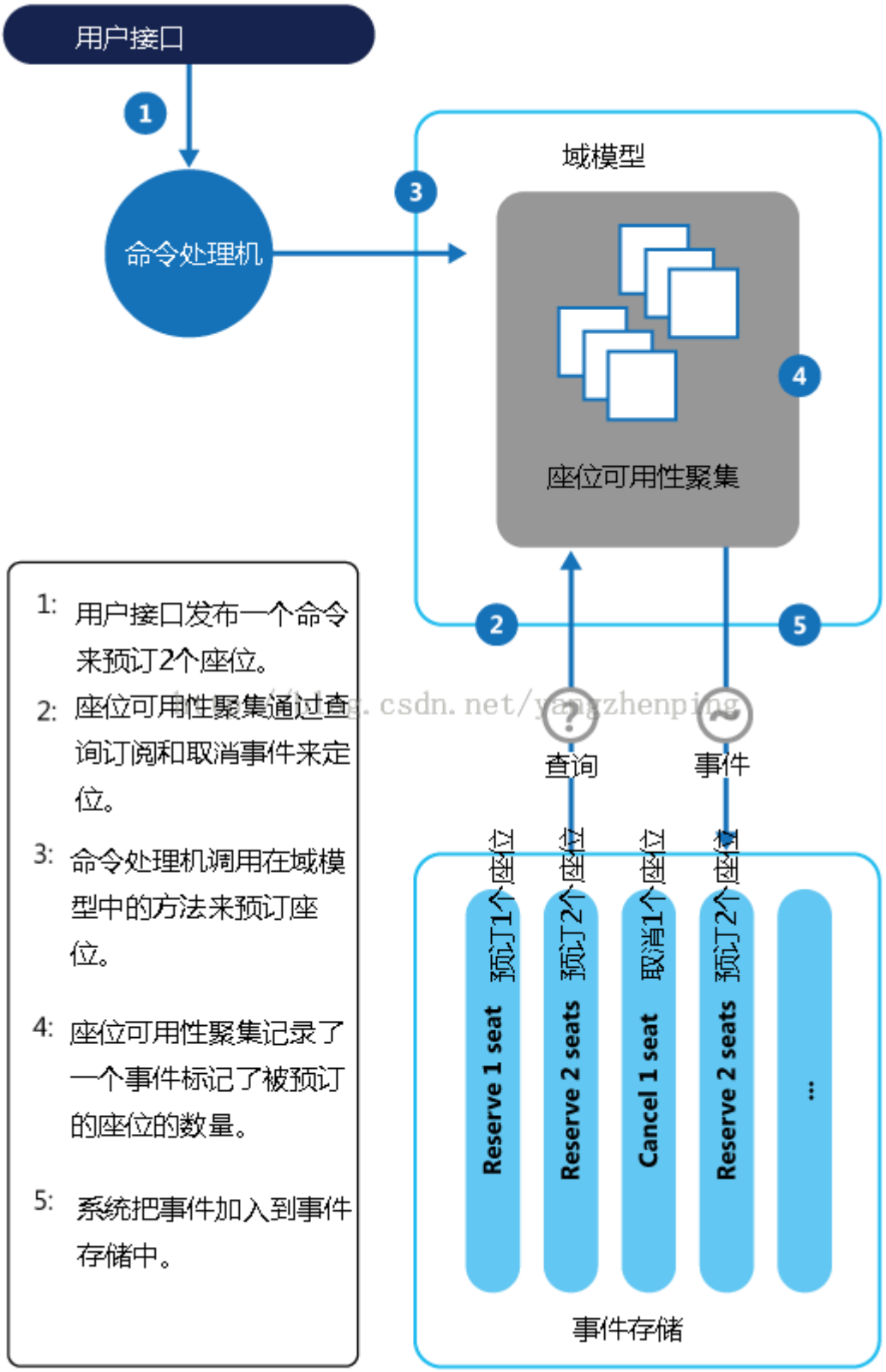


图2 – 使用事件来源获取关于座位预订信息，在会议管理系统

行动预留两个座位的顺序如下：

1.用户界面发出命令，以预留座位有两个人参加。该命令是由一个单独的命令处理程序（一块逻辑的是从用户界面分离，并负责张贴的命令处理请求）处理。 包含所有预订的会议信息

2.一种聚集通过查询描述预订和取消的事件构成。该集合体被称为 SeatAvailability，并且被包含在暴露在聚合查询和修改的数据的方法的域模型。

> 注意： 一些优化利用快照（这样就不需要查询和重放事件的完整列表，以获得聚集的当前状态），并维持聚合的在存储器中的高速缓存副本来考虑。

3，命令处理程序调用的域模型曝光，使保留的方法。

4.SeatAvailability 骨料记录包含的席位被保留数量的事件。聚合应用事件接下来的时间，所有的预订将被用来计算的座位有多少仍然存在。

5.系统追加了新的事件在事件存储的事件列表。

如果一个用户希望取消一个座位，该系统遵循不同的是，命令处理程序发出，其生成一个座位取消事件，并将其追加到事件存储区中的命令类似的过程

以及可扩展性提供更大的空间，使用事件店内还提供了一个完整的历史，或审计追踪，预订和取消了会议。记录在事件存储在事件真相的权威和唯一来源。没有必要持续聚集体中的任何其他方法，因为该系统可以很容易地重放这些事件和恢复状态，以任意的时间点。

8

外部配置存储模式

移动配置信息从应用部署包到一个集中位置。这个模式可以提供机会，以便管理和配置数据的控制，以及用于跨应用程序和应用程序实例共享的配置数据。

背景和问题

大多数应用程序运行时环境包括位于应用程序文件夹内的在部署应用程序文件保持配置信息。在某些情况下也能够编辑这些文件来改变该应用程序的行为，它已经被部署之后。然而，在许多情况下，改变配置所需要的应用程序被重新部署，从而导致不可接受的停机时间和额外的管理开销。

本地配置文件还配置限制为单个应用程序，而在某些情况下将是有用的，以在多个应用程序之间共享的配置设置。例子包括数据库连接字符串，UI 主题的信息，或队列和存储所使用的一组相关的应用程序的URL。

变更管理跨应用程序的多个运行实例的本地配置，尤其是在云托管的情况，也可能是具有挑战性的。它可能会导致使用不同的配置设置的实例，而更新正被部署。

另外，更新应用程序和组件可能需要更改的配置方案。许多配置系统不支持不同版本的配置信息。

解决方案

存储在外部存储器中的配置信息，并提供可用于快速和有效地读取和更新的配置设置的接口。外部存储的类型取决于应用程序的主机和运行时环境。在一个云托管的情况下它是一个典型的基于云的存储服务，但可能是一个托管数据库或其他系统。

选择用于配置信息的备份存储应通过适当的接口，它提供了一个可控制的方式，使回用保持一致和易于使用的访问被朝向。理想情况下，它应该公开在键入正确，结构化的格式的信息。的实施也可能需要对用户进行授权“，以保护结构的数据访问，并且具有足够的灵活性，以允许要被存储的多个版本的配置（例如，开发，分段，或生产，并且每一个的多个发行版本）。

> 注意： 许多内置的系统配置中读取数据时，应用程序启动和高速缓存内存中的数据提供快速访问，并尽量减少对应用程序性能的影响。根据所使用的后备存储器的类型，以及该商店的等待时间，这可能是有利的，以实现外部配置存储器内的高速缓存机制。有关实现缓存的详细信息，请参阅缓存指导。

图1示出了本模式的概述。

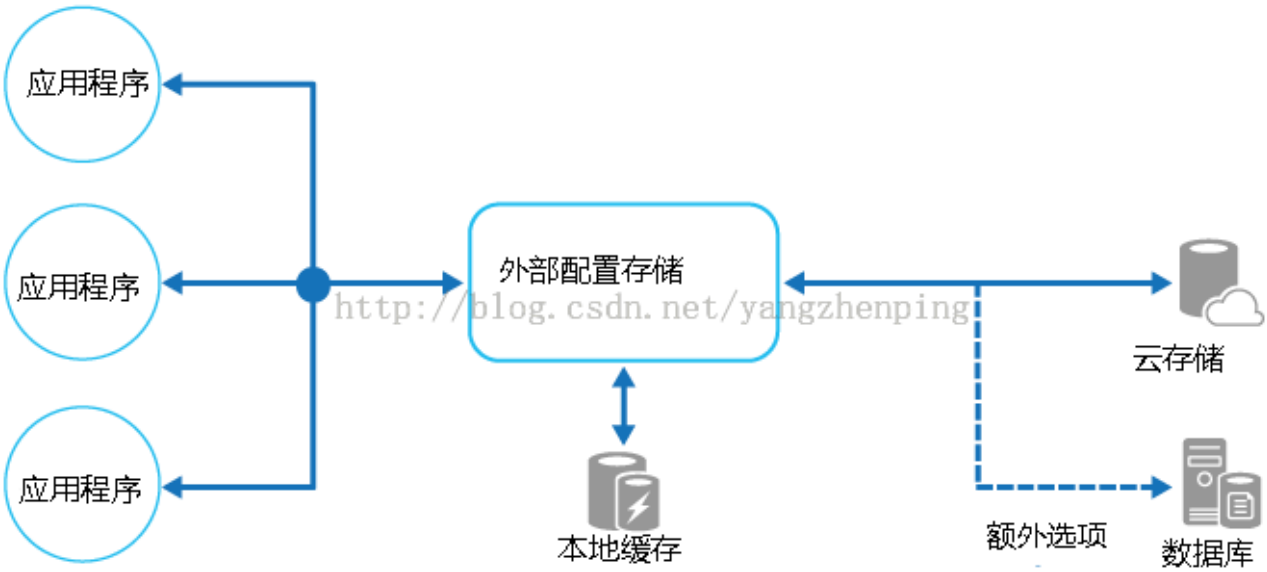


图1 – 外部配置存储模式可选本地缓存概述

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 选择一个后备存储，提供可接受的性能，高可用性，健壮性和可备份作为应用程序的维护和管理过程的一部分。在一个云托管的应用程序，使用云存储的机制通常是一个不错的选择，以满足这些要求。
- 设计的后备存储的架构允许在信息能够保存类型的灵活性。确保它提供了一种使用它可以要求该申请的所有配置的要求，例如输入数据中，设置的集合，多个版本的设置，以及任何其他功能。该模式应该是易于扩展的需求，以支持更多的设置更改。
- 考虑后备存储的物理性能，它与配置信息的存储方式，以及对性能的影响。例如，存储一个包含XML文件的配置信息将要求使用配置界面或应用程序解析该文件以读取各个设置，将使得更新的设置更加复杂，尽管高速缓存中的设置可有助于抵消较慢的读取性能。
- 考虑如何配置界面将允许配置设置的范围和继承的控制权。例如，它可能是一个要求的范围的配置设置在组织，应用程序和设备的水平;支持在访问不同范围的控制下放;并且，以防止或允许单独的应用程序，以覆盖设置。
- 确保配置界面可以在需要的格式的配置数据暴露，如输入值的集合，键/值对，或财产包。然而，考虑能力和API的复杂性之间的平衡，以使其有用的，但尽可能地易于使用。
- 考虑配置存储界面将如何表现时，设定有误，或没有在内部存储存在。它可能是适当的，返回默认设置和记录错误。也可以考虑，如配置设置按键或者名称，二进制数据的存储和处理，以及null或空值处理方式的情况下，灵敏度方面。

- 考虑如何将保护配置数据仅允许访问相应的用户和应用程序。这很可能是在配置存储器接口的一个特征，但它也是必要的，以确保在后备存储器中的数据不能被直接访问，而不适当的权限。确保读取和写入配置数据所需的权限之间的严格分离。也可以考虑是否需要加密部分的配置设置或全部，以及如何将配置存储接口中实现。
- 请记住，集中存储配置，这在运行时改变应用程序的行为，是非常重要的，并应部署，更新，并使用相同的机制，部署应用程序代码进行管理。例如，可能会影响多个应用程序的更改，必须进行使用一个完整的测试和分阶段部署的方式，以确保变化是适用于所有使用该配置的应用程序。如果管理员简单地进行编辑的设置来更新一个应用程序，它可以产生不利使用相同设置的其他应用程序的影响。
- 如果应用程序的高速缓存的配置信息，应用程序可能需要的，如果配置更改被提醒。有可能实现的过期策略在缓存中的配置数据，使这些信息被自动定期刷新任何更改拿起（和付诸行动）。本指南中其他地方所描述的运行模式重构可能有关您的方案。 何时使用这个模式

这种模式非常适合于：

- 被多个应用程序和应用程序实例，或在标准配置中，必须跨多个应用程序和应用程序实例执行之间共享配置设置。
- 在标准配置的系统不支持所有所需的配置设置，如存储图像或复杂的数据类型。
- 作为补充商店的一些应用程序的设置，或许允许应用程序重写一些集中存储或所有设置。
- 作为一种机制，通过记录的部分或全部类型的访问来配置存储监控使用的配置设置简化了多个应用程序管理，以及可选。

例子

在微软 Azure 托管应用，用于从外部存储配置信息的典型的选择是使用 Azure 存储。这是有弹性的，提供高性能，并重复 3 次自动故障切换提供高可用性。 Azure 的表格提供了一个键/值存储与使用一个灵活的架构的价值的能力。 Azure 的 Blob 存储提供了一个分层的基于容器的存储，可以保存任何类型的单独命名的 blob 数据。

下面的示例显示了如何配置存储可以通过 Azure 的 Blob 存储来实现存储和揭露的配置信息。该BlobSettingsStore 类文摘 Blob 存储用于保存配置信息，并实现在下面的代码所示 ISettingsStore 接口。

> 注意： 此代码在 ExternalConfigurationStore 解决方案 ExternalConfigurationStore.Cloud 项目提供。该解决方案可用于下载本指导意见。

```
public interface IsettingsStore
{
 string Version { get; }
```

```
  Dictionary<string, string> FindAll();

  void Update(string key, string value);
}
```

该接口定义的方法，用于检索和更新在配置存储中保持的配置设置，并且包括可用于检测是否有任何配置设置最近已修改的版本号。何时配置设置被更新时，版本号的变化。该 BlobSettingsStore 类使用 BLOB 的 ETag 的属性来实现的版本。一个 blob 的 ETag 的属性将 BLOB 写入每一次自动更新。

注意

需要注意的是，按照设计，这个简单的解决方案，展现了所有的配置设置为字符串值，而不是类型的值。 该 ExternalConfigurationManager 类提供了围绕 BlobSettingsStore 物体的包装。应用程序可以使用这个类来存储和检索配置信息。这个类使用 Microsoft 无扩展库来揭露过的 IObservable 接口的实现做出任何配置更改。如果设置是通过调用SetAppSetting法修改，更改的事件引发，所有订阅者此事件将被通报。 请注意，所有的设置也缓存到 ExternalConfigurationManager 类快速访问内部 Dictionary 对象。该 SetAppSetting 方法更新该高速缓存中，并且该应用程序可以使用以检索配置设置的 GetSetting 方法从高速缓存中读取数据（如果未在该高速缓存中找到该设置，它从 BlobSettingsStore 对象检索代替）。

所述的 getSettings 方法调用 CheckForConfigurationChanges 的方法来检测在 Blob 存储的配置信息是否通过检查版本号，并将它与所述 ExternalConfigurationManager 对象保持当前的版本号进行比较已经改变。如果一个或多个已经发生了变化，改变的事件引发，并缓存在 Dictionary 对象的配置设置被刷新。这是缓存除了图案的应用。

下面的代码示例演示如何更改的情况下，SetAppSettings 方法，该方法的 getSettings 和 CheckForConfigurationChanges 方法实现

```
public class ExternalConfigurationManager : IDisposable
{
  // An abstraction of the configuration store.
  private readonly ISettingsStore settings;
  private readonly ISubject<KeyValuePair<string, string>> changed;
  ...
  private Dictionary<string, string> settingsCache;
  private string currentVersion;
  ...
  public ExternalConfigurationManager(ISettingsStore settings, ...)
  {
    this.settings = settings;
    ...
  }
  ...
  public IObservable<KeyValuePair<string, string>> Changed
```

```
{
  get { return this.changed.AsObservable(); }
}
...
public void SetAppSetting(string key, string value)
{
  ...
  // Update the setting in the store.
  this.settings.Update(key, value);

  // Publish the event.
  this.Changed.OnNext(
      new KeyValuePair<string, string>(key, value));

  // Refresh the settings cache.
  this.CheckForConfigurationChanges();
}

public string GetAppSetting(string key)
{
  ...
  // Try to get the value from the settings cache.
  // If there is a miss, get the setting from the settings store.
  string value;
  if (this.settingsCache.TryGetValue(key, out value))
  {
    return value;
  }

  // Check for changes and refresh the cache.
  this.CheckForConfigurationChanges();

  return this.settingsCache[key];
}
...
private void CheckForConfigurationChanges()
{
  try
  {

    // Assume that updates are infrequent. Lock to avoid
    // race conditions when refreshing the cache.
    lock (this.settingsSyncObject)
    {        {
      var latestVersion = this.settings.Version;
```

```
      // If the versions differ, the configuration has changed.
      if (this.currentVersion != latestVersion)
      {
        // Get the latest settings from the settings store and publish the changes.
        var latestSettings = this.settings.FindAll();
        latestSettings.Except(this.settingsCache).ToList().ForEach(
                        kv => this.changed.OnNext(kv));

        // Update the current version.
        this.currentVersion = latestVersion;

        // Refresh settings cache.
        this.settingsCache = latestSettings;
      }
    }
  }
  catch (Exception ex)
  {
    this.changed.OnError(ex);
  }
 }
}
```

注意

该 ExternalConfigurationManager 类还提供了一个名为 Environment 属性。此属性的目的是为了支持不同的配置为在不同的环境中，如临时和生产运行的应用程序。

一个 ExternalConfigurationManager 对象也可以定期查询 BlobSettingsStore 对象的任何变化（通过使用定时器）。该 StartMonitor 和 StopMonitor 方法如下图所示的启动代码示例和停止计时器。该 OnTimerElapsed 方法当定时器到期时，并调用 CheckForConfigurationChanges 方法来检测的任何变化，并提高了变更的情况下，如前面所描述运行。

```
public class ExternalConfigurationManager : IDisposable
{
 ...
 private readonly ISubject<KeyValuePair<string, string>> changed;
 private readonly Timer timer;
 private ISettingsStore settings;
 ...
 public ExternalConfigurationManager(ISettingsStore settings,
                                     TimeSpan interval, ...)
 {
  ...
```

```
  // Set up the timer.
  this.timer = new Timer(interval.TotalMilliseconds)
  {
    AutoReset = false;
  };
  this.timer.Elapsed += this.OnTimerElapsed;

  this.changed = new Subject<KeyValuePair<string, string>>();
  ...
}

...

public void StartMonitor()
{
  if (this.timer.Enabled)
  {
    return;
  }

  lock (this.timerSyncObject)
  {
    if (this.timer.Enabled)
    {
      return;
    }
    this.keepMonitoring = true;

    // Load the local settings cache.
    this.CheckForConfigurationChanges();

    this.timer.Start();
  }
}

public void StopMonitor()
{
  lock (this.timerSyncObject)
  {
    this.keepMonitoring = false;
    this.timer.Stop();
  }
}
```

```
  private void OnTimerElapsed(object sender, EventArgs e)
  {
    Trace.TraceInformation(
          "Configuration Manager: checking for configuration changes.");

    try
    {
      this.CheckForConfigurationChanges();
    }
    finally
    {
      ...
      // Restart the timer after each interval.
      this.timer.Start();
      ...
    }
  }
  ...
}
```

该 ExternalConfigurationManager 类被实例化作为由 ExternalConfiguration 类如下所示的单一实例。

```
public static class ExternalConfiguration
{
  private static readonly Lazy<ExternalConfigurationManager> configuredInstance
                        = new Lazy<ExternalConfigurationManager>(
    () =>
    {
      var environment = CloudConfigurationManager.GetSetting("environment");
      return new ExternalConfigurationManager(environment);
    });

  public static ExternalConfigurationManager Instance
  {
    get { return configuredInstance.Value; }
  }
}
```

下面的代码取自 WorkerRole 类中 ExternalConfigurationStore.Cloud 项目。它显示了如何在应用程序使用 ExternalConfiguration 类读取和更新设置。

```
public override void Run()
{
  // Start monitoring for configuration changes.
  ExternalConfiguration.Instance.StartMonitor();
```

```
  // Get a setting.
  var setting = ExternalConfiguration.Instance.GetAppSetting("setting1");
  Trace.TraceInformation("Worker Role: Get setting1, value: " + setting);

  Thread.Sleep(TimeSpan.FromSeconds(10));

  // Update a setting.
  Trace.TraceInformation("Worker Role: Updating configuration");
  ExternalConfiguration.Instance.SetAppSetting("setting1", "new value");

  this.completeEvent.WaitOne();
}
```

下面的代码，也是从 WorkerRole 类，展示了如何应用订阅配置事件。

```
public override bool OnStart()
{
  ...
  // Subscribe to the event.
  ExternalConfiguration.Instance.Changed.Subscribe(
    m => Trace.TraceInformation("Configuration has changed. Key:{0} Value:{1}",
        m.Key, m.Value),
    ex => Trace.TraceError("Error detected: " + ex.Message));
  ...
}
```

9

联合身份模式

验证委托给外部身份提供者。这种模式可以简化开发，最大限度地减少对用户管理的要求，并提高了应用程序的用户体验。

背景和问题

用户通常需要使用由提供，并通过与它们有商业关系的不同组织主持的多个应用程序一起工作。但是，这些用户可能被迫使用特定的（和不同的）的凭证，每一个。这可以：

- 原因脱节的用户体验。用户经常忘记登录凭据时，他们有很多不同的的。
- 暴露安全漏洞。当用户离开公司的帐户，必须立即取消设置。这是很容易忽略这在大型组织中。
- 复杂的用户管理。管理员必须管理凭据的所有用户，以及执行等方面提供密码提示的其他任务。

用户会相反，通常期望使用相同的凭证用于这些应用。

解决方案

实现了可以使用联合身份的认证机制。从应用程序代码中分离的用户身份验证和身份验证委派到受信任的身份提供者，可以大大简化开发，让用户使用更广泛的身份提供者（国内流离失所者），同时最大限度地减少管理开销进行身份验证。它也可以让你清楚地分离的授权认证。

可信身份提供者可能包括公司目录，内部部署联合身份验证服务，其他安全令牌服务（STS的）业务合作伙伴提供的，或社会身份提供者可以验证谁拥有用户，例如，微软，谷歌，雅虎或Facebook帐户。

图1示出了当客户端应用程序需要访问要求身份验证的服务的联合身份模式的原理。该认证是通过身份提供者（IDP），在演唱其工作与安全令牌服务（STS）的执行。境内流离失所者问题的主张有关身份验证的用户的信息安全令牌。该信息被称为权利要求中，包括用户的身份，并且还可以包括其他信息，例如角色成员和更细粒度的访问权限。

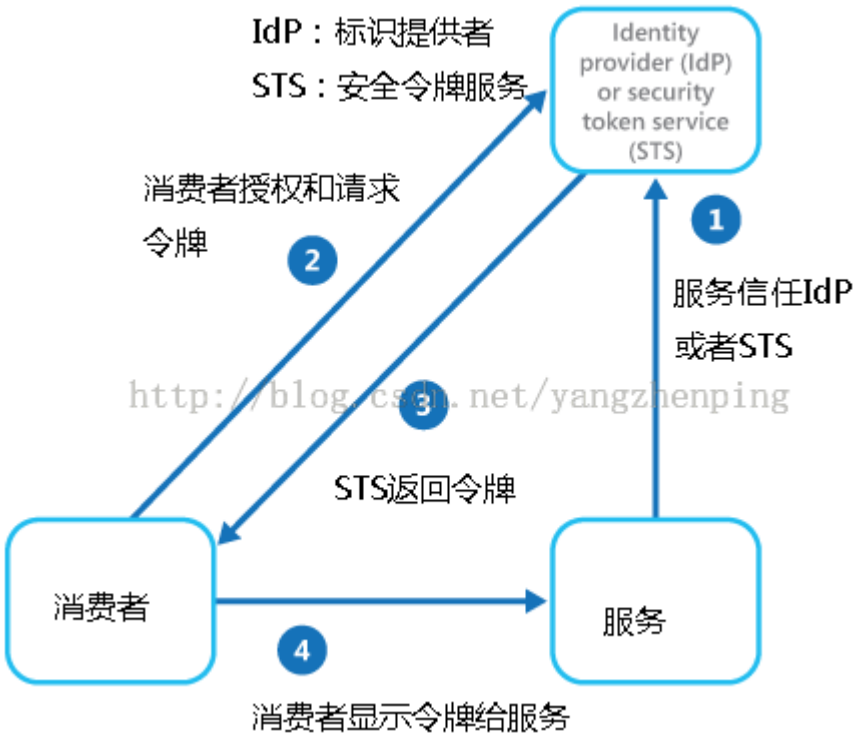


图1 – 联合身份验证概述

该模型通常被称为基于声明的访问控制。应用程序和服务授权访问基于包含在令牌中的权利要求的特征和功能。要求身份验证必须相信国内流离失所者的服务。客户端应用程序的联系人执行身份验证境内流离失所者。如果认证成功，则的 IdP 返回包含用于识别用户于 STS 的权利要求书的令牌（注意的 IdP 和 STS 可以是相同的服务）。在 STS 可以改变和增大中根据预定义的规则，令牌中的权利要求书，将其返回到客户端之前。然后，客户端应用程序可以将此令牌传递给服务作为其身份证明。

> 注意 在某些情况下可能会有额外的 STS 的信任链。例如，在微软 Azure 的场景描述后，内部部署 STS 信任 STS 另一个是负责访问的身份提供者对用户进行认证。这种方法是在企业的情况普遍，其中有一个本地 STS 和目录。

联合身份验证提供了一个基于标准的解决方案，在不同信任域身份的问题，并且可以支持单点登录。它正在成为在所有类型的应用，特别是云托管的应用越来越普遍，因为它支持上，而不需要直接网络连接到身份提供单点登录。用户不必输入凭据为每一种应用。这增加了安全性，因为它阻止了访问许多不同的应用程序所需的凭据的扩散，同时也隐藏了用户的凭据所有，但原来的身份提供者。应用程序只看到包含的令牌中的身份验证信息。

联合身份也具有重大的优点，即人的身份和凭证管理是身份提供者的责任。应用程序或服务并不需要提供身份管理功能。另外，在企业环境中，企业目录不需要知道关于用户（提供它信任的身份提供者），它去除了管理该目录中的用户身份的所有的管理开销。

问题和注意事项

设计实现联合身份验证的应用程序时考虑以下因素：

- 身份验证可以是一个单点故障。如果您将应用程序部署到多个数据中心，考虑部署身份管理机制，以相同的数据中心，以保持应用程序的可靠性和可用性。
- 身份验证机制，可以提供工具来配置基于包含在认证令牌的作用索赔的访问控制。这通常被称为基于角色的访问控制（RBAC），并且它可以允许控制权访问的功能和资源的更精细的水平。
- 与企业目录，利用社会身份提供者通常不提供有关身份验证的用户以外的电子邮件地址，也许名称的信息基于声明的身份。一些社会身份提供者，如 Microsoft 帐户，只提供一个唯一的标识符。应用程序通常将需要保持对注册用户的一些信息，并且能够匹配该信息，包含在令牌中的权利要求书的标识符。典型地，这是通过一个注册过程完成的用户第一次访问该应用程序时，信息然后被注入到令牌作为每个认证后附加的权利要求。
- 如果配置为 STS 多个身份提供者，它必须检测其身份提供者，用户应该被重定向到身份验证。这个过程被称为主领域发现。在 STS 可能能够基于电子邮件地址或用户名，该用户提供，该用户被访问时，用户的 IP 地址范围，该应用程序的一个子域，或上的 cookie 中存储的用户的内容自动地执行此浏览器。例如，如果用户在微软域，如user@live.com 输入一个电子邮件地址，在 STS 将用户重定向到 Microsoft 帐户登录页面。在随后的访问中，STS可以使用 cookie 来表示最后登录用的 Microsoft 帐户。如果自动发现无法确定主领域时，STS 将显示一个家庭领域发现（HRD）页，其中列出了受信任的身份提供者，用户必须选择他们想要使用的人。 何时使用这个模式

此模式是非常适合的范围内的情况下，如：

- 在企业单点登录。在这种情况下，您需要验证员工被托管在云中的企业安全边界以外的企业应用程序，而不需要他们每次访问应用程序时签署。用户体验是一样的使用本地应用程序，他们签约到公司网络时，最初通过身份验证的时候，并从此获得所有相关的应用程序，而无需再次登录。
- 与多个合作伙伴联合身份。在这种情况下，您需要验证这两个企业的员工和业务合作伙伴谁没有在公司目录帐户。这是企业对企业（B2B）的应用程序，集成与第三方服务，并在那里与不同的IT系统公司合并或共享资源的应用普遍。
- 在SaaS应用联合身份。在这种情况下独立软件供应商（ISV）提供了一个即用型服务，为多个客户或租户。每个租户将要使用适当的身份提供者进行身份验证。例如，企业用户会希望我们自己的企业资格证书，而租户的消费者和客户可能希望使用自己的社会身份凭证。

这种模式可能不适合于下列情况：

- 应用程序的所有用户都可以通过一个标识提供者进行身份验证，并没有要求使用任何其他身份提供者进行身份验证。这是典型的只使用企业目录进行身份验证业务应用，并获得该目录可在应用程序中直接使用VPN，或（在云中托管的情况下），通过导通之间的虚拟网络连接本地目录和应用程序。
- 应用程序最初建使用不同的认证机制，或许与自定义用户存储，或者不具有处理所用的权利要求为基础的技术的协商标准的能力。改造基于声明的身份验证和访问控制到现有的应用程序可能很复杂，可能不符合成本效益。

例子

组织举办了多租户软件即在 Azure 中的服务（SaaS）应用程序。该应用程序 incudes 一个网站，租户可以用它来管理应用程序为自己的用户。该应用程序允许租户使用由活动目录联合服务（ADFS）产生的，当用户通过该组织自己的 Active Directory 身份验证的联合身份访问租户的网站。图2示出了该过程的概述。

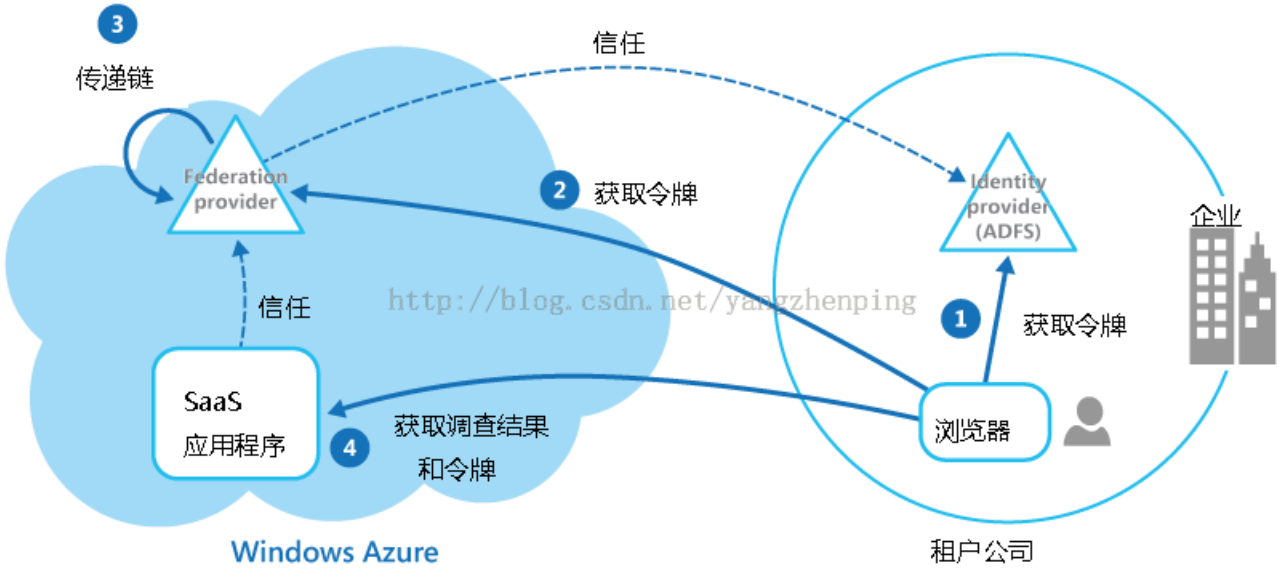


图2 – 用户如何在大型企业用户访问应用程序

在图 2 所示的场景中，商户验证自己的身份提供者（步骤1），在这种情况下 ADFS。在成功验证租客，ADFS 发出的令牌。客户端浏览器转发此令牌至 SaaS 应用的联合提供者，其信任的租户的 ADFS 发出令牌，以便取回的令牌是有效的 SaaS 的联合提供者（步骤 2）。如果有必要，在 SaaS 联合会提供商上执行令牌中的权利要求书的权利要求到该应用程序识别的新令牌返回给客户机浏览器之前（步骤3）的变换。应用程序信任的 SaaS 的联合提供者发出的令牌，并使用在令牌中的权利要求书申请授权规则（步骤 4）。

租户将不再需要记住不同的凭据来访问应用程序，以及管理员租户的公司将能够在自己的 ADFS 配置可以访问应用程序的用户的列表。

10

守门员模式

通过使用充当客户端和应用程序或服务之间的代理，验证和进行消毒的请求，并将它们之间的请求和数据的专用主机实例保护的应用程序和服务。这可以提供一个额外的安全层，并限制了系统的攻击面。

背景和问题

应用程序通过接受和处理请求揭露它们的功能提供给客户。在云托管方案，应用程序暴露终端客户机连接，一般包括代码来处理来自客户端的请求。此代码可以执行认证和验证，一些或所有请求的处理，并有可能访问存储等服务代表客户端的。

如果恶意用户能够危及系统和访问应用程序的托管环境，它使用安全机制，诸如凭证和存储密钥，并且该服务并访问数据，被暴露。因此，恶意用户可能能够获得无节制访问敏感信息和其他服务。

解决方案

为了尽量减少接触到敏感信息和服务客户的风险，去耦，揭露出从处理请求并访问存储在代码公共端点的主机或任务。这可以通过使用一个立面或专用任务，与客户端进行交互，然后手拿开的请求（可能通过一个去耦接口）连接到主机或任务将要处理的请求来实现。图1示出了这种方法的一个高层视图。

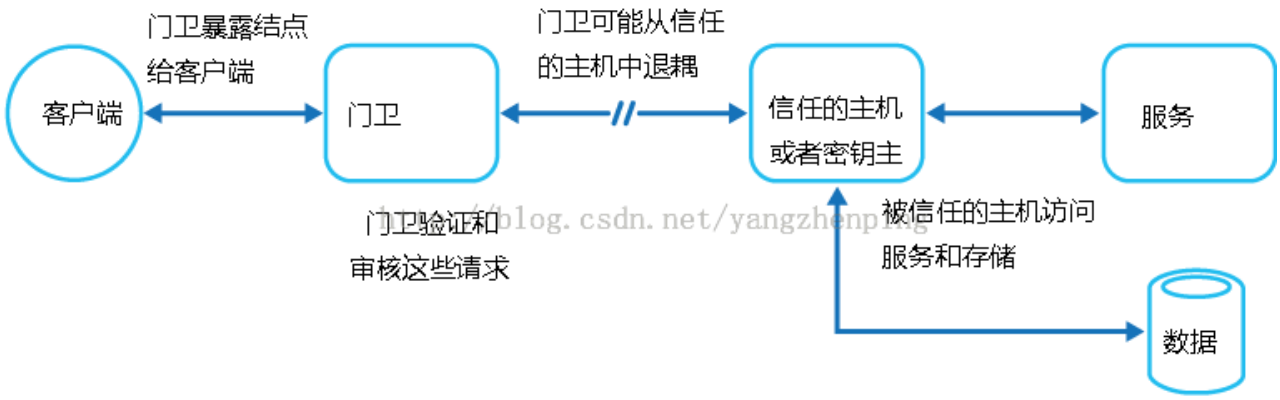


图1 – 这种模式的高级概述

守门员模式可以简单地用来保护存储，或者它可被用作一个更全面的立面，以保护所有的应用程序的功能。的重要因素是：

- 控制验证。守门员验证所有请求，并拒绝那些不符合验证要求。
- 有限的风险和曝光。守门员不具有访问所使用的可信主机访问存储和服务的凭证或密钥。如果关守被攻破，攻击者无法获得访问这些凭据或密钥。
- 适当的安全性。守门员运行在一个有限的特权模式，而应用程序的其余部分在访问存储和服务所需要的完全信任模式下运行。如果关守被破坏，它不能直接访问应用程序的服务或数据。

此图案有效地作用就像一个防火墙在一个典型的网络拓扑。它允许关守来检查请求并做出关于是否将请求传递到可信主机决定（有时也被称为钥匙之王），执行所需的任务。这一决定通常需要守门员来验证并将其传递到受信任主机前消毒要求的内容。

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 确保受信任主机到网守请求通过仅暴露内部或保护端点，只有连接到守门员。受信任主机不应该暴露任何外部端点或接口。
- 关守必须在有限的特权模式下运行。通常，这意味着运行守门员和独立的托管服务或虚拟机的可信主机。
- 关守不应该执行相关的应用程序或服务，或访问任何数据的任何处理。它的功能是纯粹的验证和消毒要求。受信任的主机可能需要执行的请求额外的验证，但核心的验证应该由守门员进行。
- 使用守门员和信任的主机或任务，如果这是可能的之间的安全通信通道（HTTPS，SSL或TLS）。然而，一些托管环境可能不支持HTTPS内部端点。
- 添加额外的层，以实现守门员模式的应用有可能对应用程序的性能造成一定影响，由于它需要额外的处理和网络通信。
- 关守实例可能是一个单点故障。为了最大限度地减少故障的影响，考虑部署其他实例，并使用自动缩放机制，以确保有足够的能力来保持可用性。

何时使用这个模式

这种模式非常适合于：

- 应用程序，处理敏感信息，揭露必须具有高一定程度的保护免受恶意攻击，或执行不得破坏关键业务服务。
- 分布式应用中，有必要从主要任务分别执行请求验证，或集中此验证，以简化维护和管理。

例子

在一个云托管的情况下，该模式可以通过使用一个内部端点，一个队列，或存储作为中间通信机制解耦从受信任的角色和服务应用程序中的关守角色或虚拟机来实现。图 2 示出了使用内部的端点时的基本原则。

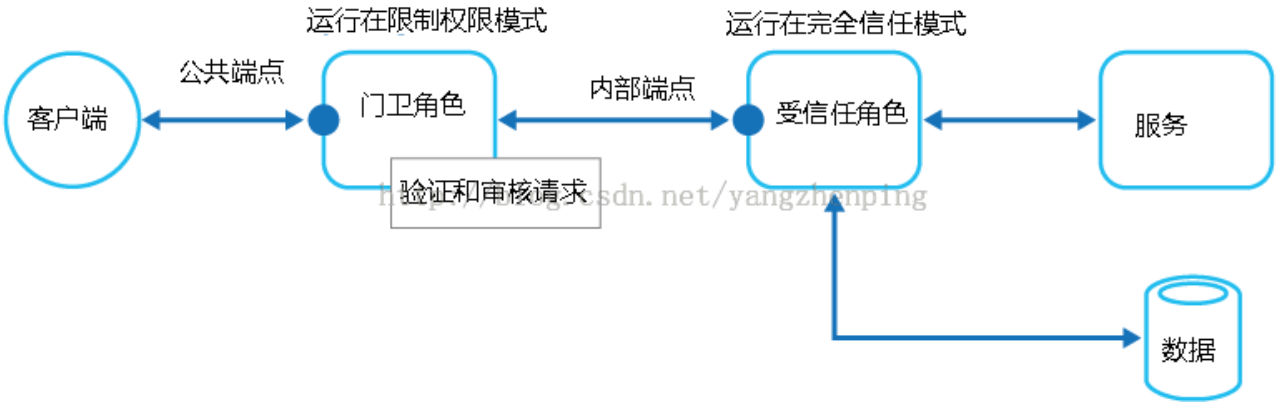


图 2 – 的模式使用云服务的网络和辅助角色的一个例子

11

健康端点监控模式

实施外部工具可以定期通过暴露终端访问应用程序中的功能检查。这个模式可以帮助验证的应用和服务被正确执行。

背景和问题

它是很好的做法，并且通常是一个业务需求，并监控web应用程序，和中间层和共享服务，以确保它们是可用的，并执行正确的。然而，它更难以监测在云中运行比它要监控本地服务的服务。举例来说，你不必完全控制主机环境，而服务通常依赖于平台，供应商和其他公司提供其他服务。

也有一些影响云托管的应用，如网络延迟，性能和下面的计算和存储系统的可用性，以及它们之间的网络带宽的因素很多。由于任何这些因素的服务可能完全或部分失败。因此，您必须定期验证服务正在执行正确，以确保可用性，这可能是您的服务级别协议（SLA）的一部分所要求的水平。

解决方案

通过将请求发送到应用程序的端点实施健康监测。该应用程序应该执行必要的检查，并返回其状态的指示。

一种保健监测检查通常结合了两个因素：检查（如果有的话）的应用程序或服务响应于所述请求发送到健康验证端点执行，并且结果由工具或框架正在执行健康检查验证的分析。的响应代码表示的应用程序的状态和任选的任何组件或服务，它使用。的延迟或响应时间检查由监测工具或框架进行。图1示出了该模式的执行的概述。

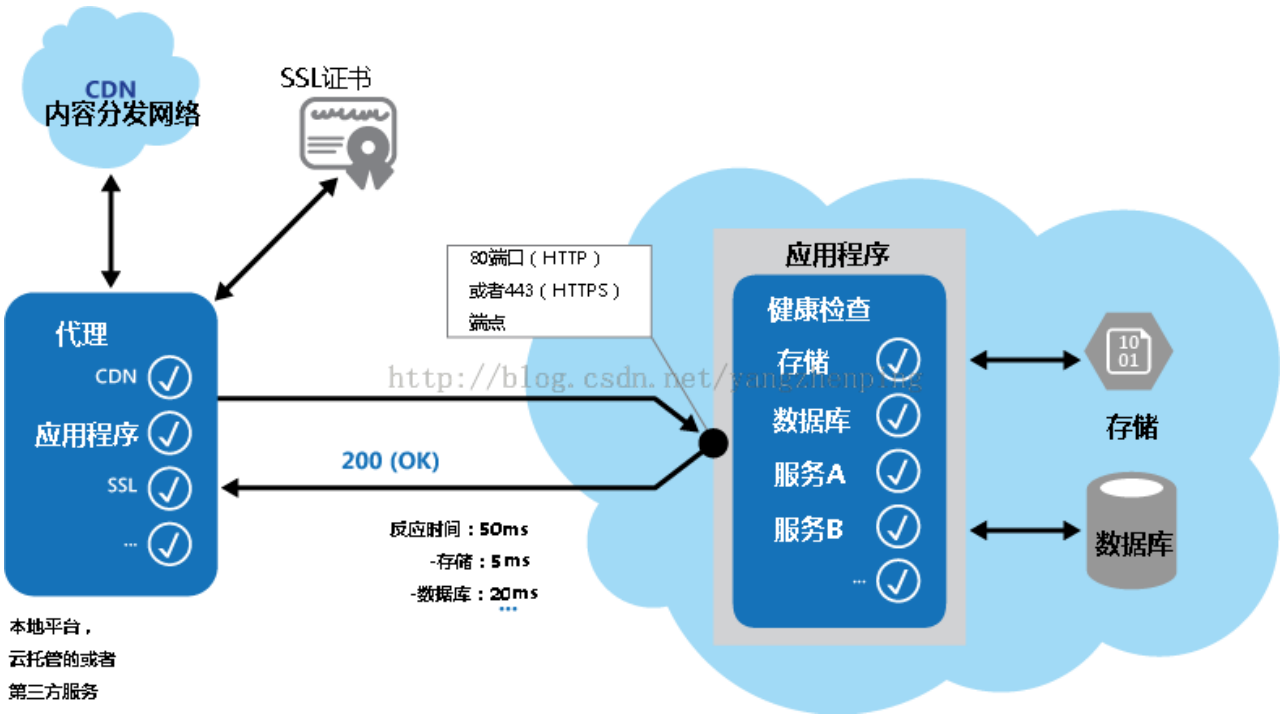


图1 – 模式概述

附加的检查，可能如下进行：在该应用程序的运行状况监视代码包括：

- 检查云存储或可用性和响应时间的数据库。
- 检查位于所述应用程序内，或位于其它地方，但应用程序使用的其他资源或服务。

几个现有的服务和工具可用于监视 web 应用程序通过提交一个请求到一组可配置的端点，并评价针对一组可配置的规则的结果。它相对容易地创建一个服务端点，其唯一的目的是要在系统上执行一些功能测试。

这可以通过监控工具来执行典型的检查包括：

- 验证的响应代码。例如，200 的 HTTP 响应（OK）的指示应用程序作出反应而不会出现错误。该监控系统也可能会检查是否有其他响应代码，给出的结果更全面的指标。
- 检查响应的内容，以检测错误，甚至当返回 200（OK）的状态码。这可以检测到影响返回的网页或服务响应的仅有部分的错误。例如，检查一个页面的标题或寻找某个特定的词组，表示正确的页面被退回。
- 测量响应时间，这表明网络延迟和应用程序把执行请求的时间相结合。增加的值可以指示一个新兴的问题与该应用程序或网络。
- 检查资源或位于该应用程序之外的服务，如由应用程序使用，以从全局高速缓存传递内容的内容分发网络。
- 检查 SSL 证书过期。
- 测量用于该应用程序的 URL 的 DNS 查询的响应时间，以便测量的 DNS 延迟和 DNS 故障。
- 验证返回的 DNS 查询，以确保正确输入的URL。这有助于通过在 DNS 服务器上成功的攻击，以避免恶意请求的重定向。

它也是有用的，在可能情况下，以内部部署和托管的位置运行，从这些不同的检查，以测量和来自不同地方比较的响应时间。理想情况下，你应该监视那些贴近客户，以得到每个位置的性能进行精确的视图位置的应用程序。除了提供一个更为坚固的检查机制，其结果可能会影响部署位置的选择的应用程序，以及是否在一个以上的数据中心部署。

试验还应该对所有客户使用，以确保应用程序正常工作的所有顾客的服务实例运行。例如，如果客户的存储空间分布在多个存储账户，在监测过程中，必须检查所有的这些。

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 如何验证响应。例如，仅仅是一个200（ OK ）状态码足以验证应用程序是否工作正常？虽然这提供了应用程序的可用性的最基本的措施，而且是最小执行这个模式中，它提供了有关操作，趋势，并在应用中可能即将出现的问题的信息很少。

Note

确保应用程序不会正确地当目标资源是发现和处理仅返回200状态码。在某些情况下，使用母版页来承载目标网页的时候，例如，服务器可能会返回一个 200 OK 状态码，而不是一个 404 未找到的代码，即使没有找到目标内容页面。

- 端点的数量，以暴露于一个应用程序。一种方法是将暴露的至少一个端点的应用程序所使用的核心服务，而另一个用于辅助或低优先级的服务，使得不同级别的重要性将被分配给每个监控结果。也可以考虑暴露多个端点，如为每个核心服务，以提供额外的监控粒度。举例来说，一个健康的验证检查可以检查数据库，存储和应用程序使用外部地理编码服务;每个都需要不同级别的正常运行时间和响应时间。应用程序可能仍然是健康的，如果地理编码服务，或其他一些后台任务，是几分钟不可用。
- 是否使用相同的终点监测作为用于一般访问，而是设计为健康验证检查一个特定的路径;例如，/健康检查/{GUID}/对一般接入端点。这允许应用程序内的某些功能测试由监测工具，例如添加新的用户注册，登录，以及将一个测试的顺序被执行，同时也证实，一般接入终端是可用的。
- 收集在服务响应于监控请求，以及如何返回该信息的信息类型。大多数现有的工具和框架只看该 HTTP 状态代码端点的回报。要恢复和验证其他信息，可能需要创建一个自定义监控实用程序或服务。
- 多少信息收集。在检查过程中进行过度处理可以重载应用和影响其他用户，并且所花费的时间可能超过监控系统的超时，使得它标志着该应用程序为不可用。大多数的应用包括仪表，如错误处理程序，并且记录性能和详细的错误信息的性能计数器，这可能是足够的，而不是从一个健康验证检查返回的附加信息。
- 如何配置安全监控端点保护他们免受公众使用;这可能暴露该应用程序的恶意攻击，风险敏感信息的曝光，还是吸引拒绝服务（ DoS ）攻击。典型地，这应该在应用程序的配置来完成，以便它可以容易地更新，而无需重新启动该应用程序。可以考虑使用以下一种或多种技术：通过要求认证◦Secure 端点。这可以通过使用在请求报头中的身份验证的安全密钥或通过传递凭证与请求来实现，条件是，监控服务或工具支持认证。
- Use 一个不起眼的或隐藏的端点。例如，暴露在端点上一个不同的 IP 地址，以所使用的默认的应用程序的 URL，一个非标准的 HTTP 端口上配置端点，和/或使用复杂的路径测试页。它通常可以指定在应用程序配置额外的端点地址和端口，并为这些端点的 DNS 服务器（如果需要），以避免直接指定 IP 地址添加条目。
- Expose 上接受一个参数的端点的方法，诸如键的值或操作模式的值。根据不同的请求时收到的代码可以执行特定测试或一组测试，或者返回一个 404 这个参数提供的值（未找到）错误，如果不能被识别的参数值。所识别的参数值可以在该应用程序的配置进行设置。

Note1

DoS 攻击是可能对一个单独的端点，它执行基本功能测试，而不会影响应用程序的动作的影响较小。理想情况下，应避免使用测试可能暴露敏感信息。如果你必须返回，可能是对攻击者有用的信息，考虑如何将保护端点免受未经授权的访问数据。在这种情况下，仅仅依靠默默无闻是不够的。还应该考虑使用 HTTPS 连接和加密的任何敏感数据，尽管这会增加服务器上的负载。

- 如何访问正在使用认证固定的端点。不是所有的工具和框架可被配置为包括与健康验证请求的凭证。例如，微软的 Azure 内置健康验证功能无法提供身份验证凭据。一些第三方的替代品，可以是 Pingdom 的，Panopta，NewRelic 的，和 Statuscake。
- 如何确保监控代理是否正确地履行。一种方法是，以暴露一个端点仅返回来自应用程序的配置或可被用来测试剂的随机值的值。

Note2

还要确保监控系统进行自身检查，如自检和内置的测试，以避免它在发出假阳性结果。

何时使用这个模式

这种模式非常适合于：

- 监控网站和 Web 应用程序，以验证可用性。
- 监控网站和 Web 应用程序，以检查其是否工作正常。
- 监控中间层或共享服务来检测和隔离故障，可能影响其他应用程序。
- 要在应用程序中补充现有的仪器，如性能计数器和错误处理程序。卫生检验检查并不能取代的日志和审计中的应用程序的需求。仪表能够提供有价值的信息为现有的框架，监视计数器和错误日志来检测故障或其他问题。然而，它不能提供的信息，如果该应用程序是不可用的。

例子

下面的代码示例，从 HealthCheckController 类的 HealthEndpointMonitoring.Web 项目采取包括可以下载本指南的样品，演示露出一个端点进行一系列健康检查。

该 CoreServices 方法，如下所示，执行在应用程序中使用的服务的一系列检查。如果所有的测试中没有错误执行，该方法返回一个 200（OK）状态码。如果有任何的测试引发了异常，该方法返回一个 500（内部错误）状态码。当发生错误时的方法，可任选地返回附加信息，如果该监控工具或框架能够利用它。

```
public ActionResult CoreServices()
{
 try
 {
  // Run a simple check to ensure the database is available.
  DataStore.Instance.CoreHealthCheck();

  // Run a simple check on our external service.
  MyExternalService.Instance.CoreHealthCheck();
 }
 catch (Exception ex)
 {
  Trace.TraceError("Exception in basic health check: {0}", ex.Message);

  // This can optionally return different status codes based on the exception.
  // Optionally it could return more details about the exception.
  // The additional information could be used by administrators who access the
  // endpoint with a browser, or using a ping utility that can display the
  // additional information.
  return new HttpStatusCodeResult((int)HttpStatusCode.InternalServerError);
 }
 return new HttpStatusCodeResult((int)HttpStatusCode.OK);
}
```

该 ObscurePath 方法显示了如何读取应用程序配置的路径，并用它作为测试端点。这个例子也说明了如何接受一个 ID 作为参数，并用它来检查有效的请求。

```
public ActionResult ObscurePath(string id)
{
 // The id could be used as a simple way to obscure or hide the endpoint.
 // The id to match could be retrieved from configuration and, if matched,
 // perform a specific set of tests and return the result. It not matched it
 // could return a 404 Not Found status.

 // The obscure path can be set through configuration in order to hide the endpoint.
 var hiddenPathKey = CloudConfigurationManager.GetSetting("Test.ObscurePath");

 // If the value passed does not match that in configuration, return 403 "Not Found".
 if (!string.Equals(id, hiddenPathKey))
 {
```

```
    return new HttpStatusCodeResult((int)HttpStatusCode.NotFound);
  }

  // Else continue and run the tests...
  // Return results from the core services test.
  return this.CoreServices();
}
```

该 TestResponseFromConfig 方法显示了如何可以公开执行一个指定的配置设定值检查的端点。

```
public ActionResult TestResponseFromConfig()
{
  // Health check that returns a response code set in configuration for testing.
  var returnStatusCodeSetting = CloudConfigurationManager.GetSetting(
                                        "Test.ReturnStatusCode");

  int returnStatusCode;

  if (!int.TryParse(returnStatusCodeSetting, out returnStatusCode))
  {
    returnStatusCode = (int)HttpStatusCode.OK;
  }

  return new HttpStatusCodeResult(returnStatusCode);
}
```

监控端点在 Azure 中托管的应用程序

在Azure应用程序监控终端的一些选项包括：

- 使用微软的Azure，的内置功能，如管理服务或流量管理器。
- 使用第三方服务或Microsoft系统中心操作管理器的框架等。
- 创建一个自定义的工具，或者在您自己的或托管的服务器上运行的服务。

> 注意： 尽管 Azure 提供一个合理的全面的监控选项，您可以决定使用额外的服务和工具，以提供额外的信息。

Azure 管理服务提供了各地的警报规则建立了一个全面的内置监控机制。管理服务网页中的 Azure 管理门户 Alerts 部分，可以配置高达每认购10警报规则为您服务。这些规则指定一条件和用于服务诸如 CPU 负载的阈值，或每秒请求或错误的数量，并且该服务可以自动发送电子邮件通知给你在每个规则定义的地址。

您可以监视具体费用取决于您选择适合您的应用程序的托管机制的条件下（如网站，云服务，虚拟机，或移动服务），但所有这些，包括创建使用网络端点警报规则的能力您在为您服务的设置指定。此端点应该及时地作出反应，以使警报系统可以检测到该应用程序是否正常运行。

> 注意： 有关创建监视警报的详细信息，请参阅 MSDN 上的管理服务。

如果你的主机在 Azure 云服务网络和工作角色或虚拟机应用程序时，您可以采取的内置服务在Azure中所谓的流量管理器中的一个优势。流量管理器是一个路由和负载平衡服务，可以将请求分发到您的云服务托管的应用程序基于一系列的规则和设置的具体实例。

除了请求路由，流量管理坪的 URL，端口和相对你定期指定的路径来确定其规则中定义的应用程序的实例是活动的，并响应请求。如果它检测到一个状态代码 200（OK）它标志着应用程序可用，其他状态的代码会导致流量管理器来标记应用程序离线。您可以查看流量管理器控制台的状态和配置规则来重新路由请求被响应的应用程序的其他实例。

但是，请记住，流量管理器将只等待10秒钟，以接收来自监控URL的响应。因此，你应该确保你的健康验证码这个时间范围内执行，允许网络延迟从流量管理器往返于您的应用程序，然后再返回。

> 注意： 有关使用 Windows 流量管理器来监视你的应用程序的更多信息，请参阅 MSDN 上微软 Azure Traffic Manager 的。流量管理器在多个数据中心部署指南进行了讨论。

12

索引表模式

创建索引过的被查询条件经常被引用的数据存储等领域。这种模式可以通过允许应用程序更快速地定位数据来从数据存储中检索提高查询性能。

背景和问题

许多数据存储通过使用主键组织为实体的集合的数据。应用程序可以使用此键来查找和检索数据。图 1 显示了一个数据存储区保持顾客的信息的例子。主键是客户 ID。

主键（客户 Id）	客户数据
1	LastName: Smith, Town: Redmond, ...
2	LastName: Jones, Town: Seattle, ...
3	LastName: Robinson, Town: Portland, ...
4	LastName: Brown, Town: Redmond, ...
5	LastName: Smith, Town: Chicago, ...
6	LastName: Green, Town: Redmond, ...
7	LastName: Clarke, Town: Portland, ...
8	LastName: Smith, Town: Redmond, ...
9	LastName: Jones, Town: Chicago, ...
...	...
1000	LastName: Clarke, Town: Chicago, ...
...	...

图1 – 按主键组织的客户信息（客户ID）

而主键是该取基于该关键字的值的数据的查询宝贵的，应用程序可能不能够使用主键是否需要基于其它字段来检索数据。在顾客例如，应用程序不能使用该客户ID主键来检索客户，如果它通过指定引用的一些其他属性的值，如在其中客户位于镇标准查询数据完全。要执行一个查询，如这可能需要申请获取并检查每一个客户的记录，这可能是一个缓慢的过程。

许多关系数据库管理系统支持二级索引。一种二次指数是由一个或多个非主（辅助）键领域举办一个单独的数据结构，它表示，其中每个索引值的数据被存储。在一第二索引的项目通常排序方法的第二个键的值，使数据的快速查找。这些指标通常是由数据库管理系统自动进行维护。

由于需要支持您的应用程序执行不同的查询，您可以创建任意多个二级指标。例如，在一个关系数据库中凡客ID是主键的表的客户，也可能是有益的补充辅助指数在镇域如果应用程序频繁查找的客户在其居住的小镇。

然而，尽管二级指标是关系型系统的共同特征，使用云应用大部分NoSQL数据存储不提供同等的功能。

解决方案

如果数据存储不支持二级索引，你可以通过创建自己的索引表手动效仿他们。索引表由指定的键组织数据。三种策略通常用于构建一个索引表，这取决于所需要的二次索引的数目和该应用程序执行的查询的性质：

重复数据的每个索引表中，而是由不同的密钥（完全非规范化）组织它。图2显示了索引表的组织包括城市和姓氏相同的客户信息：

次键（城镇）	客户数据
Chicago	ID: 5, LastName: Smith, Town: Chicago, ...
Chicago	ID: 9, LastName: Jones, Town: Chicago, ...
Chicago	ID: 1000, LastName: Clarke, Town: Chicago, ...
...	...
Portland	ID: 3, LastName: Robinson, Town: Portland, ...
Portland	ID: 7, LastName: Clarke, Town: Portland, ...
Redmond	ID: 1, LastName: Smith, Town: Redmond, ...
Redmond	ID: 4, LastName: Brown, Town: Redmond, ...
Redmond	ID: 6, LastName: Green, Town: Redmond, ...
Redmond	ID: 8, LastName: Smith, Town: Redmond, ...
Seattle	ID: 2, LastName: Jones, Town: Seattle, ...
...	...

次键（姓）	客户数据
Brown	ID: 4, LastName: Brown, Town: Redmond, ...
Clarke	ID: 7, LastName: Clarke, Town: Portland, ...
Clarke	ID: 1000, LastName: Clarke, Town: Chicago, ...
Green	ID: 6, LastName: Green, Town: Redmond, ...
Jones	ID: 2, LastName: Jones, Town: Seattle, ...
Jones	ID: 9, LastName: Jones, Town: Chicago, ...
...	...
Robinson	ID: 3, LastName: Robinson, Town: Portland, ...
Smith	ID: 1, LastName: Smith, Town: Redmond, ...
Smith	ID: 5, LastName: Smith, Town: Chicago, ...
Smith	ID: 8, LastName: Smith, Town: Redmond, ...
...	...

图2 – 索引表执行二级指标的客户数据。数据被复制到每个索引表中。

如果比较的时候，它是通过使用每个键查询的数目的数据是相对静态的这一策略可能是适当的。如果数据是更加动态，保持每个索引表的处理开销可能会变得太大，这种方法是有用的。此外，如果数据量非常大，空间来存储重复的数据所需要的量将显著。

创建由不同的密钥组织的归索引表和通过使用主键而不是重复它引用原始数据，如示于图3中的原始数据被称为一个事实表：

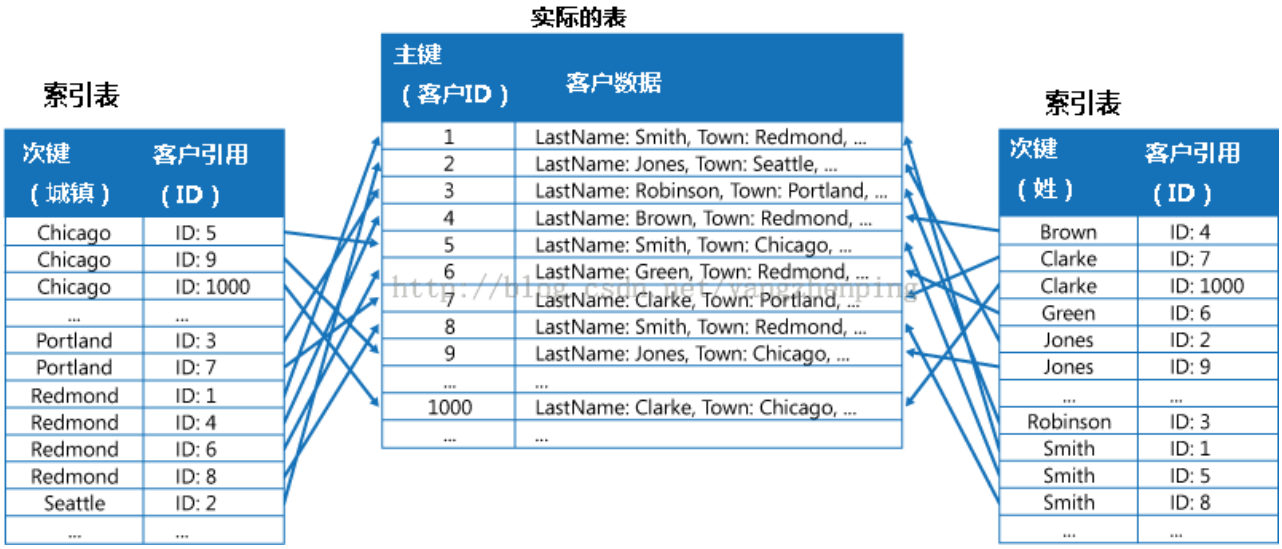


图3 – 索引表执行二级指标的客户数据。该数据是由每个索引表所引用。

这种技术可以节省空间，降低了维护的重复数据的开销。的缺点是，一个应用程序具有通过使用第二密钥来执行两个查找操作以查找数据（找到的主键的索引表中的数据，然后查找在事实表中的数据通过使用主键）。

创建由重复的频繁检索的字段不同的按键组织的部分归索引表。引用原始数据来访问较少频繁访问的字段。图4示出了这种结构。

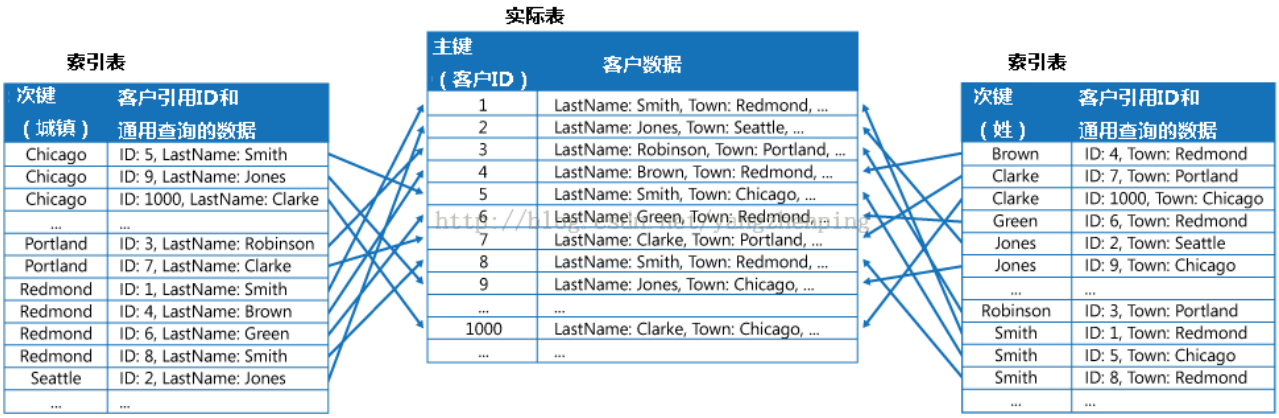


图4 – 索引表执行二级指标的客户数据。经常访问的数据是重复的每个索引表中。

使用这种技术，你可以前两种方法之间取得平衡。可以快速地检索到通过使用单个查找，常用的查询数据，而空间和维护开销是不一样大，复制整个数据集。

如果应用程序通过指定值的组合频繁地查询数据（例如，“查找生活在雷德蒙和具有史密斯的姓所有客户”），则可以实现键的索引表中的项目作为一个级联城市属性和姓氏属性的，如示于图5中的键由镇排序，然后通过名字为那些具有镇相同的值的记录。

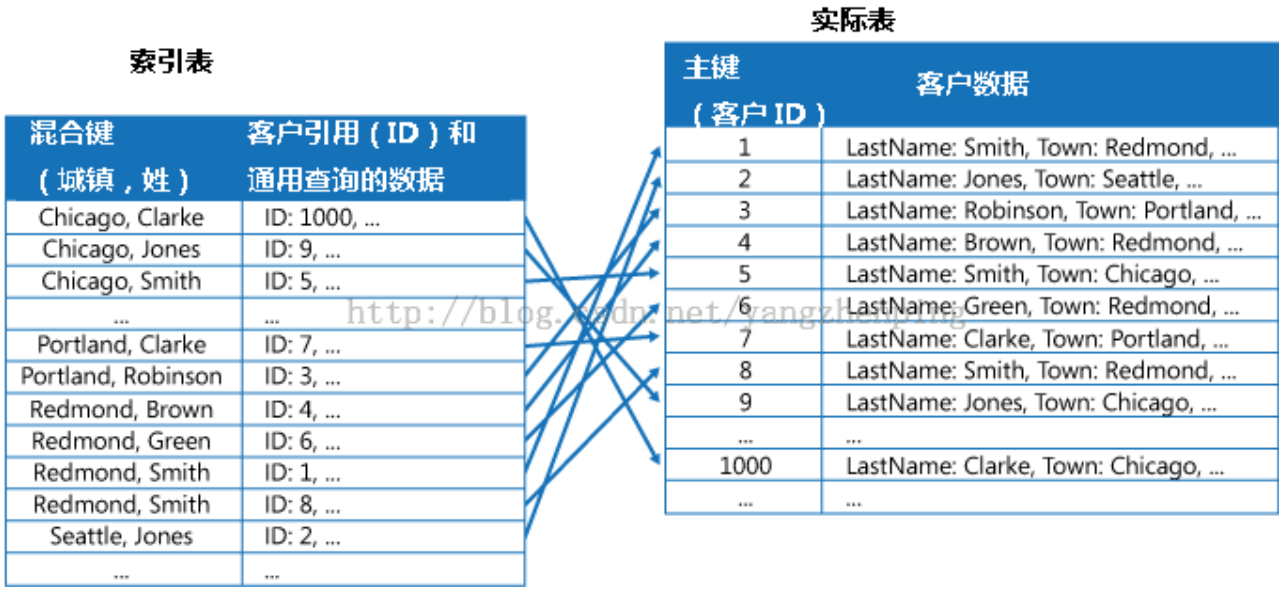


图5 – 基于复合主键索引表

索引表可以加快了分片的数据查询操作，并在那里的碎片密钥散列特别有用。图 6 显示了一个示例，其中分片密钥是客户 ID 的散列。索引表可以由非散列值（城市和名字）组织数据，并提供该哈希分片键作为查找数据。这样可以节省从重复计算散列键的应用（其可以是昂贵的操作），如果它需要检索的数据落在一个范围之内，或者它需要读取的数据，以便在非散列密钥。例如，诸如“查找生活在雷德蒙所有客户”可以由通过定位在索引表中的匹配项（其全部存储在一个连续的块），并按照引用的客户数据尽快解决的查询使用存储在索引表中的碎片的键。

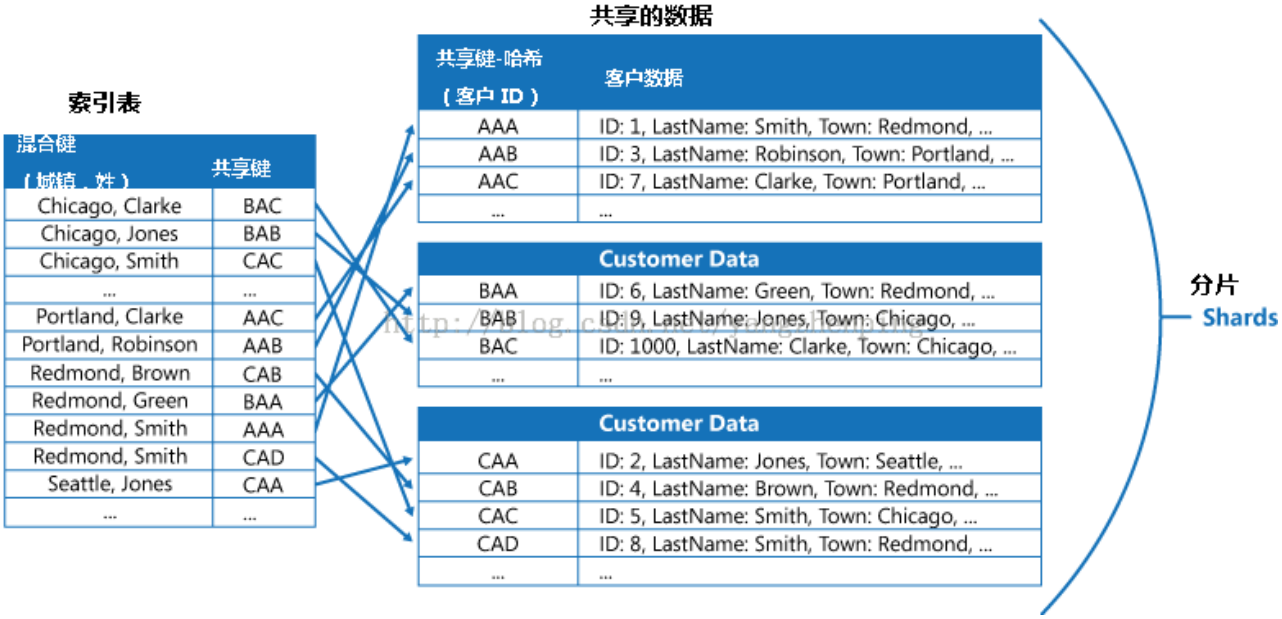


图6 – 索引表中提供了快速查找的分片数据

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 保持辅助索引的开销可能是显著。你必须分析和了解，您的应用程序使用的查询。只创建他们很可能被经常使用的索引表。不要投机创建索引的表，以支持应用程序不执行查询，或者一个应用程序只执行非常偶然。
- 在索引表中复制的数据所用的存储成本和维护数据的多个副本所需的工作条件添加显著开销。
- 执行一个索引表，作为标准化的结构，引用原始数据可能需要的应用程序，以执行两个查找操作以查找数据。第一操作搜索索引表来检索主键，第二个使用的主密钥来获取数据。
- 如果系统包含大量索引表在非常大的数据集，也可以是难以维持索引表和原始数据之间的一致性。有可能设计围绕最终一致性模型的应用。例如，插入，更新或删除数据，一个应用程序可以发送一条消息给一个队

列，并让一个独立的任务执行操作和维护引用该数据不同步的索引表。有关实现最终一致性的更多信息，请参阅数据一致性底漆。

> 注意： 微软 Azure 存储表支持事务更新到同一个分区中保存的数据进行更改（简称实体组的事务）。如果你可以存储一个事实表和在同一个分区的一个或多个索引表中的数据，您可以使用此功能来帮助确保一致性。

索引表可以自行进行分区或分片。

何时使用这个模式

使用这种模式来提高查询性能，当应用程序经常需要使用一键以外的主（或子库）键来检索数据。

这种模式可能不适合时：

- 数据是不稳定的。索引表可能变得过时的速度非常快，使其无效，或者使保持在索引表大于用它制成的任何节省的开销。
- 选作索引表中的二级密钥的场是非常不鉴别，只能有一个小的值的集合（例如，性别）。
- 数据值的选择为一个索引表中的二级密钥的场的平衡是高度倾斜。例如，如果 90% 的记录中包含相同的值中的一个字段，然后创建和维护一个索引表中查找基于该字段中的数据可以施加更大的开销比通过数据扫描顺序。然而，如果查询非常频繁地针对位于对剩余的 10% 的值，该索引可以是有用的。你必须明白的疑问，您的应用程序正在执行，以及如何他们经常执行。

例子

Azure 存储表在云中运行的应用程序提供了一个高度可扩展的键/值数据存储。应用程序存储，并通过指定一个键检索数据值。的数据值可以包含多个字段，但一个数据项的结构是不透明的表存储，这仅仅处理一个数据项作为一个字节数组。

Azure 存储表还支持分片。分片密钥包括两个元件，一个分区键和行密钥。有相同的分区键的数据项都存储在同一分区（碎片），并且项目被存储在一个子库中排键顺序。表存储优化用于执行获取数据下降分区中的连续范围的行键值范围内的查询。如果您正在构建存储在 Azure 的表的信息的云应用，你应该组织你的数据在考虑这个功能。

例如，考虑存储有关电影的信息的应用程序。应用程序经常按流派查询电影（动作片，纪录片，历史，喜剧，戏剧，等等）。可以通过使用类型作为分区键，并指定电影的名称作为行密钥创建一个天青表的分区的每个类型，如图7。

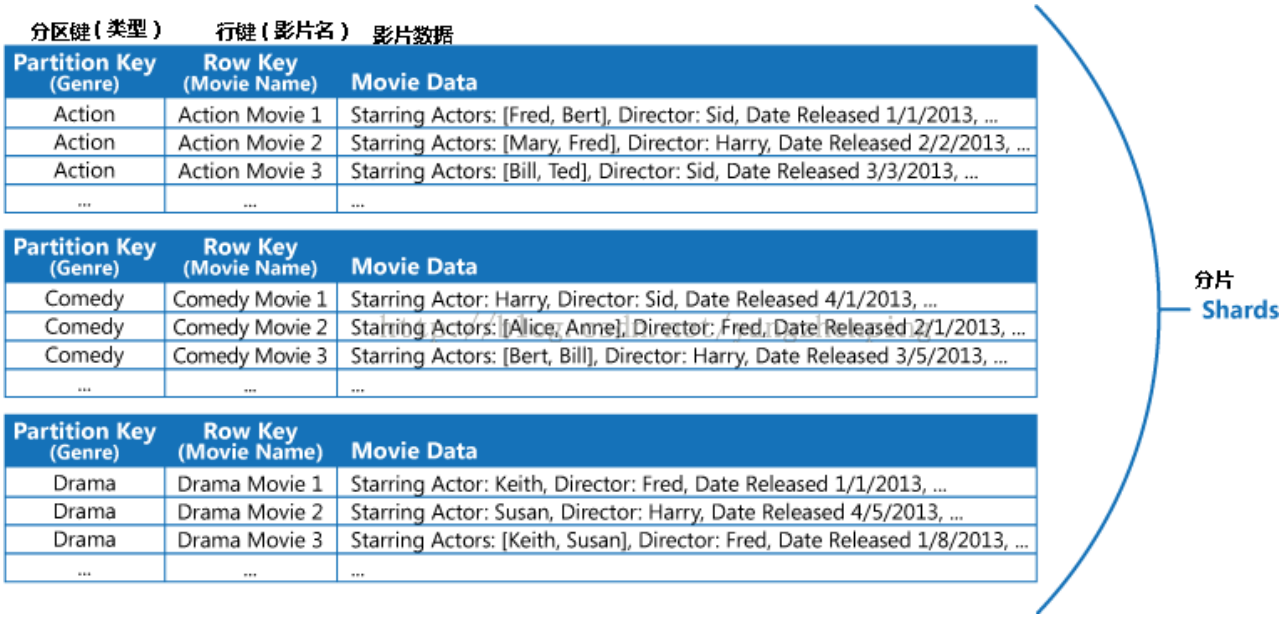


图7 – 存储在 Azure Table 中的电影数据，按流派划分和排序的电影名称

这种方法不太有效，如果该应用程序还需要通过演员主演查询电影。在这种情况下，你可以创建一个单独的Azure表作为一个索引表。分区键是演员和行关键是电影的名字。对于每个演员的数据将被存储在单独的分区。如果一个电影明星不止一个演员，同一部电影会出现在多个分区。

可以通过采用上面的解决方案部分中所述的第一种方式重复在每个分区中保存的值的电影数据。然而，很可能是每个影片将（对于每个演员一次）重复几次，所以它可能是更有效的，以部分非规范化，以支持最常见的查询（如其他演员的姓名）的数据，并实现一个应用程序由包括必要找到在体裁分区的完整信息，分区键来检索任何剩余的细节。这种方法是通过在解决方案部分中的第三项中的说明。图8描述了这种方法。

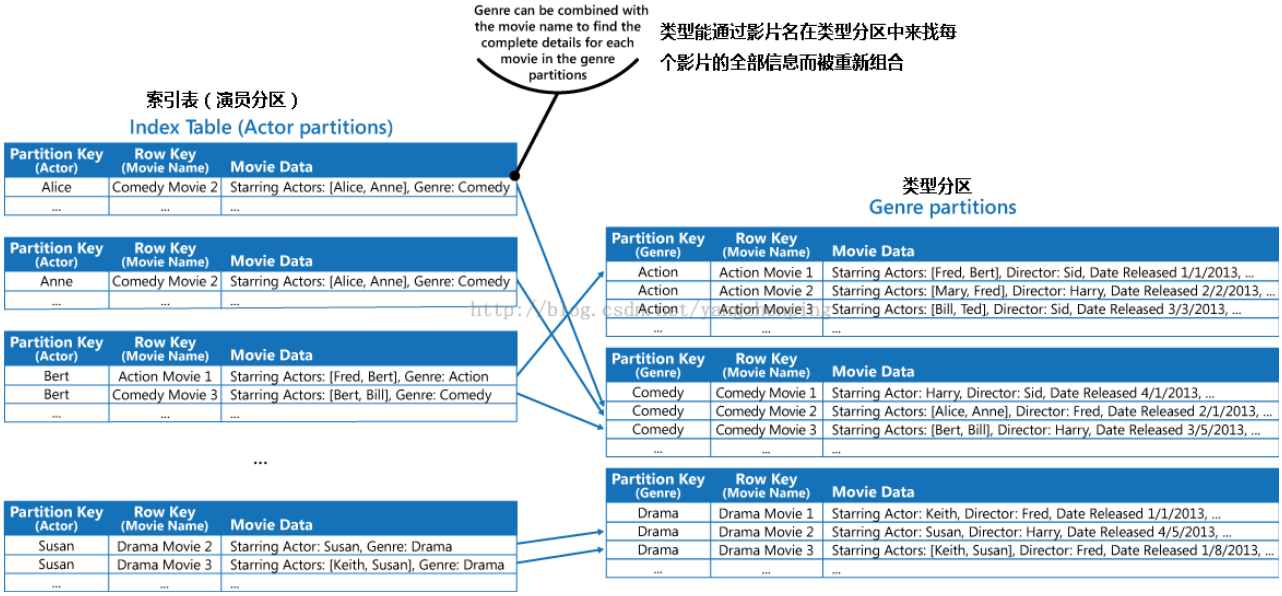


图8 – 演员分区作为索引表的影像数据

13

领导人选举模式

通过协调合作，在分布式应用程序的任务实例集合执行的操作，选举一个实例作为承担管理的其他实例责任的领导者。这个模式可以有助于确保任务实例不互相冲突，导致争用共享资源，或与其他的任务实例正在执行的工作无意中干扰。

背景和问题

一个典型的云应用包括行动协调的方式很多任务。这些任务都可以是实例运行相同的代码和需要访问相同的资源，或者它们可能是可并行工作，以执行复杂计算的各个部分。

任务实例可能为多的时间自主运行，但它也可能是必要的，以协调各实例的操作，以确保它们不发生冲突，导致争用共享资源，或无意中妨碍工作，其他的任务实例正在执行。例如：

- 在基于云的系统，该系统实现了横向扩展，同一个任务的多个实例可以与每个实例服务于不同的用户同时运行。如果这些实例写入到共享资源，也可能是必要的，以协调它们的操作，以防止每个实例从盲目地覆盖由他人进行的更改。
- 如果任务正在执行复杂的计算以并行的单个元素，其结果将需要被聚合时，他们都完成了。

由于任务实例都是同行，没有天生的领导者，能够充当协调员或聚合器。

解决方案

单个任务实例应选充当领导者，这种情况下应协调其他从属任务实例的操作。如果所有的任务实例都运行相同的代码，他们可能都能够充当领导者。因此，选举过程必须谨慎管理，以防止两个或多个实例接管领导者的角色在同一时间。

该系统必须选择的领导者提供了一个坚固的机构。这种机制必须能够应付诸如网络中断或进程故障事件。在许多解决方案中，下属的工作情况监控的领导者，通过某种类型的心跳机制，或通过轮询。如果指定的领导者意外终止或网络故障使得领导者不通下属的工作情况，有必要为他们选出了新的领导人。

有可用于选举的领导者之间的一组任务在分布式环境中，包括几个策略：

- 与排名最低的实例或进程ID选择任务实例。
- 竞速获得一个共享的分布式互斥。第一个任务实例获取该互斥锁处于领先地位。然而，系统必须保证，如果领导者终止或变得与系统的其余部分断开，该互斥被释放，以允许另一个任务实例成为领导者。
- 实施的共同领导人选举算法，如恶霸算法或环算法之一。这些算法是相对简单的，但也有一些更复杂的技术提供。这些算法假定每个候选参选具有唯一的 ID，并且，它们可以用可靠的方式在其他候选人进行通信。

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 选出一个领导者的过程应该是弹性的瞬时和永久失效。
- 必须能够检测到当领导失败，或变成不可用（可能是由于通讯故障）。在这是需要这种检测的速度将取决于系统。有些系统可能能够发挥作用了一会儿没有一个领导者，在此期间造成的领导人变得不可用瞬时故障可能已被排除。在其他情况下，可能有必要立即检测领袖失败并引发新的选举。
- 在实现自动缩放水平的系统中，如果系统鳞背面和关闭一些计算资源的领导者可能被终止。
- 使用一个共享的分布式互斥引入外部服务，提供了互斥锁的可用性依赖。该服务可以构成一个单点故障。如果此服务应该以任何理由变得不可用时，系统将无法选出一个领导者。
- 使用一个专用进程的领导者是一个比较简单的方法。然而，如果该过程失败，可能有显著延迟而被重新启动，并且将得到的延迟可能影响其他进程的性能和响应时间，如果他们正在等待领导人来协调的动作。
- 实施的领导人选举算法之一手动为调整和优化代码的最大灵活性。

何时使用这个模式

使用此模式时，在分布式应用程序的任务，比如云托管解决方案，需要认真协调，也没有天生的领导者。

注意： 避免使领导者在系统中的瓶颈。领导者的目的是协调的附属任务完成的工作，它不一定有机会参加这项工作本身，尽管它应该是有能力这样做，如果任务没有当选领导人。

这种模式可能不适合：

- 如果有一个天生的领导者或专用的过程，可以随时充当领导者。例如，有可能实现一个单进程，其协调该任务的实例。如果这个过程失败或变得不健康，系统可以将其关闭并重新启动它。
- 如果任务之间的协调，可以很容易地通过使用更轻便的机构来实现的。例如，如果几个任务实例仅仅需要对共享资源的访问协调，一个最好的解决办法可能是使用乐观或悲观锁来控制访问该资源。
- 如果一个第三方的解决方案是比较合适的。例如，微软的 Azure HDInsight 服务（基于 Apache Hadoop 的）使用所提供的 ApacheZookeeper 协调的 map / reduce 的汇总任务和汇总数据的服务。它也可以安装并在Azure的虚拟机配置动物园管理员，并将其集成到自己的解决方案，或使用可从微软开放技术的动物园管理员预置的虚拟机映像。欲了解更多信息，请参阅 Apache 的动物园管理员在微软的 Azure 在微软开放技术的网站。

例子

在列入可用于本指南中的示例代码中 LeaderElection 解决方案 DistributedMutex 项目展示了如何使用租赁在 Azure 存储 BLOB 提供了一种机制，实现共享的分布式互斥。此互斥锁可以用来选择在 Azure 云服务的领导者之间的一组角色的实例。第一个角色实例获得租约当选的领导人，并保持领先直至其租赁或直到它无法续租。其他角色实例可以继续监视在领导不再可用的情况下将 BLOB 租赁。

注意

一个 BLOB 租赁是在一个 blob 的排他写锁。单个 BLOB 可以是一整租的在任何一个时间点的问题。角色实例可以请求租约在指定的斑点，而且将被授予租约，如果没有其他租赁在同一个斑点，是由这个或任何其他作用，比如举行，否则将抛出一个异常。

为了减少一个故障角色实例保留无限期租用的可能性，指定了一辈子的租约。当此期满后，租赁变为可用。然而，当一个角色实例持有的租赁也可以请求租约到期，并且将被授予租约的时间再延长。角色实例可以不断重复这一过程，如果它希望保留租约。

有关如何租用一个 blob 的详细信息，请参阅租赁斑点（REST API）在 MSDN 上。

```
public class BlobDistributedMutex
{
 ...
 private readonly BlobSettings blobSettings;
 private readonly Func<CancellationToken, Task> taskToRunWhenLeaseAcquired;
 ...

 public BlobDistributedMutex(BlobSettings blobSettings,
        Func<CancellationToken, Task> taskToRunWhenLeaseAquired)
 {
  this.blobSettings = blobSettings;
  this.taskToRunWhenLeaseAquired = taskToRunWhenLeaseAquired;
 }

 public async Task RunTaskWhenMutexAcquired(CancellationToken token)
 {
  var leaseManager = new BlobLeaseManager(blobSettings);
  await this.RunTaskWhenBlobLeaseAcquired(leaseManager, token);
 }
 ...
```

代码示例中的 RunTaskWhenMutexAquired 上述方法调用以下代码示例来实际获得的租赁所示的 RunTaskWhenBlobLeaseAcquired 方法。该 RunTaskWhenBlobLeaseAcquired 方法异步运行。如果租赁的成功获取，角色实例已经当选的领导者。该 taskToRunWhenLeaseAcquired 委托的目的是为了执行协调其它角色实例的工作。如果未取得租赁，另一个角色实例已当选为领导人和当前角色实例仍然是一个下属。注意，TryAcquireLeaseOrWait 方法是使用 BlobLeaseManager 对象获取租赁一个辅助方法。

```
...
 private async Task RunTaskWhenBlobLeaseAcquired(
  BlobLeaseManager leaseManager, CancellationToken token)
 {
  while (!token.IsCancellationRequested)
  {
   // Try to acquire the blob lease.
   // Otherwise wait for a short time before trying again.
   string leaseId = await this.TryAquireLeaseOrWait(leaseManager, token);

   if (!string.IsNullOrEmpty(leaseId))
   {
    // Create a new linked cancellation token source so that if either the
    // original token is cancelled or the lease cannot be renewed, the
    // leader task can be cancelled.
    using (var leaseCts =
     CancellationTokenSource.CreateLinkedTokenSource(new[] { token }))
    {
     // Run the leader task.
     var leaderTask = this.taskToRunWhenLeaseAquired.Invoke(leaseCts.Token);
     ...
    }
   }
  }
 }
 ...
```

由领导开始的任务还异步执行。虽然这个任务正在运行，下面的代码示例所示的 RunTaskWhenBlobLeaseAquired方法周期性地尝试续订租约。这个动作有助于确保该角色实例保持领先。在简单解决方案中，续订请求之间的延迟小于对租赁期限，以防止另一角色实例当选的领导人指定的时间。如果更新失败，以任何理由，任务被取消。如果租用未能被更新或任务被取消（可能为角色实例关停的结果），租赁被释放。在这一点上，这个或另一个角色实例可能被选为领导者。下面的代码提取物显示出此过程的一部分。

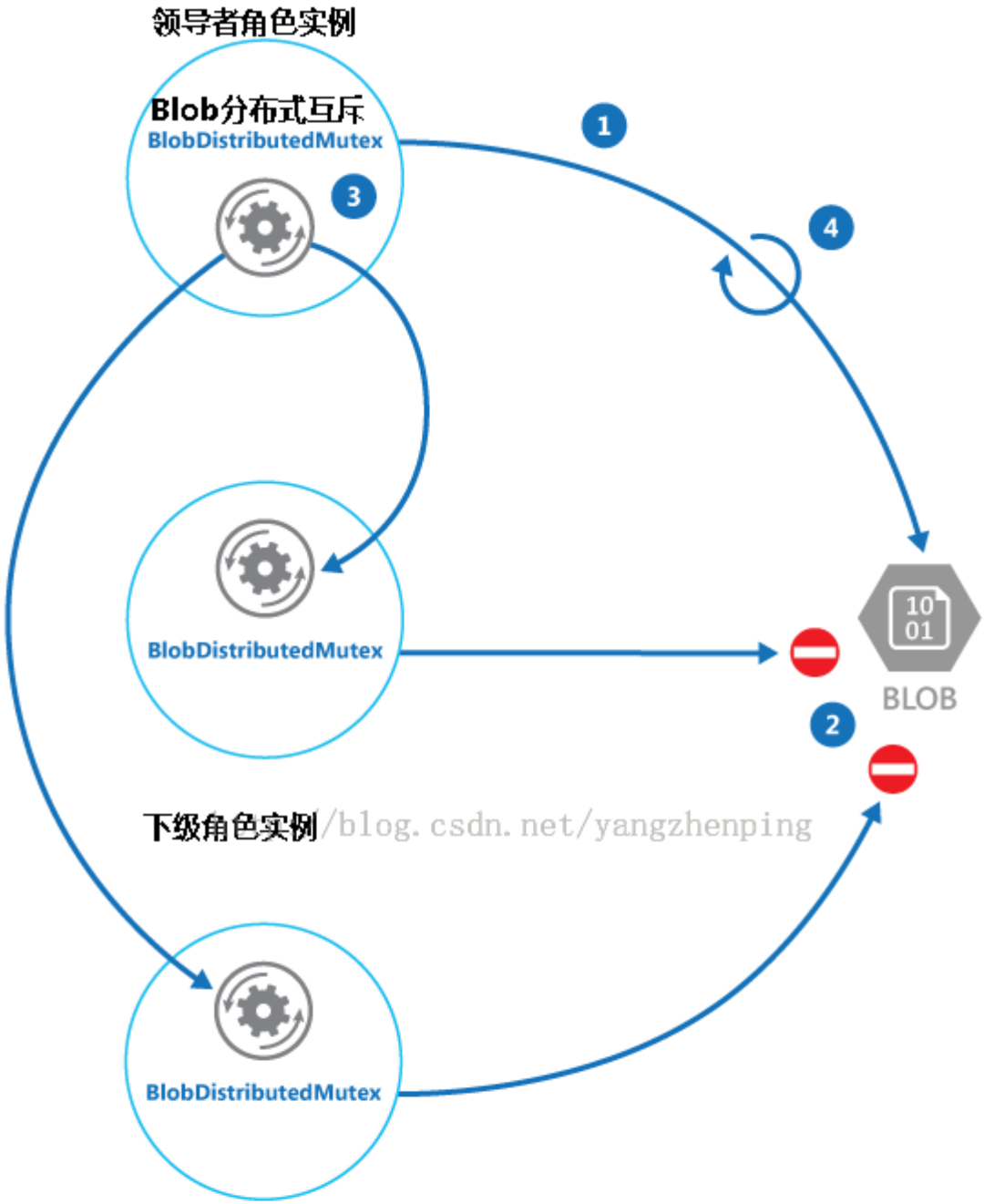
```
...
 private async Task RunTaskWhenBlobLeaseAcquired(
  BlobLeaseManager leaseManager, CancellationToken token)
 {
```

```
  while (...)
  {
    ...
    if (...)
    {
      ...
      using (var leaseCts = ...)
      {
        ...
        // Keep renewing the lease in regular intervals.
        // If the lease cannot be renewed, then the task completes.
        var renewLeaseTask =
          this.KeepRenewingLease(leaseManager, leaseId, leaseCts.Token);

        // When any task completes (either the leader task itself or when it could
        // not renew the lease) then cancel the other task.
        await CancelAllWhenAnyCompletes(leaderTask, renewLeaseTask, leaseCts);
      }
    }
  }
 }
 ...
}
```

该 KeepRenewingLease 方法是使用 BlobLeaseManager 对象续租另一个 helper 方法。该 CancelAllWhenAnyCompletes 方法取消指定为前两个参数的任务。

图 1 示出了 BlobDistributedMutex 类的功能。



1.角色实例调用BlobDistributedMutex对象的RunTaskWhenMutexAcquired方法，并授予租约过的BLOB。角色实例当选的领导人。
2.其他角色实例调用RunTaskWhenMutexAcquired方法和被封锁。
3.领导者的RunTaskWhenMutexAcquired方法运行的coordeinates从属角色实例的工作任务。
4.在领导的RunTaskWhenMutexAcquired方法定期续订租约。

图1 – 使用 BlobDistributedMutex 类选出一个领导者和运行协调操作的任务

下面的代码示例显示了如何使用 BlobDistributedMutex 类的辅助角色。此代码获取租赁了一个名为 MyLeaderCoordinatorTask 在开发的仓储租赁容器中的 blob，并指定在 MyLeaderCoordinatorTask 方法定义的代码应该运行，如果角色实例当选的领导人。

```
var settings = new BlobSettings(CloudStorageAccount.DevelopmentStorageAccount,
  "leases", "MyLeaderCoordinatorTask");
var cts = new CancellationTokenSource();
var mutex = new BlobDistributedMutex(settings, MyLeaderCoordinatorTask);
mutex.RunTaskWhenMutexAcquired(this.cts.Token);
...

// Method that runs if the role instance is elected the leader
private static async Task MyLeaderCoordinatorTask(CancellationToken token)
{
  ...
}
```

请注意有关样品溶液中的以下几点：

- BLOB 是一个潜在的单点故障。如果 Blob 服务不可用，或的 blob 是人迹罕至，领导者将无法续租，并没有其他的作用，比如将能够获得租约。在这种情况下，没有作用，例如将能够充当领导者。然而，blob 服务被设计为弹性的，所述 blob 的服务，以便彻底失败被认为是极不可能的。
- 如果被领导者摊位正在执行的任务，领导者可能会继续续租，防止任何其他角色实例从获得租约，并接管了领导作用，以协调任务。在现实世界中，领导者的健康应该频繁地进行检查。
- 在选举过程具有不确定性。你不能做任何假设哪个角色实例将得到的blob租约，成为领导者。
- 不应使用任何其他目的用作的 blob 租赁的目标的 blob。如果一个角色实例尝试将数据存储在该的 blob，该数据将不能被访问，除非该角色实例是领导者和持有的 blob租约。

14

实体化视图模式

产生过在一个或多个数据存储中的数据预填充的观点时，数据被格式化以不利于所需的查询操作的一种方式。这种模式可以帮助支持高效的查询和提取数据，并提高应用程序的性能。

背景和问题

何时存储数据时，优先级为开发者和数据管理员经常集中在如何将数据存储，而不是它是如何读出。所选择的存储格式通常是密切相关的数据，用于管理数据的大小和数据的完整性，并且在使用的那种存储的要求的格式。例如，使用的 NoSQL 文献商店时，该数据通常被表示为一系列的聚集体，其每一个包含了所有的信息，该实体。

然而，这可能对查询产生负面影响。当查询需要从一些实体，如订单的几个客户没有所有的顺序的信息的汇总的数据的一个子集，它必须提取所有的相关实体的数据，以获得所需的信息。

解决方案

以支持高效的查询，一个常见的解决方案是生成，预先，即物化数据中最适合于所要求的结果集的格式的图。其中源数据不是一个格式适合于查询，在那里产生一个合适的查询中的实体化视图模式描述产生数据的预先填充的观点在环境中是困难的，或者其中的查询性能差，由于该数据或该性质数据存储区。

这些实例化视图，其中只包含一个查询所需的数据，以便应用程序能够快速获得他们需要的信息。除了连接表或组合的数据实体，物化视图可以包括计算列或数据项的指定作为查询的一部分的当前值，组合值或对数据项执行的转换的结果，和值。物化视图甚至可以就某一个单一的查询优化。

关键的一点是，一个实体化视图，它包含的数据完全是一次性的，因为它可以完全从源数据存储重建。实例化视图是不能直接更新的应用程序，因此它实际上是一个专门的缓存。

何时该视图更改源数据，视图必须被更新以包括新的信息。这可以在适当的日程自动发生，或在系统检测到变化到原始数据的时候。在其他情况下，可能需要手动重新生成视图。

图1示出如何实体化视图图案可能被使用的例子。

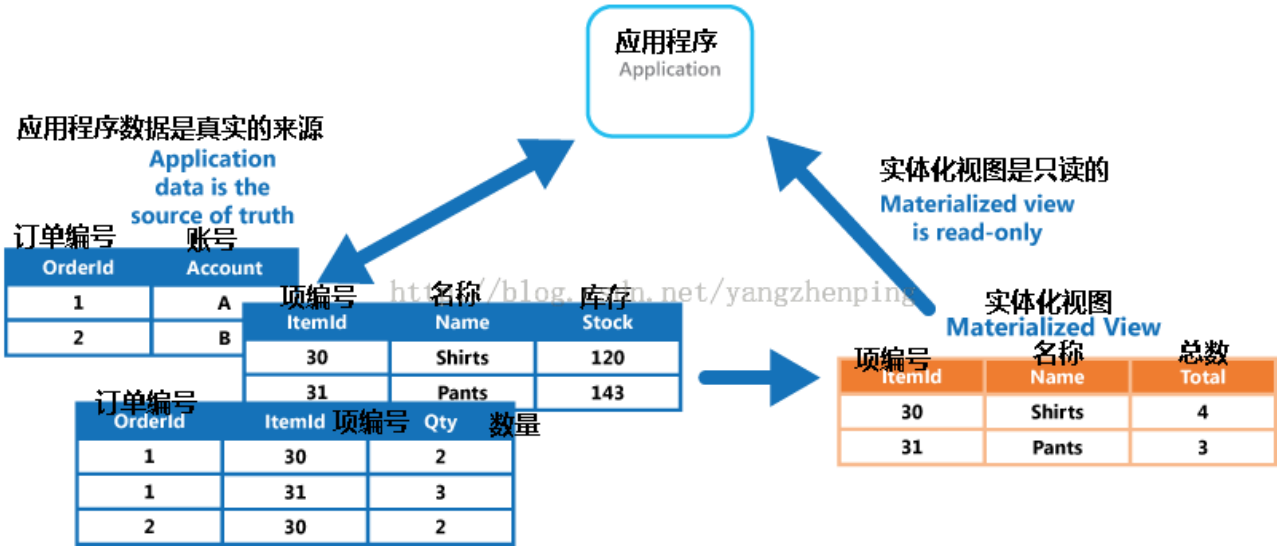


图1 – 实体化视图模式

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 考虑如何以及何时该视图将被更新。理想的情况下，将被再生响应于一个事件，指示改变到所述源数据，尽管在某些情况下，这可能导致过度的开销，如果源数据发生急剧的变化。或者，考虑使用计划任务，外部触发或手动操作来启动该视图的再生。
- 在某些系统中，使用事件采购图案保持仅修改的数据的事件存储区时，例如，实体化视图可能是必要的。通过检查所有事件，以确定当前状态预先填充的观点可以得到从事件存储信息的唯一方式。在使用事件采购时比其它情况下，有必要测量的优点是物化视图可以提供。物化视图往往是专门针对一个或少数的查询。如果许多查询必须被使用，维护实例化视图可能会导致不可接受的存储容量的要求和存储成本。
- 生成的视图时，和更新视图时，如果这发生在一个日程表考虑数据一致性的影响。如果源数据发生了变化时，生成的视图时的点，在该视图中的数据的复制可能会与原来的数据完全一致。
- 考虑在那里你将存储的视图。认为不必位于同一商店或分区的原始数据。它可能是从几个不同的分区合并的一个子集。
- 如果视图是短暂的，仅仅是用来通过反映该数据的当前状态来提高查询性能，或提高扩展性，它可被存储在高速缓存中或者在一个较不可靠的位置。它可以的，如果失去了重建。
- 当定义一个实体化视图中，在数据项或列的基础上计算的或现有的数据项的转换的视图，在查询传递的值，或者对这些值，其中，这是适当的组合发挥其最大价值。
- 凡存储机制支持它，考虑索引实体化视图，以进一步提高性能。大多数关系型数据库支持索引的意见，因为这样做是基于Apache Hadoop的大数据解决方案。

何时使用这个模式

这种模式非常适合于：

- 创建实例化视图以上数据是难以直接查询，或者查询必须以提取存储在归一化，半结构化或非结构化的方式数据非常复杂。
- 创建临时视图，可以显着提高查询性能，或可直接充当UI源视图或数据传输对象（ DTO 的），进行报告，或进行显示。
- 支持偶尔连接或断开连接的情况，其中连接到数据存储并不总是可用的。该视图可能在这种情况下被本地缓存。
- 简化查询和在不需要源数据格式的知识的方式曝光数据用于实验。例如，通过在一个或多个数据库，或在 No SQL 的存储的一个或多个结构域结合不同的表，然后格式化的数据，以满足它的最终用途。
- 提供访问源数据的特定子集，出于安全或隐私原因，不应该是一般访问，公开进行修改，或者完全暴露给用户。
- 使用基于他们的个人能力不同的数据存储在弥合脱节。例如，通过使用云存储中是有效率的用于写入作为基准数据存储，并能提供良好的查询和读取性能保持实例化视图的关系数据库。

这种模式可能不适合于下列情况：

- 源数据简单，便于查询。
- 源数据非常迅速的变化，或者可以在不使用视图来访问。中创建视图处理开销可能会避免在这些情况下。
- 一致性是一个高优先级。的意见可能并不总是与原始数据完全一致。

例子

图2示出了使用实体化视图模式的一个例子。在订单，订单项数据，并在微软的Azure存储帐户单独的分区表的客户相结合，生成包含在电子类别中的每个产品销售总额的视图，客户是谁的采购数量的计数在一起每个项目。

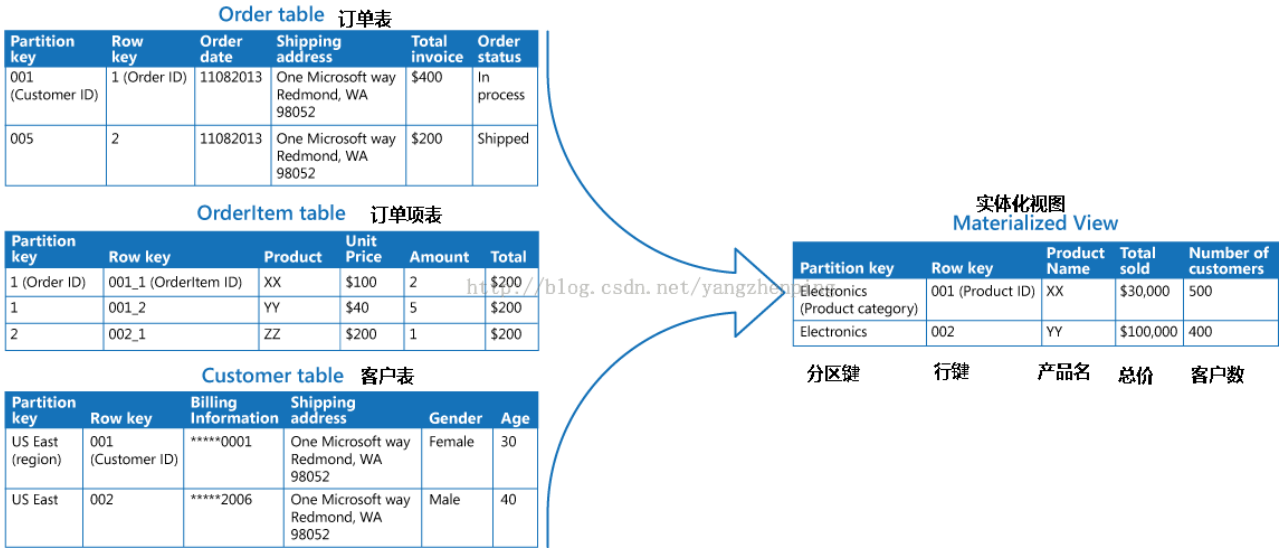


图2 – 使用实体化视图模式产生销售的总结

创建这个实例化视图需要复杂的查询。然而，通过将查询结果作为实体化视图，用户可以轻松获得的结果和直接使用它们，或将其纳入另一个查询。观点很可能在一个报告系统或仪表板中使用，所以可以更新计划的基础上，如每周一次。

> 注意： 虽然这个例子使用的 Azure 表存储，许多关系数据库管理系统还提供了实例化视图的原生支持。

15

管道和过滤器模式

分解，执行复杂处理成一系列可重复使用分立元件的一个任务。这种模式可以允许执行的处理进行部署和独立缩放任务元素提高性能，可扩展性和可重用性。

背景和问题

一个应用程序可能需要执行各种关于它处理的信息不同复杂的任务。一个简单，但不灵活的方式来实施这个应用程序可以执行此处理为单一模块。然而，这种方法有可能减少用于重构代码，对其进行优化，或者重新使用它，如果是在应用程序中其他地方所需要的相同的处理的部件的机会。

图1通过使用单片式的方式示出了与处理数据的问题。一个应用程序接收并处理来自两个来源的数据进行处理。从每个源数据是由执行一系列任务来转换该数据，并传递结果给应用程序的业务逻辑之前的独立模块进行处理。

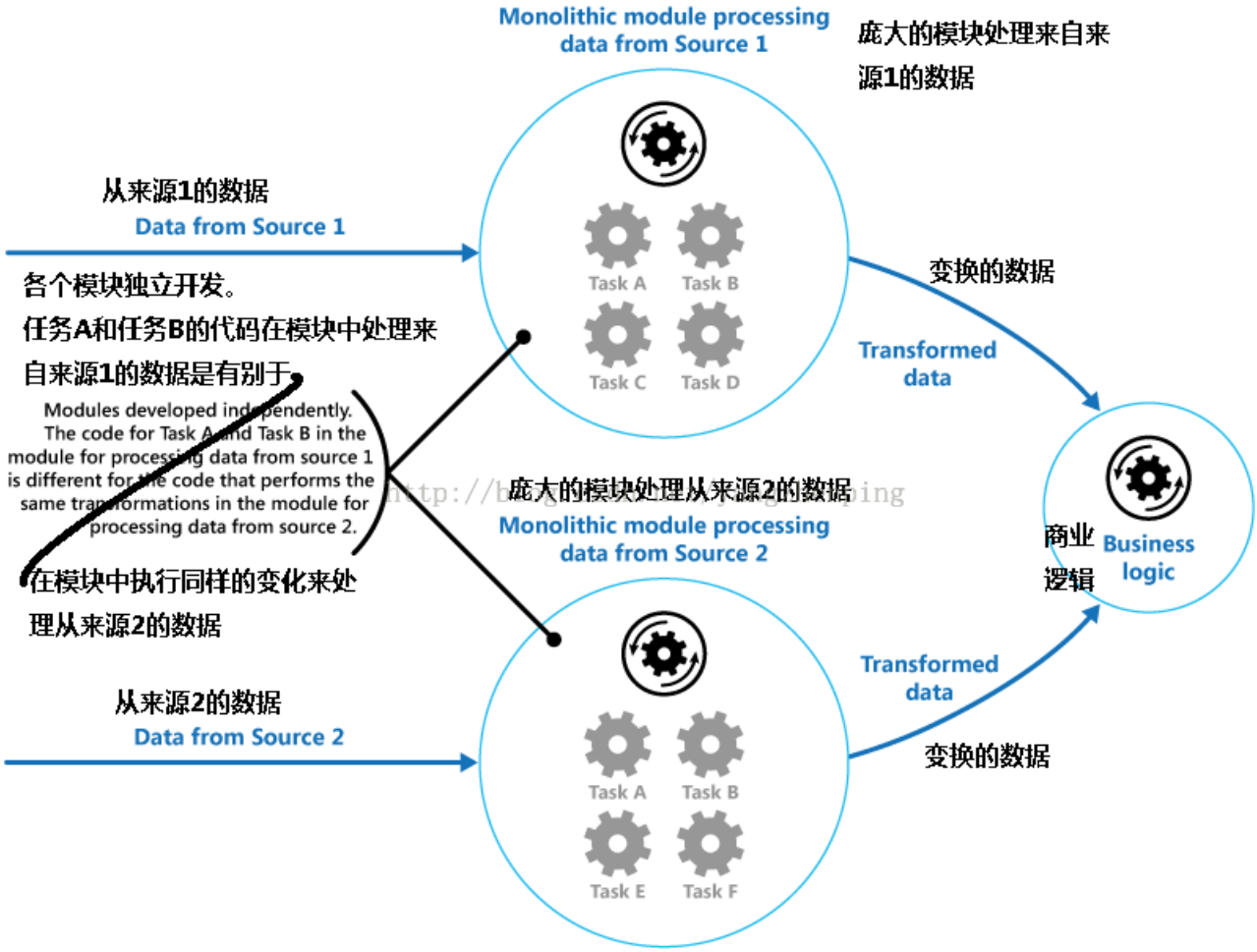


图1 – 使用单一模块实现的解决方案

部分的单片模块执行的任务在功能上是非常相似的，但在模块已被分开设计的。实现该任务的代码被紧密模块内耦合，并且此代码已开发具有很少或没有给定重新使用或可伸缩性的思想。

然而，由每个模块或每个任务的部署要求执行的处理任务，可能会改变，因为业务需求进行修改。有些任务可能是计算密集型的，并可能受益于强大的硬件上运行，而其他人可能并不需要如此昂贵的资源。此外，额外的处理可能需要在将来，或顺序，其中由所述处理执行的任务可能会改变。一个解决方案是必需的，解决了这些问题，并且增加的可能性代码重用。

解决方案

分解需要为每个数据流转换为一组离散的元件（或过滤器）的处理，其中每一个执行单任务。通过标准化每个组件接收和发射的数据的格式，这些过滤器可以组合在一起成为一个管道。这有助于避免重复代码，并且可以很容易地移除，替换或集成额外的组件，如果处理要求改变。图2显示了这种结构的一个例子。

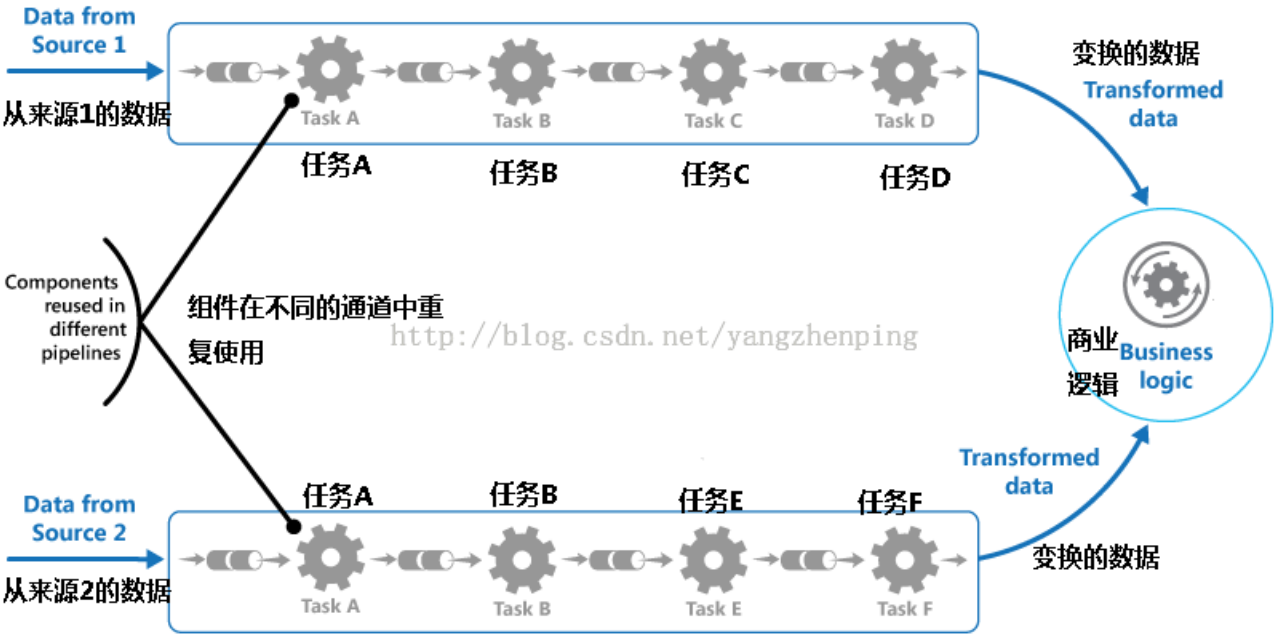


图2 – 通过使用管道和过滤器实现的解决方案

处理一个请求所花费的时间取决于最慢的过滤器管道中的速度。这可能是一个或多个滤波器可能被证明是一个瓶颈，尤其是如果出现在从一个特定的数据源的数据流的大量请求。流水线结构的一个关键优点是它提供了机会，运行速度慢的过滤器的并联情况下，使系统能够分散负载并提高吞吐量。

可以独立缩放组成一个管道可以在不同的机器上运行过滤器，使他们和可以利用的弹性，许多云计算环境提供的优势。过滤器是计算密集型可以在高性能的硬件上运行，而其他要求不高的过滤器可以对商品（便宜）的硬件来承载。过滤器甚至不需要是在同一数据中心或地理位置，它允许在一个管道中的每个元素的环境下接近它需要的资源来运行。

图3示出了从源 1 施加到管道中的数据的一个例子。

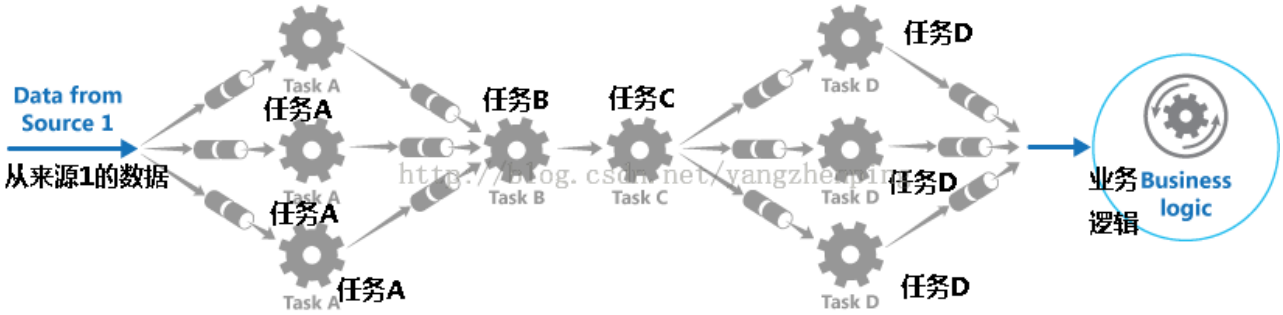


图3 – 在一个管道负载平衡组件

如果一个滤波器的输入和输出被构造为一个流，它可能是能够进行的处理并行的每个过滤器。在流水线的第一个过滤器可以开始工作，并开始发射其结果，它们会直接传递到序列中的下一个过滤器之前的第一过滤器已经完成它的工作。

另一个好处是灵活性，这种模式可以提供。如果一个过滤器发生故障或者其上运行的机器不再可用时，管道可能能够重新安排滤波器所执行的工作，并指示此工作到组件的另一个实例。单个过滤器的故障不会必然导致整个管道的故障。

使用管道和过滤器与补偿交易模式相结合的模式可以提供一种替代的方法来实现分布式事务。分布式事务可以被分解成单独的赔的任务，每个都可以通过使用一个过滤器，也实现了补偿事务图案来实现。在一个管道中的过滤器可以在运行接近它们保持数据被实现为单独的托管工作。

问题和注意事项

在决定如何实现这个模式时，您应考虑以下几点：

- 复杂性。增加的灵活性，这种模式提供了还可以引入复杂性，特别是如果被分布在不同的服务器上在管道的过滤器。
- 可靠性。使用一个基础结构，可以确保在管道中的过滤器之间流动的数据也不会丢失。
- 幂等性。如果在管道中的过滤失败接收到消息后，任务被重新调度到过滤器的另一个实例，所述部分工作可能已经完成。如果这个工作更新的全局状态的某些方面（如存储在数据库中的信息），同样更新可以重复。如果公布的结果，在管道中的下一个过滤器后，过滤器出现故障，但在此之前表示，该公司已经成功地完成了它的工作可能会出现类似的问题。在这些情况下，相同的工作可以由过滤器的另一个实例被重复，导致相同的结果要贴两次。这可能导致在管道随后过滤两次处理相同的数据。因此，在一个管道的过滤器应该被设计为幂等。欲了解更多信息，请参见乔纳森·奥利弗的博客幂等模式。

- 重复的消息。如果在管道中的过滤器可以发布一个消息给流水线的下一个阶段之后发生故障时，过滤器的另一个实例，可以执行（由幂等考虑以上所描述的），并且将发布相同消息的拷贝到流水线。这可能导致同样的信息的两个实例被传递到下一个过滤器。为了避免这种情况，该管道应检测并消除重复的消息。

> 注意： 如果要实现管道使用消息队列（如微软的Azure服务总线队列），消息队列基础设施可以提供自动重复消息检测和清除。

- 上下文和状态。在管道中，每个过滤器主要运行在孤立和不应该做这件事是如何被调用的任何假设。这意味着，每一个过滤器必须具有足够的上下文与它能够执行它的工作提供。这种情况下可包含相当数量的状态信息。

何时使用这个模式

使用这种模式时：

- 由一个应用程序所需的处理可以很容易地被分解成一组离散的，独立的步骤。
- 由应用程序执行的处理步骤具有不同的可扩展性要求。

注意： 它可能会向组过滤器应扩展一起在相同的过程。欲了解更多信息，请参阅计算资源整合模式。

- 灵活性是必需的，以允许通过一个应用程序，或能力进行添加和删除步骤中的处理步骤重新排序。
- 该系统可以受益于分配处理跨不同服务器的步骤。
- 一个可靠的解决方案是必需的，当数据正在被处理的最小化在一个步骤失败的影响。

这种模式可能不适合时：

- 通过应用程序执行的处理步骤并不是独立的，或者他们必须共同作为同一事务的一部分来执行。
- 在一个步骤所需的上下文或状态的信息量使得这种方法效率很低。它可能会持续状态信息到数据库代替，但不要使用此策略，如果在数据库上的额外负载会导致过度竞争。

例子

可以使用消息队列的一个序列，以提供执行流水线所需的基础设施。最初的消息队列接收未处理的消息。实现为过滤器的任务侦听此队列的消息的组件，它执行其工作，然后投递转化的消息序列中的下一个队列。另一个过滤器的任务可以侦听在这个队列中的消息，对其进行处理，后的结果到另一个队列，依此类推，直到完全转化的数据出现在队列中的最后一个消息。

如果你正在构建一个解决方案，在 Azure 上，你可以使用服务总线队列提供了可靠的，可扩展的排队机制。下面所示的 ServiceBusPipeFilter 类提供了一个例子。它演示了如何实现接收从队列中输入消息，处理这些邮件的过滤器，并张贴结果到另一个队列。

> 注意： 该 ServiceBusPipeFilter 类在 PipesAndFilters 解决方案 PipesAndFilters.Shared 项目定义。此示例代码都可以可以下载本指导意见。

```
public class ServiceBusPipeFilter
{
 ...
 private readonly string inQueuePath;
 private readonly string outQueuePath;
 ...
 private QueueClient inQueue;
 private QueueClient outQueue;
 ...

 public ServiceBusPipeFilter(..., string inQueuePath, string outQueuePath = null)
 {
   ...
   this.inQueuePath = inQueuePath;
   this.outQueuePath = outQueuePath;
 }

 public void Start()
 {
  ...
  // Create the outbound filter queue if it does not exist.
  ...
  this.outQueue = QueueClient.CreateFromConnectionString(...);

  ...
  // Create the inbound and outbound queue clients.
  this.inQueue = QueueClient.CreateFromConnectionString(...);
 }

 public void OnPipeFilterMessageAsync(
  Func<BrokeredMessage, Task<BrokeredMessage>> asyncFilterTask, ...)
 {
```

```
  ...

  this.inQueue.OnMessageAsync(
    async (msg) =>
  {
    ...
    // Process the filter and send the output to the
    // next queue in the pipeline.
    var outMessage = await asyncFilterTask(msg);

    // Send the message from the filter processor
    // to the next queue in the pipeline.
    if (outQueue != null)
    {
      await outQueue.SendAsync(outMessage);
    }

    // Note: There is a chance that the same message could be sent twice
    // or that a message may be processed by an upstream or downstream
    // filter at the same time.
    // This would happen in a situation where processing of a message was
    // completed, it was sent to the next pipe/queue, and then failed
    // to complete when using the PeekLock method.
    // Idempotent message processing and concurrency should be considered
    // in a real-world implementation.
  },
  options);
}

public async Task Close(TimeSpan timespan)
{
  // Pause the processing threads.
  this.pauseProcessingEvent.Reset();

  // There is no clean approach for waiting for the threads to complete
  // the processing. This example simply stops any new processing, waits
  // for the existing thread to complete, then closes the message pump
  // and finally returns.
  Thread.Sleep(timespan);

  this.inQueue.Close();
  ...
}
```

```
  ...
}
```

在 ServiceBusPipeFilter 类 Start 方法连接到一对输入和输出队列，以及关闭方法从输入队列断开。该 OnPipeFilterMessageAsync 方法执行消息的实际处理;该 asyncFilterTask 参数这种方法指定要执行的处理。该 OnPipeFilterMessageAsync 方法等待输入队列中收到的消息，因为它到达，并张贴结果到输出队列通过运行在每个邮件的 asyncFilterTask 参数指定的代码。队列本身的构造函数中指定。

样品溶液的过滤器实现了在一组工作角色。每个工人的作用可独立进行调整，这取决于它执行的业务处理的复杂性，或者它需要执行此处理的资源。此外，各辅助角色的多个实例可以并行地运行，以提高吞吐量。

下面的代码显示了一个名为 PipeFilterARoleEntry 的 Azure 工作者角色，这是在样品溶液中 PipeFilterA 项目定义。

```
public class PipeFilterARoleEntry : RoleEntryPoint
{
  ...
  private ServiceBusPipeFilter pipeFilterA;

  public override bool OnStart()
  {
    ...
    this.pipeFilterA = new ServiceBusPipeFilter(
      ...,
      Constants.QueueAPath,
      Constants.QueueBPath);

    this.pipeFilterA.Start();
    ...
  }

  public override void Run()
  {
    this.pipeFilterA.OnPipeFilterMessageAsync(async (msg) =>
    {
      // Clone the message and update it.
      // Properties set by the broker (Deliver count, enqueue time, ...)
      // are not cloned and must be copied over if required.
      var newMsg = msg.Clone();

      await Task.Delay(500); // DOING WORK

      Trace.TraceInformation("Filter A processed message:{0} at {1}",
        msg.MessageId, DateTime.UtcNow);
```

```
    newMsg.Properties.Add(Constants.FilterAMessageKey, "Complete");

    return newMsg;
   });

   ...
  }

  ...
}
```

这个角色包含 ServiceBusPipeFilter 对象。在角色 OnStart 方法连接到队列接收输入的信息并张贴输出消息（队列的名称在常量类中定义）。 Run 方法调用 OnPipeFilterMessagesAsync 方法来对接收到的（在本例中，该处理通过等待较短的时间段模拟的）的每个消息执行某些处理。何时处理完成时，一个新的消息被构造包含结果（在这种情况下，输入消息被简单地增加了一个自定义属性），并将该消息发送到输出队列。

示例代码中包含一个名为 PipeFilterBRoleEntry 在 PipeFilterB 项目的另一名工人的作用。这个角色类似于 PipeFilterARoleEntry 不同之处在于它的 Run 方法进行不同的处理。在本例中的解决方案，这两种作用结合起来，构建一个管道;为 PipeFilterARoleEntry 角色输出队列是用于 PipeFilterBRoleEntry 角色的输入队列。

样品溶液还提供了两个名为 InitialSenderRoleEntry（在 InitialSender 项目）和 FinalReceiverRoleEntry（在 FinalReceiver 项目），进一步的角色。该 InitialSenderRoleEntry 作用提供了在管道中的初始消息。OnStart 方法连接到单个队列和运行方法的帖子的方法来此队列。这个队列是所使用的 PipeFilterARoleEntry 作用，所以发布一条消息到这个队列的输入队列导致由 PipeFilterARoleEntry 作用来接收和处理消息。经处理的信息，然后通过 PipeFilterBRoleEntry 作用传递。

为 FinalReceiveRoleEntry 角色输入队列是用于 PipeFilterBRoleEntry 角色的输出队列。 Run 方法在 FinalReceiveRoleEntry 作用，如下图所示，接收到该消息，并且执行一些最后的处理。然后将其写入了过滤器的管道跟踪输出添加自定义属性的值。

```
public class FinalReceiverRoleEntry : RoleEntryPoint
{
 ...
 // Final queue/pipe in the pipeline from which to process data.
 private ServiceBusPipeFilter queueFinal;

 public override bool OnStart()
 {
  ...
  // Set up the queue.
  this.queueFinal = new ServiceBusPipeFilter(...,Constants.QueueFinalPath);
  this.queueFinal.Start();
```

```
    ...
  }

  public override void Run()
  {
    this.queueFinal.OnPipeFilterMessageAsync(
      async (msg) =>
      {
        await Task.Delay(500); // DOING WORK

        // The pipeline message was received.
        Trace.TraceInformation(
          "Pipeline Message Complete - FilterA:{0} FilterB:{1}",
          msg.Properties[Constants.FilterAMessageKey],
          msg.Properties[Constants.FilterBMessageKey]);

        return null;
      });
    ...
  }

  ...
}
```

16

优先级队列模式

优先发送到服务，以便具有较高优先级的请求被接收和高于一个较低优先级的更快速地处理请求。这种模式是在应用程序是有用的，它提供不同的服务级别保证或者针对独立客户。

背景和问题

应用程序可以委托给其他服务的具体任务;例如，为了执行后台处理或与其他应用程序或服务的整合。在云中，消息队列通常用于将任务委派给后台处理。在许多情况下，请求由服务接收的顺序是不重要的。然而，在某些情况下，可能需要优先考虑的具体要求。这些要求必须早于较低优先级的其他可能先前已发送由应用程序进行处理。

解决方案

队列通常是先入先出（FIFO）结构，而消费者通常会收到他们发布到队列中的顺序相同的消息。然而，一些消息队列支持优先级的消息传递;应用程序发布一条消息可以分配优先级的消息，并在队列中的消息会自动重新排序，使得具有较高优先级的消息将这些优先级较低的前被接收。图1示出了一个队列，它提供优先权的消息。

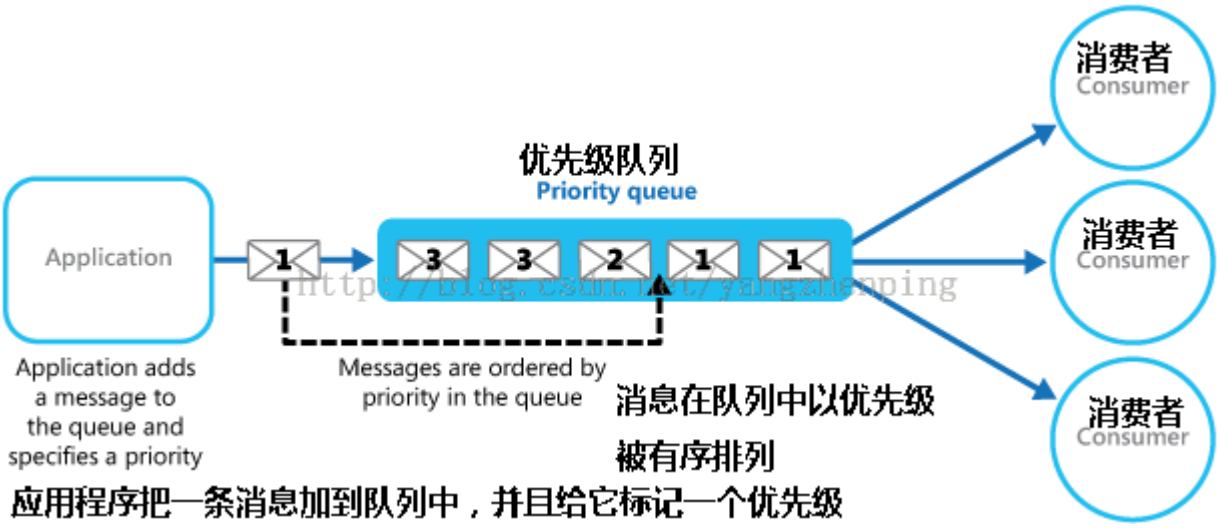


图1 – 使用支持消息优先级排队机制

> 注意： 大多数消息队列的实现支持多个消费者（以下的竞争消费者模式）和消费过程的数量可以按比例增加或减小的需求支配。

在不支持基于优先级的消息队列系统中，一种替代的解决方案是为每一个优先级的独立队列。该应用程序负责将邮件投递到适当的队列。每个队列可以有一个单独的消费者池。高优先级队列可以有更快的硬件比低优先级队列中运行的消费者一个更大的泳池。图2示出了这种方法。

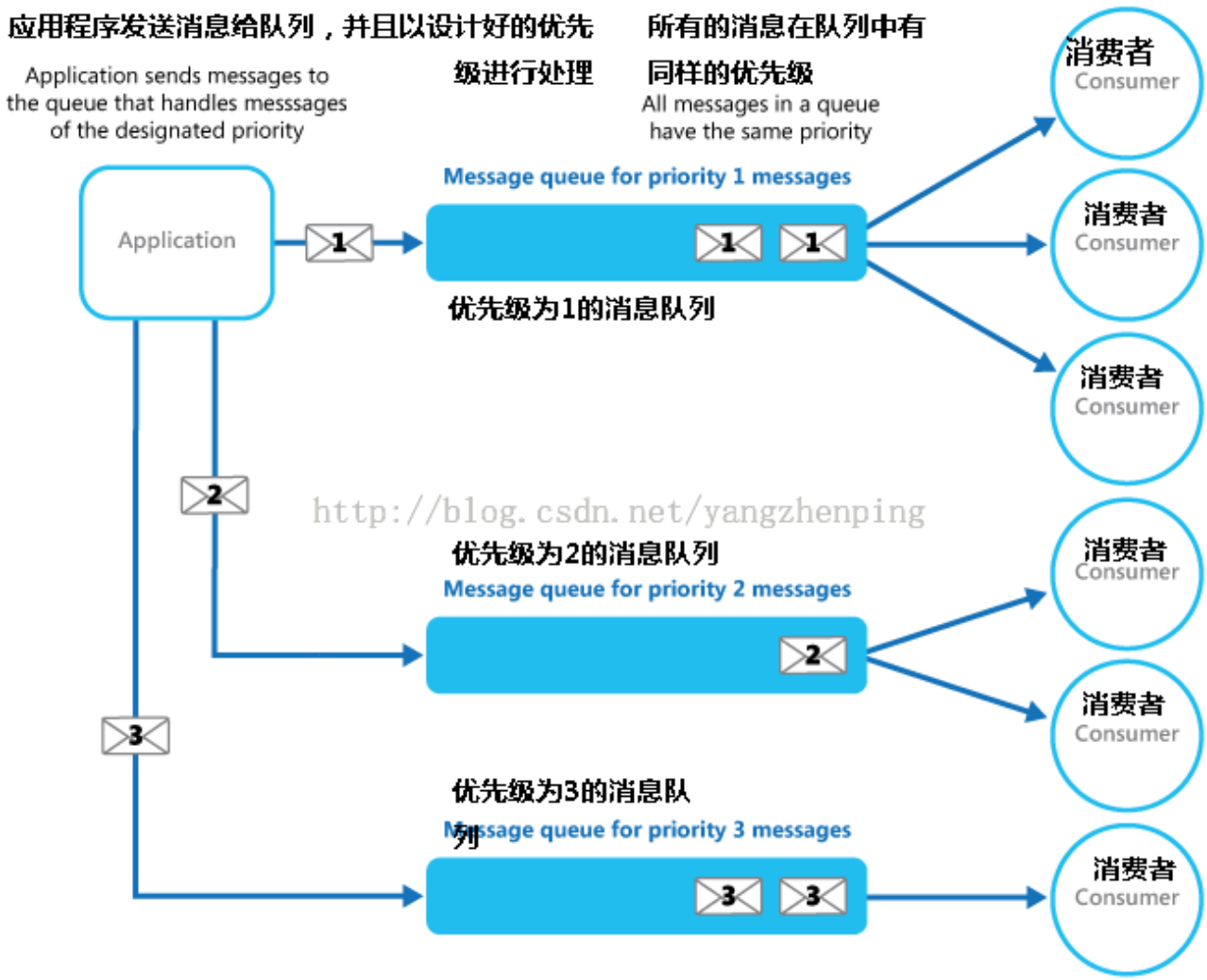


图2 – 使用不同的消息队列为每个优先级

这种策略的变化是有消费者认为检查对高优先级队列中的消息，然后再才开始从低优先级队列中读取消息，如果没有更高优先级的消息都在等待的一个池。还有，使用消费过程的一个池的溶液之间的一些语义差异（或者使用支持不同的优先级或多个队列，每个处理一个单一的优先级消息的消息的单个队列），以及使用多个队列用溶液为每个队列一个单独的游泳池。

在单池的做法，高优先级的消息总是会收到以前低优先级的消息处理。在理论中，具有非常低的优先级的消息可以被不断地取代，并且可能永远不会被处理。在多池的方法，较低优先级的报文将总是被处理，只是不一样迅速的那些更高的优先级的（取决于池和它们具有可用资源的相对大小）。

使用优先级排队机制可提供以下优点：

- 它允许应用程序以满足必要的可用性或性能优先的业务需求，如提供不同级别的服务，以客户的特定群体。
- 它可以帮助最大限度地降低运营成本。在单队列的方式，你可以缩减用户的数量，如果有必要的。高优先级消息仍将被首先处理（虽然可能更慢），和低优先级的消息可能会延迟更长。如果您已实现与消费者的每一

个单独的队列池多个消息队列的方式，可以减少消费者的池低优先级队列，或者甚至停止所有监听的讯息的消费者暂停处理一些非常低优先级队列这些队列。

- 在多个消息队列的方法可以帮助划分的基础上处理要求的消息，以最大限度地提高应用程序的性能和可扩展性。例如，重要的任务，可以优先被立即运行，而不太重要的后台任务可以被安排在不太繁忙的时段运行的接收器来处理接收处理。

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 定义优先级的解决方案的情况下。例如，“高优先级”可能意味着，信息应该在十秒内进行处理。标识要求处理高优先级的项目，以及其他什么资源必须分配给符合这些标准。
- 确定是否所有高优先级的项目必须在任何优先级较低的项目之前进行处理。如果该消息是由消费者的一个池被处理，可能有必要提供一种可抢先和暂停正在处理的低优先级消息，如果更高优先级的消息，有一个任务的机制。
- 在多个队列中的方法，使用该监听所有的队列，而不是一个专门的客户池的每个队列的消费过程的一个池时，消费者必须应用一种算法，以确保它总是从那些从低之前较高优先级的队列提供服务的消息优先级队列。
- 监视处理的高和低优先级队列中的速度，以确保在这些队列中的消息的预期的速率进行处理。
- 如果需要，以保证低优先级的消息将被处理时，可能有必要实现与消费者的多个池的多个消息队列的方法。或者，在一个支持消息优先队列，它可能会动态地增加一个排队的消息的优先级，因为它的年龄。然而，该方法依赖于消息队列提供此功能。
- 使用单独的队列中每个消息优先级最适合有少数明确定义的优先级系统。
- 消息优先级可以通过系统逻辑决定的。例如，而不是明确的高和低优先级的消息，他们可以被指定为“自费客户”，或“非自费的客户。”根据您的商业模式，你的系统可能会分配更多的资源，从收费处理消息付费用户比非自费的。
- 有可能是检查队列的消息相关联的金融和处理成本（一些商业邮件系统的消息被发布或检索每次收取一小笔费用，每次一个队列中查询消息）。检查多个队列时，该成本将有所增加。
- 它可以是能够动态调整的基础上，该池所服务的队列的长度消费者的一个池的大小。欲了解更多信息，请参阅自动缩放指导。

何时使用这个模式

这种模式非常适合场景：

- 系统必须处理可能有不同的侧重点多个任务。
- 不同的用户或租户应配以不同的优先级。

例子

微软 Azure 不提供经过整理的本地支持邮件自动优先级排队机制。然而，它确实提供了 Azure 的服务总线主题和订阅，支持排队机制，提供邮件过滤，具有多种灵活的功能，使其非常适合用在几乎所有的优先级队列的实现在一起。

一个 Azure 的解决方案，可以实现服务总线话题，其中一个应用程序可以发布消息，以同样的方式作为一个队列。消息可以包含在应用程序定义的自定义属性的形式的元数据。服务总线订阅可以与主题相关联，并且这些订阅可以筛选根据它们的属性信息。当一个应用程序将消息发送到一个主题，该消息被定向到从那里它可以被消费者阅读相应的订阅。消费者的过程可以检索使用相同的语义消息队列（订阅是一个逻辑队列）从一个订阅消息。

图 3 示出了使用的 Azure 服务总线主题和订阅的解决方案

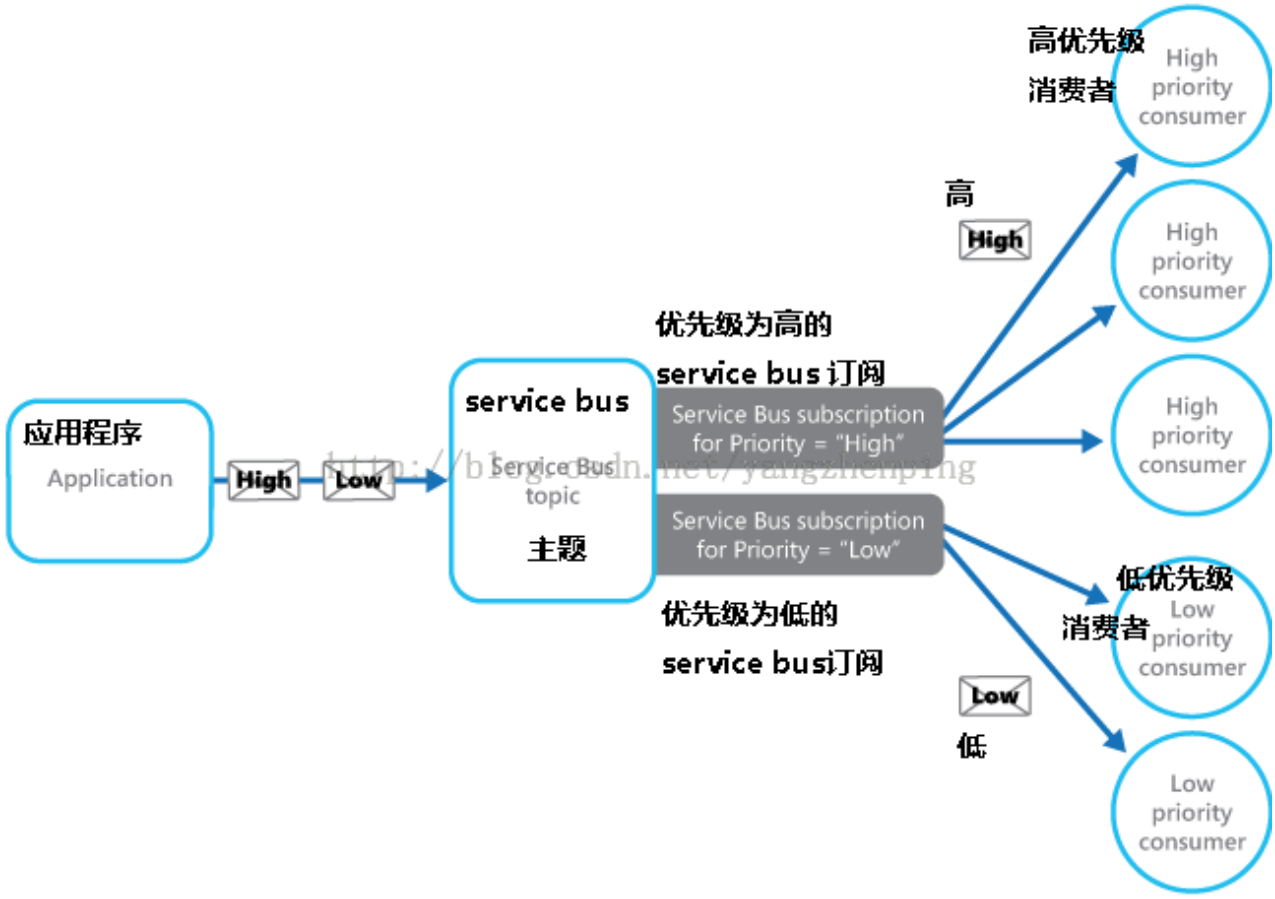


图3 – 实现与 Azure 的服务总线主题和订阅优先级队列

在图3中的应用程序创建多个消息和每个消息与价值分配被称为优先级的自定义属性，无论是高还是低。该应用程序的帖子，这些消息的一个话题。这个主题有两个相关的订阅，这两个滤波器的消息通过检查优先级属性。一位接受认购，其中优先级属性设置为高的消息，而其他接受其中优先级属性设置为低的消息。消费者池读取每个订阅的消息。高优先认购有较大的游泳池，而这些消费者可能会更强大（且昂贵）的计算机上运行有提供比消费者在低优先级池的更多资源。

请注意，没有什么特别的高，低优先级消息在这个例子中指定。这些仅仅是指定为每个消息中的属性的标签，并用于引导消息发送到一个特定的订阅。如果附加的优先级是必需的，它是比较容易地创建进一步的订阅和消费者进程池来处理这些优先级。

在可用于此引导代码时 Queue 解决方案包含这种方法的一个实现。该解决方案包含一个名为 PriorityQueue.High 和 PriorityQueue.Low 两个工作角色的项目。这两个辅助角色继承的类被称为 PriorityWorkerRole 它包含用于连接到一个指定的预订中 OnStart 方法的功能。

该 PriorityQueue.High 和 PriorityQueue.Low 辅助角色连接到不同的预订，他们的配置设置来定义。管理员可以配置每个角色的不同数量要运行;通常会有比 PriorityQueue.Low 工作者角色的 PriorityQueue.High 辅助角色更多的实例。

在 PriorityWorkerRole 类的 Run 方法安排虚拟 ProcessMessage 的方法（在 PriorityWorkerRole 类定义）的队列中接收到的每个消息被执行。下面的代码显示了运行和 ProcessMessage 的方法。在类的 QueueManager，在 PriorityQueue.Shared 项目定义，提供了辅助方法使用的 Azure 服务总线队列。

```
public class PriorityWorkerRole : RoleEntryPoint
{
  private QueueManager queueManager;
  ...

  public override void Run()
  {
    // Start listening for messages on the subscription.
    var subscriptionName = CloudConfigurationManager.GetSetting("SubscriptionName");
    this.queueManager.ReceiveMessages(subscriptionName, this.ProcessMessage);
    ...;
  }
  ...

  protected virtual async Task ProcessMessage(BrokeredMessage message)
  {
    // Simulating processing.
    await Task.Delay(TimeSpan.FromSeconds(2));
  }
}
```

该 PriorityQueue.High 和 PriorityQueue.Low 辅助角色既覆盖 ProcessMessage 的方法的默认功能。下面的代码显示了 ProcessMessage 的方法为 PriorityQueue.High 辅助角色。

```
Copy


protected override async Task ProcessMessage(BrokeredMessage message)
{
  // Simulate message processing for High priority messages.
  await base.ProcessMessage(message);
  Trace.TraceInformation("High priority message processed by " +
    RoleEnvironment.CurrentRoleInstance.Id + " MessageId: " + message.MessageId);
}
```

当一个应用程序将消息发布到与所使用的 PriorityQueue.High 和 PriorityQueue.Low 辅助角色的订阅相关联的主题，它指定了优先使用优先级的自定义属性，如在下面的代码示例。此代码（这是在 PriorityQueue.Sender 项目 WorkerRole 类实现），使用的 QueueManager 类的 SendBatchAsync 辅助方法发帖分批的话题。

```
// Send a low priority batch.
var lowMessages = new List<BrokeredMessage>();

for (int i = 0; i < 10; i++)
{
  var message = new BrokeredMessage() { MessageId = Guid.NewGuid().ToString() };
  message.Properties["Priority"] = Priority.Low;
  lowMessages.Add(message);
}

this.queueManager.SendBatchAsync(lowMessages).Wait();
...

// Send a high priority batch.
var highMessages = new List<BrokeredMessage>();

for (int i = 0; i < 10; i++)
{
  var message = new BrokeredMessage() { MessageId = Guid.NewGuid().ToString() };
  message.Properties["Priority"] = Priority.High;
  highMessages.Add(message);
}

this.queueManager.SendBatchAsync(highMessages).Wait();
```

17

基于队列的负载均衡模式

使用队列，作为一项任务，它调用才能顺利间歇重物，可能会以其他方式导致失败的服务或任务超时服务之间的缓冲区。这个模式可以帮助最小化峰中的可用性和响应需求为任务和服务的影响。

背景和问题

许多解决方案在云中涉及运行调用服务的任务。在这种环境下，如果一个服务进行间歇重物，它可能会导致性能或可靠性问题

一个服务可以是一个组件，它是相同的溶液作为利用它的任务的一部分，或者它可以是第三方服务提供访问经常使用的资源，如高速缓存或存储服务。如果相同的服务是由多个同时运行的任务的使用，它可以是难以预料到的服务可能在任何给定时间点来进行请求的数量。

它可能是一个服务可能会遇到在需求高峰，导致它变得过载和不能对及时响应请求。有大量的并发请求驱服务也可能会导致服务失败，如果它不能处理的论点，即这些请求可能导致。

解决方案

重构的解决方案和介绍的任务和服务之间的队列。任务和服务异步运行。任务帖含有由服务于一个队列所需要的数据的消息。队列作为缓冲，存储该消息，直到它被检索到的服务。该服务从队列中检索消息并进行处理。从多个任务，它可以在一个高度可变的速率产生的请求，可以通过同一个消息队列被传递给服务。图1示出了这种结构。

图1 – 使用队列水平上的服务的负载

队列有效地从服务解耦的任务，并且该服务可以按自己的速度处理从并行任务的请求量的信息无关。此外，不存在延迟到一个任务，如果该服务是不可用的时候它投递一个消息到队列中。

这种模式提供了以下好处：

- 它可以帮助最大限度地提高可用性，因为服务而产生的延迟将不会对应用程序，它可以继续发布消息队列，即使该服务不可用或不正在处理消息的即时和直接的影响。
- 它可以有助于最大化可扩展性，因为队列的数目和服务的数量可以变化，以满足需求。
- 它可以有助于控制成本，因为服务实例的数量部署仅需要足以满足平均负荷，而不是峰值负荷。

注意： 有些服务可以实现节流，如果需求达到阈值，超过该系统可能会失败。节流可能会降低功能可用。你也许可以实现与这些服务负载均衡，以确保这一阈值没有达到。

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 来实现控制的速率服务处理消息，以避免急剧的目标资源的应用程序逻辑是必要的。避免将尖峰需求到该系统的下一个阶段。测试系统在负载下，以确保它提供所需的流平，并调整队列的数目和处理消息来实现该服务实例的数量。
- 消息队列是一个单向的沟通机制。如果一个任务期望的服务的答复，可能有必要执行该服务可用于发送的响应的机制。欲了解更多信息，请参阅异步消息底漆。
- 您一定要小心，如果你申请自动缩放到被监听的队列中的请求服务，因为这可能会导致更多的争夺任何资源，这些服务的份额，并减少使用队列级负载的有效性。

何时使用这个模式

此图案是非常适合于使用可能会受到重载服务的任何类型的应用程序。

这种模式可能不是合适的，如果该应用程序期望以最小延迟的服务的响应。

例子

微软的 Azure Web 角色存储数据使用单独的存储服务。如果大量的 Web 角色实例同时运行，则可能是存储服务可以是不堪重负，无法向请求的速度不够快，以防止超时或没有响应这些请求。图 2 列出了这个问题。

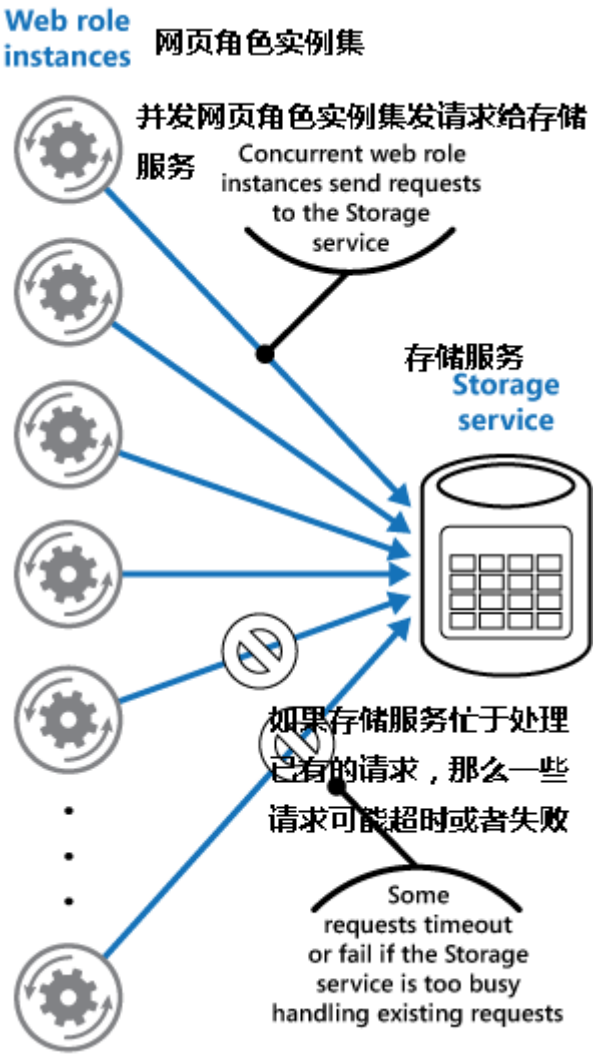


图2 – 服务从一个 Web 角色实例大量并发请求正在被压垮

要解决此问题，可以使用一个队列地级Web角色实例和存储服务之间的负载。但是，存储服务被设计为接受同步请求，并且不能很容易地修改，以读取信息以及管理的吞吐量。因此，可以引入一个辅助角色作为接收从该队列中的请求，并将其转发到所述存储服务的代理服务。在辅助角色的应用程序逻辑可以控制在它传递请求到存储服务，以防止存储服务从被压垮的速率。图3示出了这种解决方案。

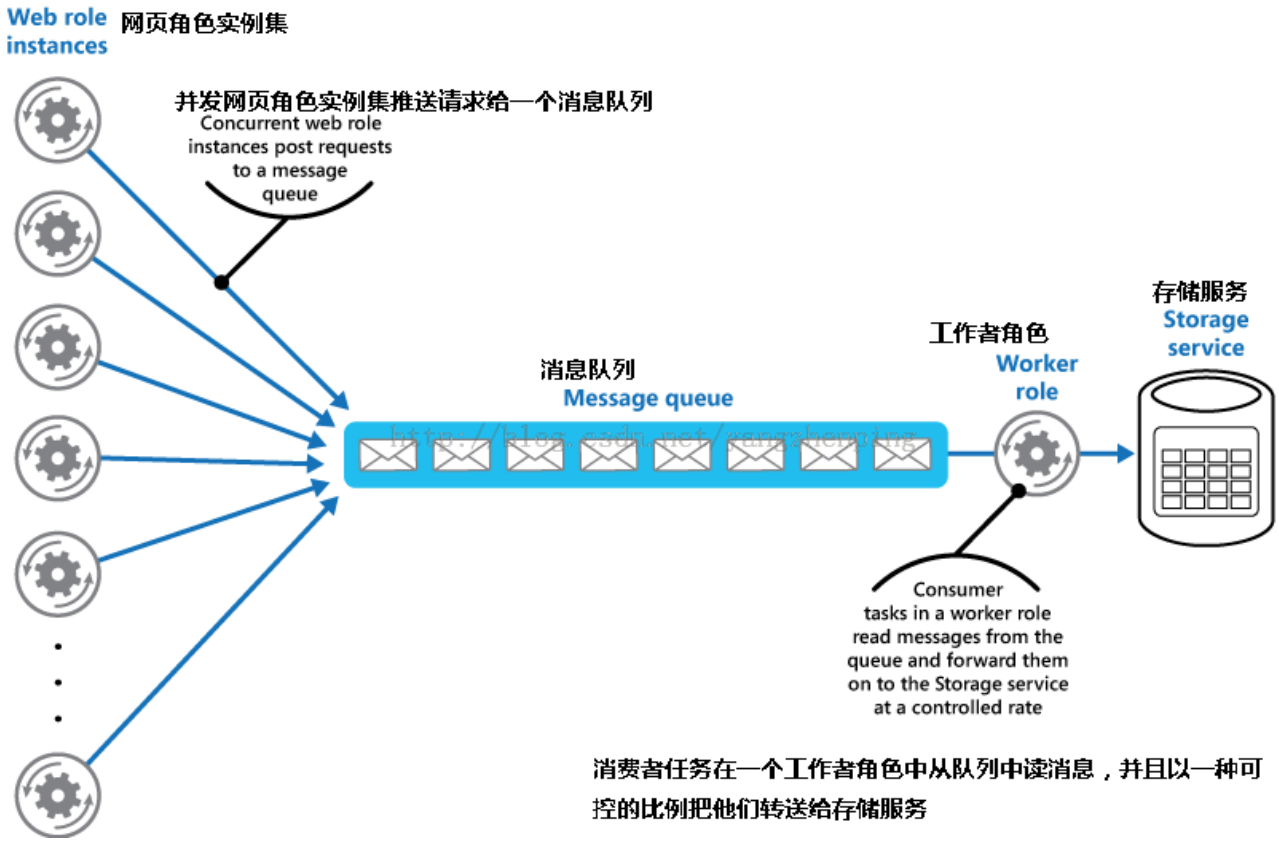


图3 – 使用队列和辅助角色成水平的幅作用和服务实例之间的负载

18

重试模式

启用应用程序来处理预期的，暂时的失败时，它会尝试连接到由透明的重试操作了以前失败的期望，失败的原因是瞬时的服务或网络资源。这种模式可以提高应用程序的稳定性。

背景和问题

该通信的应用程序与在云中运行的元素必须是可能发生在这样的环境中的瞬时故障敏感。这些故障包括网络连接的过程中出现时，一个服务是忙碌的瞬时损失的组件和服务中，服务的临时不可用，或超时。

这些故障一般是自校正的，如果经过一个合适的延迟被重复触发一个故障的动作很可能是成功的。例如，数据库服务，它正在处理大量并发请求可以实现节流策略，暂时拒绝，直到它的工作量有所缓和任何进一步的请求。试图访问该数据库的应用程序可能无法连接，但如果它经过一个合适的延迟再次尝试它可能会成功。

解决方案

在云中，瞬时故障的情况并不少见和应用应该被设计为优雅和透明地处理它们，减少的影响，这种故障可能对应用程序正在执行业务任务。

如果一个应用程序检测到故障时，它试图将请求发送到远程服务，它可以通过使用以下策略处理失败：

- 如果故障指示故障不是瞬时的或不太成功，如果重复（例如，所造成的无效提供凭据的认证失败是不可能成功的，无论是多少次未遂），应用程序应中止操作和报告一个合适的异常。
- 如果报道的具体故障是不寻常的或罕见的，这可能是由于反常的情况，如网络数据包成为损坏，同时它被发送。在这种情况下，应用程序可以再次立即重试失败的请求，因为相同的故障是不可能被重复和请求将可能是成功的。
- 如果故障是由一种更加普遍的连接，或“忙”的失败，网络或服务可能需要在短期内同时连接问题纠正或工作的积压被清除。应用程序应该等待请求重试前一个合适的时间。

对于比较常见的短暂故障，重试期间，应选择以传播从应用程序中尽可能均匀的多个实例的请求。这可以减少繁忙的业务持续过载的可能性。如果一个应用程序的多个实例不断轰击与重试请求的服务，则可能需要该服务更长的时间来恢复。

如果请求仍然失败，应用程序可以等待进一步的时期，再次尝试。如果需要的话，这个过程可以重复而增加重试的延迟，直到请求的某一最大数目已经尝试都失败了。延迟时间可以逐步增加，或可使用的定时策略，如指数回退，取决于故障的性质和可能性，这将在这段时间内被校正。

图1示出了这种模式。如果尝试后的预定数量的请求不成功，应用程序应将故障为异常，并相应地处理它。

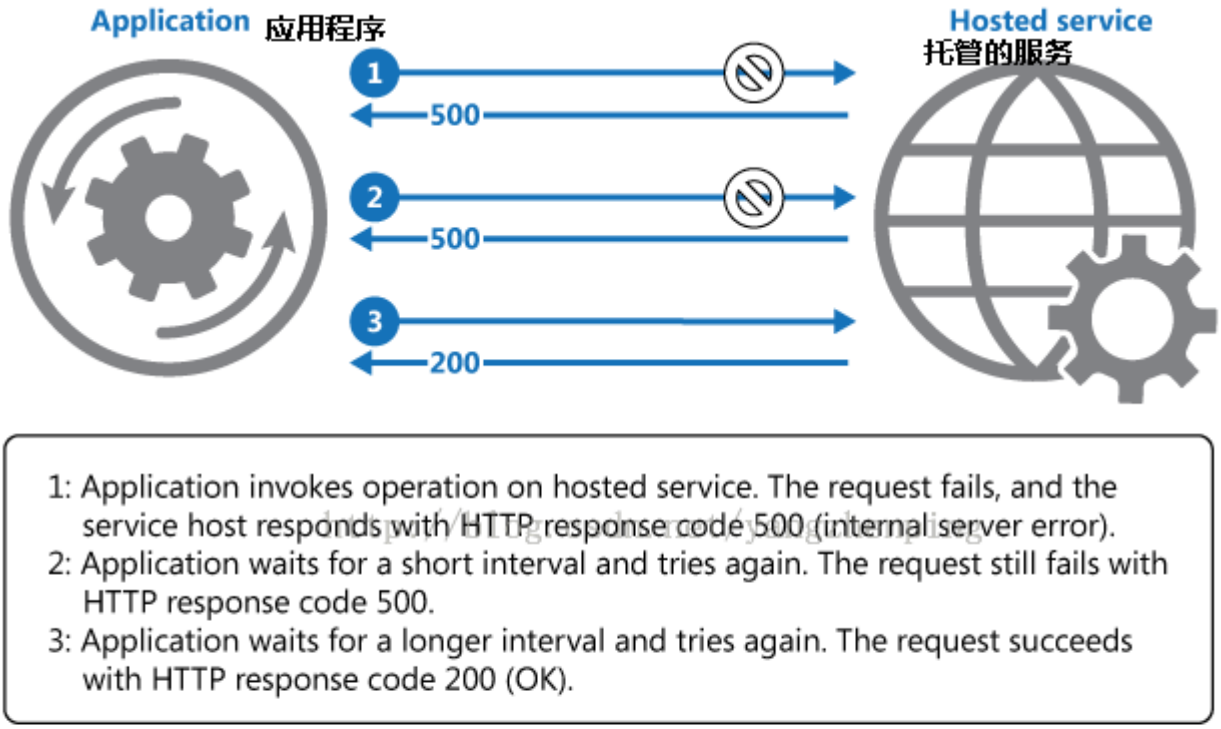


图1 – 使用重试模式中调用托管服务的操作

应用程序应该换所有试图访问远程服务，实现重试政策配套上面列出的策略之一的代码。发送到不同的服务请求会受到不同的政策，有的供应商提供封装这种方法库。这些库通常执行的政策是参数化的，而应用程序开发人员可以指定，如重试次数和重试之间的时间项的值。

在检测故障和重试失败的操作都应该记录这些故障的详细信息的应用程序的代码。这个信息可能是有用的运算符。如果一个服务被频繁报道为不可用或忙，往往是因为该服务已耗尽其资源。则可以减少与这些故障发生时通过换算出该服务的频率。例如，如果数据库服务正在不断超载，它可能是有利的分区数据库和负载分散到多个服务器。

> 注意： 微软 Azure 提供了重试模式的广泛支持。该模式与实践瞬态故障处理块允许应用程序通过一系列的重试策略来处理许多 Azure 服务瞬态故障。微软实体框架版本6提供了用于重新尝试数据库操作。此外，许多在 Azure Service Bus 和 Azure 存储的 API 透明地执行重试逻辑。

问题和注意事项

在决定如何实现这个模式时，您应考虑以下几点：

- 重试政策应进行调整，以满足应用和故障性质的业务需求。它可能是更好一些非关键操作失败快而不是重试几次，并影响应用程序的吞吐量。例如，在试图访问远程服务的交互式Web应用程序，这可能是更好的重试之后用重试之间只有一个短的延迟的数量较少失败，并显示一个适当的消息给用户（例如，“请稍后“），再次尝试阻止应用程序变得反应迟钝。对于批处理应用程序，它可以是更合适的，以增加重试尝试的次数与尝试之间的指数增加的延迟。
- 与尝试之间最小的延迟和大量的重试的高攻击重试的政策，可能会进一步降低正在接近运行或容量的占用。此重试策略也可能会影响应用程序的响应，如果它被不断地在尝试执行失败的操作，而不是做有用功。
- 如果后一个显著次数的重试请求仍然失败，则可能是更好的应用程序，以防止进一步的请求将要在相同的资源为一个周期，并简单地立即报告故障。当期限届满后，该应用程序可以暂时允许通过一个或多个请求，看看他们是否成功。对于这一策略的详细信息，请参阅断路器格局。
- 在由它实现了一个重试策略可能需要为幂等的应用程序调用的服务的操作。例如，发送到服务的请求可以被接收和处理成功，但是，由于瞬时故障，它可能无法发送响应，指示该处理已完成。然后在应用程序的重试逻辑可能试图重复上没有接收到所述第一请求的假定该请求。
- 一个请求到服务失败可能由于各种原因而提出不同的异常，根据故障的性质。一些例外可指示故障，可以非常迅速地得到解决，而另一些可能表明该故障持续时间更长。可能是有益的重试策略，调整基于所述异常的类型的重试尝试之间的时间。
- 考虑如何重试的操作是事务的一部分，会影响整体交易的一致性。这可能是有用的微调对于事务性操作的重试政策，最大限度地取得成功的机会，并减少需要撤消所有交易步骤。
- 确保所有重试代码是完全针对各种故障条件下进行测试。检查它不会严重影响应用程序的性能或可靠性，导致在服务和资源的过度负荷，或产生竞态条件或瓶颈。
- 实现只在一个失败的操作的全方面了解重试逻辑。例如，如果包含的重试策略任务调用另一个任务还包含一个重试策略，这个额外的重试的层可加长的延迟的处理。它可能是更好的配置的低级任务失败快速并报告失败返回调用它的任务的原因。然后这个更高级别的任务可以决定如何处理是根据它自己的策略失效。
- 记录所有的连接故障，提示了重试，使潜在的问题与该应用程序，服务或资源可以被识别是很重要的。
- 研究是最有可能发生于一个服务或资源发现，如果它们有可能是持久或终端的故障。如果是这样的话，它可能是更好地处理该故障为异常。该应用程序可以报告或记录该异常，然后试图通过调用另一个服务，持续或者（如果有一个可用的），或通过提供降级功能。关于如何检测和处理持久故障的更多信息，请参阅断路器格局。

何时使用这个模式

使用这种模式：

- 当一个应用程序可能会经历短暂的故障，因为它与远程服务进行交互，或访问远程资源。这些故障预计将是短暂的，并重复了以前没有能够成功的后续尝试的请求。

这种模式可能不适合：

- 当故障很可能是持久的，因为这可能会影响应用程序的响应性。该应用程序可以简单地是浪费时间和资源试图重复请求是最有可能失败。
- 对于处理故障是不因瞬时故障，如在应用程序的业务逻辑引起错误的内部的异常。
- 作为一种替代解决系统中的可扩展性问题。如果一个应用程序有频繁的“忙”的故障，这是通常指示被访问的服务或资源应相应加大。

例子

本实施例说明的重试模式的实现。该 OperationWithBasicRetryAsync 方法，如下所示，通过 TransientOperationAsync 方法异步调用外部服务（该方法的细节将特定于服务，并从样本代码被省略）。

```
private int retryCount = 3;
...

public async Task OperationWithBasicRetryAsync()
{
  int currentRetry = 0;

  for (; ;)
  {
    try
    {
      // Calling external service.
      await TransientOperationAsync();

      // Return or break.
      break;
    }
    catch (Exception ex)
```

```
    {
      Trace.TraceError("Operation Exception");

      currentRetry++;

      // Check if the exception thrown was a transient exception
      // based on the logic in the error detection strategy.
      // Determine whether to retry the operation, as well as how
      // long to wait, based on the retry strategy.
      if (currentRetry > this.retryCount || !IsTransient(ex))
      {
        // If this is not a transient error
        // or we should not retry re-throw the exception.
        throw;
      }
    }

    // Wait to retry the operation.
    // Consider calculating an exponential delay here and
    // using a strategy best suited for the operation and fault.
    Await.Task.Delay();
  }
}

// Async method that wraps a call to a remote service (details not shown).
private async Task TransientOperationAsync()
{
  ...
}
```

调用此方法的声明被包裹在一个循环一个 try/ catch 块中封装。如果调用 TransientOperationAsync 方法成功，没有抛出异常的 for 循环退出。如果 TransientOperationAsync 方法失败，catch 块检查为失败的原因，并且如果它被认为是一个瞬时错误代码等待一个短暂的延时，然后重试该操作。

在 for 循环还跟踪该操作已经尝试的次数，并且如果代码失败三次异常被认为是更持久。如果该异常是不是暂时的，或者是长久的，catch 处理抛出的异常。此异常退出 for 循环，应捕获调用该OperationWithBasicRetryAsync 方法的代码。

该 IsTransient 方法，如下所示，检查是否有特定的一组是相关的，其中所述代码运行的环境的异常。一过异常的定义可以根据被访问的资源，并在其上执行的操作环境的不同而不同。

```
private bool IsTransient(Exception ex)
{
  // Determine if the exception is transient.
```

```
  // In some cases this may be as simple as checking the exception type, in other
  // cases it may be necessary to inspect other properties of the exception.
  if (ex is OperationTransientException)
    return true;

  var webException = ex as WebException;
  if (webException != null)
  {
    // If the web exception contains one of the following status values
    // it may be transient.
    return new[] {WebExceptionStatus.ConnectionClosed,
                  WebExceptionStatus.Timeout,
                  WebExceptionStatus.RequestCanceled }.
            Contains(webException.Status);
  }

  // Additional exception checking logic goes here.
  return false;
}
```

19

运行重构模式

设计应用程序，使得它可以在不需要重新部署或者重新启动应用程序重新配置。这有助于保持可用性并减少停机时间。

背景和问题

一个主要目的为重要的应用，如商业和企业网站是尽量减少停机时间以及由此引发的中断给客户和用户。但是，有时有必要重新配置应用程序改变特定行为或设置，而在部署和使用。因此，它是用于该应用程序被设计成这样一种方式，以允许在运行时要应用这些配置的变化，并为应用程序，以检测所述变化并且尽快地应用它们的部件的优点。

该种要应用可能被调整记录，以协助与应用程序调试问题，交换使用不同数据存储的连接字符串，或者打开或关闭特定的部分或应用程序的功能的粒度配置变化的例子。

解决方案

为实施这一模式的解决方案依赖于应用程序托管环境中可用的功能。典型地，应用程序代码将响应于由检测到的变化对应用程序配置时，主机基础设施提出的一个或多个事件。这通常是上载新的配置文件，或响应于改变通过管理门户的配置或者通过访问的API的结果。

码处理的配置变化事件可以检查变化，并将其应用到该应用程序的组件。有必要对这些部件进行检测和反应的变化，因此它们的值通常会被公开为可写的属性或方法，在事件处理程序的代码可以设置为新值，或执行。从这一点来说，该部件应使用新的值，以便在需要改变应用程序的行为发生。

如果这是不可能的部件，以应用更改在运行时，这将是必要的，重新启动该应用程序，从而当应用程序启动时再次这些更改应用。在一些托管环境中它可能会检测到这些类型的变化，并指出对环境的应用程序必须重新启动。在其他情况下，可能有必要执行该分析的设置更改，并强制必要的应用程序重新启动时的代码。

图1示出了本模式的概述。

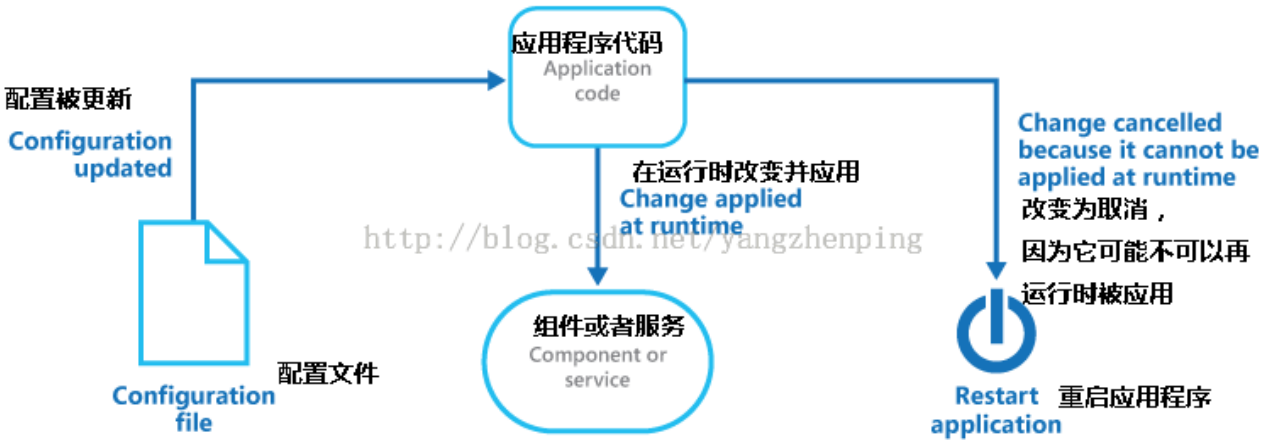


图1 – 此模式的基本概述

大多数环境中暴露响应配置更改引发的事件。在那些不这样做，定期检查更改配置并应用这些变化将是必要的轮询机制。它也可能有必要重新启动应用程序，如果变化不能在运行时被应用。例如，有可能以比较在预设的时间间隔一个配置文件的日期和时间，并运行代码以应用更改时的较新版本中找到。另一种方法是，其中包含一个控制中的应用程序的管理用户界面，或使一个安全端点可以从应用程序外部进行访问，其执行读取，并应用更新的配置的代码。

或者，该应用程序可以反应以在环境中的一些其他变化。例如，发生于特定的运行时错误可能会改变日志配置自动收集更多的信息，或者代码可以使用当前日期读取和应用主题，反映了季节或特殊事件。

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 配置设置必须存储在部署的应用程序之外，使得它们可以在不需要整个包被重新部署更新。典型的设置被存储在配置文件中，或者在外部存储库中，如一个数据库或网络存储。访问运行时配置机制，应严格控制，以及使用时的严格审核。
- 如果托管的基础设施不会自动检测配置更改的事件，揭露这些事件对应用程序代码，您必须实现一种替代机制来检测和应用更改。这可以是通过轮询机制，或者通过暴露交互式控制或端点发起更新过程。
- 如果您需要实现一个轮询机制，考虑如何经常检查更新的配置应该发生。长轮询间隔将意味着变化可能不被应用了一段时间。短的间隔可能会产生不利影响，通过吸收现有的计算和I / O资源的操作。
- 如果是应用程序的多个实例，附加的考虑因素，这取决于如何变化进行检测。如果改变是通过由宿主基础结构引发的事件自动检测到，这些变化可能不被同时应用的所有实例进行检测。这意味着，某些情况下，将要使用的原始配置为一个周期，而有些则使用新的设置。如果该更新是通过轮询机制检测到，这必须以保持一致性通信改变到所有实例。
- 一些配置的变化可能要求应用程序重新启动，甚至要求托管服务器重新启动。您必须确定这些类型的配置设置和执行的每一个相应的操作。例如，要求应用程序重新启动的变化可能会自动执行此操作，或者它可能是管理员负责发起重新启动在适当的时间时，应用过大的负荷和应用程序可以处理的其他实例下是不的负载。
- 更新并确认他们是成功的，而更新的应用程序实例正在执行正确，将更新应用到所有实例之前的分阶段部署计划。由此，能够防止应发生错误的应用程序的总的中断。凡更新需要重新启动或应用程序的重新启动，特别是在应用程序有一个显著启动或热身的时候，用一个分阶段部署的方式，以防止多个实例脱机在同一时间。
- 考虑如何将回滚造成的问题配置更改，或导致申请失败。例如，它应该能够滚动的等待轮询间隔，以检测所述变化背部的变化立即代替。

- 考虑如何配置设置的位置可能会影响应用程序的性能。例如，你应该处理将发生，如果您使用外部存储不可用的错误，当应用程序启动时，或配置更改将被应用，比如用一个默认的配置或通过本地缓存的设置在服务器和重用这些值而重试访问远程数据存储。
- 高速缓存可以帮助减少延迟，如果一个组件需要多次访问配置设置。然而，当配置改变时，应用程序代码将需要无效缓存设置，该组件必须使用更新后的设置。

何时使用这个模式

这种模式非常适合于：

- 应用程序，而您必须避免一切不必要的停机时间，同时仍然能够将更改应用到应用程序配置。
- 环境，揭露事件自动提出的主要配置更改时。通常，这是当检测到一个新的配置文件，或者更改了现有的配置文件。
- 应用的地方，往往配置更改和变化可以应用于组件，而不要求应用程序重新启动，或无需托管服务器必须重新启动。

这种模式可能不是合适的，如果运行时组件的设计使得它们只能在初始化时被配置，并更新这些部件的努力不能相比，重新启动应用程序和持久的一个短的停机时间是合理的。

例子

微软 Azure 云服务的角色发现和揭露被提了两个事件，当主机环境检测变化的 ServiceConfiguration.cscfg 文件：

- RoleEnvironment.Changing。引发此事件被检测到的结构变化后，但在此之前它被施加到该应用程序。你可以处理查询的变化，并取消运行时重新配置的活动。如果取消了变化，网页或辅助角色将自动以使新配置被应用程序使用的重新启动。
- RoleEnvironment.Changed。引发此事件后，应用程序的配置得到了应用。可以处理该事件来查询所应用的改变。

当取消在 RoleEnvironment.Changing 事件改变要表示到 Azure，一个新的设置不能被应用于该应用程序正在运行时，并且它必须以使用新的值被重新启动。有效地，你会取消更改只有在您的应用程序或组件无法反应在运行时改变，需要重新启动才能使用新的值。

> 注意： 欲了解更多信息，请参阅 RoleEnvironment.Changing 事件并使用 RoleEnvironment.Changing 事件 MSDN 上。

处理 RoleEnvironment.Changing 和 RoleEnvironment.Changed 事件，你通常会添加一个自定义处理该事件。例如，从你可以下载本手册的例子运行时重新配置的解决方案的 Global.asax.cs 类下面的代码显示了如何添加一个名为 RoleEnvironment_Changed 到事件处理链中的自定义函数。这是从实施例的的 Global.asax.cs 文件。

> 注意： 这种模式的例子是，在 RuntimeReconfiguration 解决方案的 RuntimeReconfiguration.Web 项目。

```
protected void Application_Start(object sender, EventArgs e)
{
  ConfigureFromSetting(CustomSettingName);
  RoleEnvironment.Changed += this.RoleEnvironment_Changed;
}
```

在 Web 或工作的角色，你可以在处理 RoleEnvironment.Changing 事件的作用的 OnStart 事件处理程序中使用类似的代码。这是从实施例的 WebRole.cs 文件。

```
public override bool OnStart()
{
  // Add the trace listener. The web role process is not configured by web.config.
  Trace.Listeners.Add(new DiagnosticMonitorTraceListener());

  RoleEnvironment.Changing +=  this.RoleEnvironment_Changing;
  return base.OnStart();
}
```

要注意的是，在网页的角色的情况下，所述的 OnStart 事件处理程序中从 Web 应用程序本身的单独进程中运行。这就是为什么你通常会处理在 Global.asax 文件中 RoleEnvironment.Changed 事件处理程序，让您可以更新您的 Web 应用程序的运行时配置，而 RoleEnvironment.Changing 事件中的角色本身。在辅助角色的情况下，您可以订阅双方 RoleEnvironment.Changing和RoleEnvironment.Changed 的 OnStart 事件处理程序中的事件。

> 注意： 可以在服务配置文件中存储自定义的配置设置，在自定义配置文件，在数据库中，如在虚拟机中的 Azure SQ L数据库或 SQL Server，或者在天青 blob 和表存储。您需要创建一个可以访问自定义配置设置和应用程序内设置组件的属性，这些适用于应用程序通常代码。

例如，下面的自定义函数读取设置，其名称作为参数传递，从 Azure 的服务配置文件中的值，然后将它应用到一个名为 SomeRuntimeComponent 运行时组件的当前实例。这是从实施例的的 Global.asax.cs 文件

```
private static void ConfigureFromSetting(string settingName)
{
  var value = RoleEnvironment.GetConfigurationSettingValue(settingName);
  SomeRuntimeComponent.Instance.CurrentValue = value;
}
```

注意

一些配置设置，如那些用于 Windows 标识框架，不能存储在 Azure 服务配置文件中，并且必须在 App.config 或 Web.config 文件。

在Azure中，一些配置的变化检测，并自动应用。这包括在 Diagnostics.wadcfg 文件寡妇天青诊断系统，它指定的信息类型来收集和如何保持日志文件的结构。因此，它仅需要编写处理添加到服务配置文件的自定义设置的代码。你的代码应该：

- 从更新的配置应用自定义设置您的应用程序在运行时的相应组件，使他们的行为体现了新的配置。
- 取消改变，以指示到 Azure 新的值不能在运行时应用，该应用程序必须按顺序重新开始对要应用的变化。

例如，从你可以下载本手册的例子运行时重新配置的解决方案 WebRole.cs 类下面的代码显示了如何使用 RoleEnvironment.Changing 事件取消所有设置的更新，除了可应用于那些在运行时，不需要重新启动。此示例允许在运行时应用无需重新启动应用程序（使用此设置将能够读取新的值，并相应地在运行时改变其行为的组成部分）更改为“CustomSetting”的设置。任何其他更改的配置将自动使网页或工作的角色重新启动。

```
private void RoleEnvironment_Changing(object sender,
                        RoleEnvironmentChangingEventArgs e)
{
  var changedSettings = e.Changes.OfType<RoleEnvironmentConfigurationSettingChange>()
                         .Select(c => c.ConfigurationSettingName).ToList();
  Trace.TraceInformation("Changing notification. Settings being changed: "
                    + string.Join(", ", changedSettings));

  if (changedSettings
    .Any(settingName => !string.Equals(settingName, CustomSettingName,
                        StringComparison.Ordinal)))
  {
    Trace.TraceInformation("Cancelling dynamic configuration change (restarting).");

    // Setting this to true will restart the role gracefully. If Cancel is not
    // set to true, and the change is not handled by the application, the
    // application will not use the new value until it is restarted (either
    // manually or for some other reason).
    e.Cancel = true;
  }
  Else
  {
    Trace.TraceInformation("Handling configuration change without restarting. ");
  }
}
```

注意： 这种方法证明了好的做法，因为它确保了更改应用程序代码不知道任何设置（因此不能确保它可以在运行时应用）将导致重新启动。如果更改任何一个被取消，该角色将被重新启动。

然后可以检测到并应用于应用程序的组件的新的配置后已被接受由 Azure 的框架更新未在 RoleEnvironment.Changing 事件处理程序取消。例如，在该示例解决方案的 Global.asax 文件以下代码处理 RoleEnvironment.Changed 事件。它检查每个配置设置，并且当它找到名为“CustomSetting”的设置，调用一个函数（前面所示），该应用新的设置，以在应用程序中的适当组件。

```
private void RoleEnvironment_Changed(object sender,
                        RoleEnvironmentChangedEventArgs e)
{
  Trace.TraceInformation("Updating instance with new configuration settings.");

  foreach (var settingChange in
         e.Changes.OfType<RoleEnvironmentConfigurationSettingChange>())
  {
    if (string.Equals(settingChange.ConfigurationSettingName,
                  CustomSettingName,
                  StringComparison.Ordinal))
    {
      // Execute a function to update the configuration of the component.
      ConfigureFromSetting(CustomSettingName );
    }
  }
}
```

需要注意的是，如果你不取消配置的变化，但不将新值应用到您的应用程序组件，那么更改将不会生效的下一次重新启动应用程序之前。这可能会导致不可预测的行为，尤其是当所述宿主角色实例由 Azure 的自动重启在其日常维护操作，在该点的新的设定值将被应用的一部分。

20

调度程序代理管理者模式

协调一系列在分布式服务集和其他远程资源的的行为，试图透明地处理故障，如果这些操作失败，或撤销，如果系统不能从故障中恢复执行工作的影响。这种模式可以分布式系统中增加弹性和灵活性，使之恢复和重试失败是由于短暂的异常，持久的故障和处理故障等操作。

背景和问题

应用程序执行其包括多个步骤，其中的一些可以调用远程服务或访问远程资源的任务。各个步骤可以是相互独立的，但它们是由实现该任务的应用程序逻辑编排。

只要有可能，应用程序应该确保任务运行完成和解决远程访问服务或资源时可能发生的任何故障。可能会因各种原因，这些故障。例如，该网络可能是崩溃，通信可能被中断，远程服务可能停止响应或处于不稳定的状态，或远程资源可能暂时无法访问，可能由于资源约束。在许多情况下，这些故障可能是暂时的，并且可以通过使用重试模式进行处理。

如果该应用程序检测到一个更永久的故障，从它可以不容易恢复，它必须能够将系统恢复到一致状态，并确保整个端至端的操作的完整性。

解决方案

调度代理管理者模式定义了以下角色。这些演员编排的步骤（工作，单个项目）将作为任务（整个过程）的一部分进行：

- 调度安排构成的整体任务要执行，并配合它们的操作的各个步骤。可按照下列步骤组合成一个管道或工作流，并且所述调度器是负责确保在这个工作流程中的步骤，以适当的顺序被执行。作为各步骤中进行（如“步骤还未开始”，“步运行时，”或“步骤完成”），并记录有关该状态的信息的调度器维护关于工作流的状态信息。这个状态信息也应包括的允许的步骤来完成（称为完全按时间）的时间的上限。如果一个步骤需要访问远程服务或资源时，调度程序调用适当的代理程序，通过它的工作的细节将被执行。调度通常采用异步请求/响应消息与代理进行通讯。这可以通过使用队列来实现，尽管也可以使用其它分布式消息传递技术来代替。

> 注意： 调度程序执行类似的功能，以在流程管理模式的过程管理器。实际工作流程通常被定义并通过由该调度器所控制的工作流引擎来实现。这种方法分离了业务逻辑从调度的工作流程。

- 代理包含逻辑，封装调用一个远程服务，或访问由在一个任务的步骤中引用的远程资源。每个代理通常通过它可以调用单个服务或资源，实施相应的错误处理和重试逻辑（如有超时限制，稍后说明）。如果由调度程序所运行的工作流中的步骤利用在不同步骤的若干服务和资源，每个步骤可能会引用不同代理（这是其中的模式的实现细节）。

- 监监视由调度正在执行中的任务步骤的状态。它运行周期（频率将系统专用），检查的步骤，通过调度为保持状态。如果检测到任何已超时或失败，它会安排相应的代理来恢复步骤或执行相应的补救措施（这可能涉及修改步骤的状态）。注意，恢复或补救行动通常由调度器和代理执行。主管应该简单地要求这些行动来执行。

调度程序，代理和管理者是逻辑组件和它们的物理实现取决于所使用的技术。例如，若干个逻辑代理可以被实现为一个单一的网络服务的一部分。

调度器维护关于任务的进度和持久的数据存储中的每个步骤中，被称为状态存储的状态信息。主管可以使用此信息来帮助确定一个步骤是否出现故障。图1说明了调度程序的代理，监事和状态存储之间的关系。

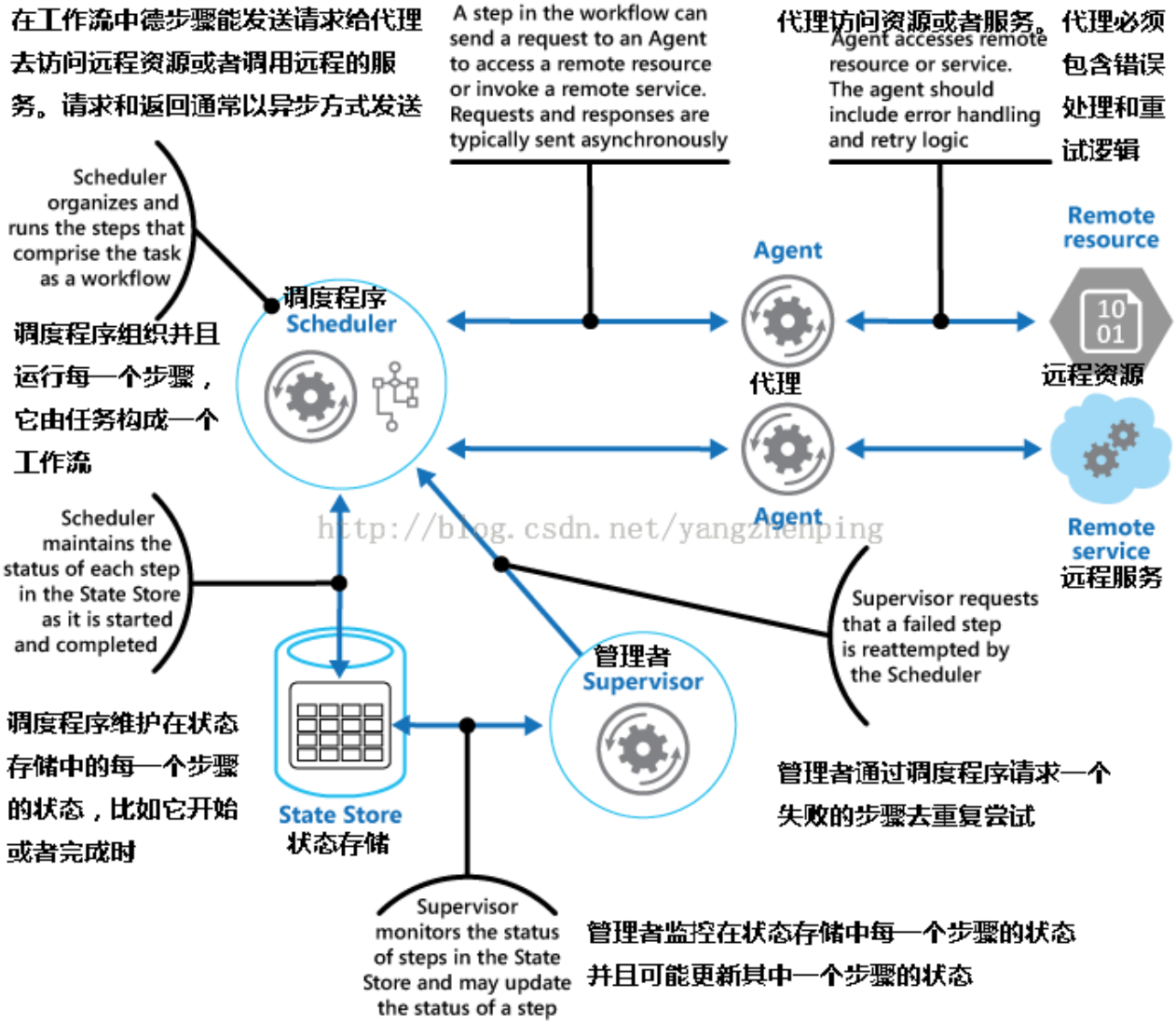


图1 – 在调度程序代理管理者模式的演员

注意：

此图显示的模式的简化图。在实际的实现中，有可能是调度程序的多个实例同时运行的任务的每个子集。类似地，系统可以运行每个代理程序的多个实例，或者甚至多个监控器。在这种情况下，主管必须协调其与对方认真的工作，以确保它们不会争来恢复同样失败的步骤和任务。在领导人选举模式提供一个可能的解决了这个问题。

何时一个应用程序希望执行一个任务，它提交一个请求给调度程序。调度记录有关的任务及其步骤（例如，“步骤还未开始”）中的状态存储的初始状态的信息，然后开始执行由流程定义的操作。作为调度程序开始每一步时，它将更新关于该步骤中的状态存储（例如，“步运行的”）的状态的信息。

如果一个步骤引用远程服务或资源时，调度程序将消息发送给适当代理。该消息可以包含代理需要传递给服务或访问该资源，除了完整的通过时间的操作的信息。如果代理成功完成其操作时，它返回到调度程序的响应。然后调度程序可以更新在状态存储的状态信息（例如，“步骤完成”），并进行下一步骤。这个过程继续，直到整个任务完成。

代理可以实现任意重试逻辑需要执行它的工作。但是，如果该代理之前，完成未完成其工作期间届满的调度程序就会认为操作失败。在这种情况下，代理应该停止其工作，并没有试图东西返回到调度程序（甚至没有错误消息），或者进行任何形式的恢复。这样做的原因的限制是，后一步骤中已超时或失败，则代理的另一个实例可被调度来运行失败的步骤（该过程将在后面描述）。

如果代理本身出现故障时，调度程序将不会收到回复。该模式可能不会让这一步骤已超时，一个已经失败的真正区别。

如果一个步骤超时或失败时，状态存储将包含一个记录，指出该步骤是否正在运行（“步骤运行”），但完全通过时间已经过去了。监事查找步骤，如本，并试图恢复它们。一个可能的策略是为超级更新完成由值来扩展可用来完成步骤的时间，然后将消息发送到调度器识别已超时的步骤。然后，调度程序可以尝试重复此步骤。然而，这样的设计要求为幂等的任务。

这可能是必需的管理者防止如果连续失败或者超时被重试相同步骤。为了实现这一点，对管理可维持一个重试计数为每一步，随着状态信息，在该状态存储。如果该计数超过预定阈值的管理者可以采用一种策略，例如，通知它应重试步骤中，在期望的故障将在这段时间内可以解决调度之前等待较长的时间。或者，该管理者可以将消息发送到调度请求将整个任务通过实现补偿交易被撤消（该方法将依赖于调度程序和代理提供所必需的信息，以实现对已成功完成各步骤的补偿操作）。

注意：

这不是管理者的目的来监控调度程序和代理商，如果他们不能重新启动它们。系统的这方面应该由其中这些组件所运行的基础设施进行处理。同样，管理者不应该是由调度所执行的任务正在运行（包括如何补偿应这些任务失败）的实际业务运作的知识。这是由调度程序执行的工作流逻辑的目的。监事的责任是确定的步骤是否已失败，并安排要么为它重复或者包含失败的步骤，整个任务被取消。

如果该调度程序是失败的，或者工作流程后，重新启动正在由调度程序进行意外终止，该调度程序应能确定任何飞行任务，这是处理失败时的状态，并准备继续这个任务从该点上失败了。该方法的实现细节都可能是特定的系统。如果任务不能恢复，这可能是必要的，以撤消该任务已经完成的工作。这可能还需要执行一个补偿事务。

这种模式的主要优点是，该系统是弹性的意想不到的临时或不可恢复故障的情况下。该系统可以构造成可自愈。例如，如果一个代理程序或调度程序崩溃时，一个新的可启动的，因而管理可以安排要恢复的任务。如果管理者发生故障，另一实例可以启动，并且可以从发生故障的接管。如果管理者计划定期运行，一个新的实例可以被自动预定义的时间间隔后启动。该状态存储可以被复制以实现更大程度的弹性。

问题和注意事项

在决定如何实现这个模式时，您应考虑以下几点：

- 该图案可以是平凡的执行，并且需要的系统的每个可能的故障模式的全面测试。
- 通过调度实现的恢复/重试逻辑可以是复杂的并且依赖于状态存储保持状态信息。它也可能是必要的，记录在一个持久的数据存储区执行一个补偿事务处理所需的信息。
- 与监事运行是非常重要的频率。它应该运行足够频繁，以防止任何失败堵塞长时间的应用程序的步骤，但它不应该运行非常频繁，它成为一个开销。
- 由代理执行的步骤可以被执行一次以上。实现这些步骤的逻辑应该是幂等。

何时使用这个模式

使用这种模式时，在分布式环境中运行，例如云的方法必须是有弹性的，以通信故障和/或运行故障。

这种模式可能不适合任务不调用远程服务或访问远程资源。

例子

实现一个电子商务系统中的 Web 应用程序已经部署在微软的 Azure。用户可以运行此应用程序来浏览提供一个组织的产品，下订单，这些产品。用户接口运行作为一个网络的作用，并且该应用程序的命令处理元件被实现为一组工作角色。订单处理逻辑的一部分包括访问远程服务，该系统的这一方面可能是易发生的瞬态或更持久的故障。为此，设计师使用了调度程序代理管理者模式实现了系统的订单处理单元。

当客户下订单时，应用程序构建了一个消息，说明的顺序和该职位的消息到队列中。一个单独的提交过程中，工人的角色运行，检索此消息，将订单到订单数据库的详细信息，并为在国家商店的订单流程的记录。请注意，插入到常规数据库和国家存储作为同一操作的一部分执行。提交过程的设计，以确保两个刀片共同完成。

该提交进程创建的订单包括状态信息：

- 订单ID：在订单数据库中的订单的 ID。
- LockedBy：的辅助角色的处理顺序的实例 ID。有可能是运行调度程序的工人角色的多个电流情况下，但每个订单只能通过一个实例来处理。
- CompleteBy：通过该命令应处理的时间。
- ProcessState：任务处理订单的当前状态。可能的状态是：
- Pending：的顺序已被创建，但处理尚未启动。
- Processing：该命令正在处理中。
- Processed：订单已成功处理。
- Error：订单处理失败。
- FailureCount：次数的处理已经尝试了顺序的号码。

在这种状态信息，OrderID 字段从新订单的订单 ID 复制。该 LockedBy 和 CompleteBy 字段设置为 null，则 ProcessState 字段设置为待定，并且 FailureCount 字段设置为 0。

注意：

在这个例子中，为了处理逻辑比较简单，只包括一个调用的远程服务的单个步骤。在一个更复杂的多步骤的情况下，提交过程很可能涉及多个步骤，所以多个记录将在状态存储，每一个描述了一个单独的步骤中的状态被创建。

该调度程序同时作为一个辅助角色的一部分，实现了处理订单的业务逻辑。调度轮询新订单的一个实例探讨了状态存储的记录，其中 LockedBy 字段为空，并在 ProcessState 场待定。何时调度程序发现一个新的订单，立刻填充LockedBy 字段与它自己的实例 ID，设置 CompleteBy 字段到一个适当的时间，并设置 ProcessState 字段来处理。执行此代码的设计是独特和原子，以确保调度程序的两个并发实例不能试图同时处理的顺序相同。

调度程序将运行业务流程处理订单异步，从状态存储传递给它的值在 OrderID 字段。工作流处理的顺序检索来自数据库的订单订单的详细信息，并执行其工作。当订单处理流程的步骤需要调用远程服务，它使用一个代理。在工作流步骤的代理通过利用对作为请求/响应信道 Azure 的服务总线消息队列进行通信。图2示出了该解决方案的一个高层视图。

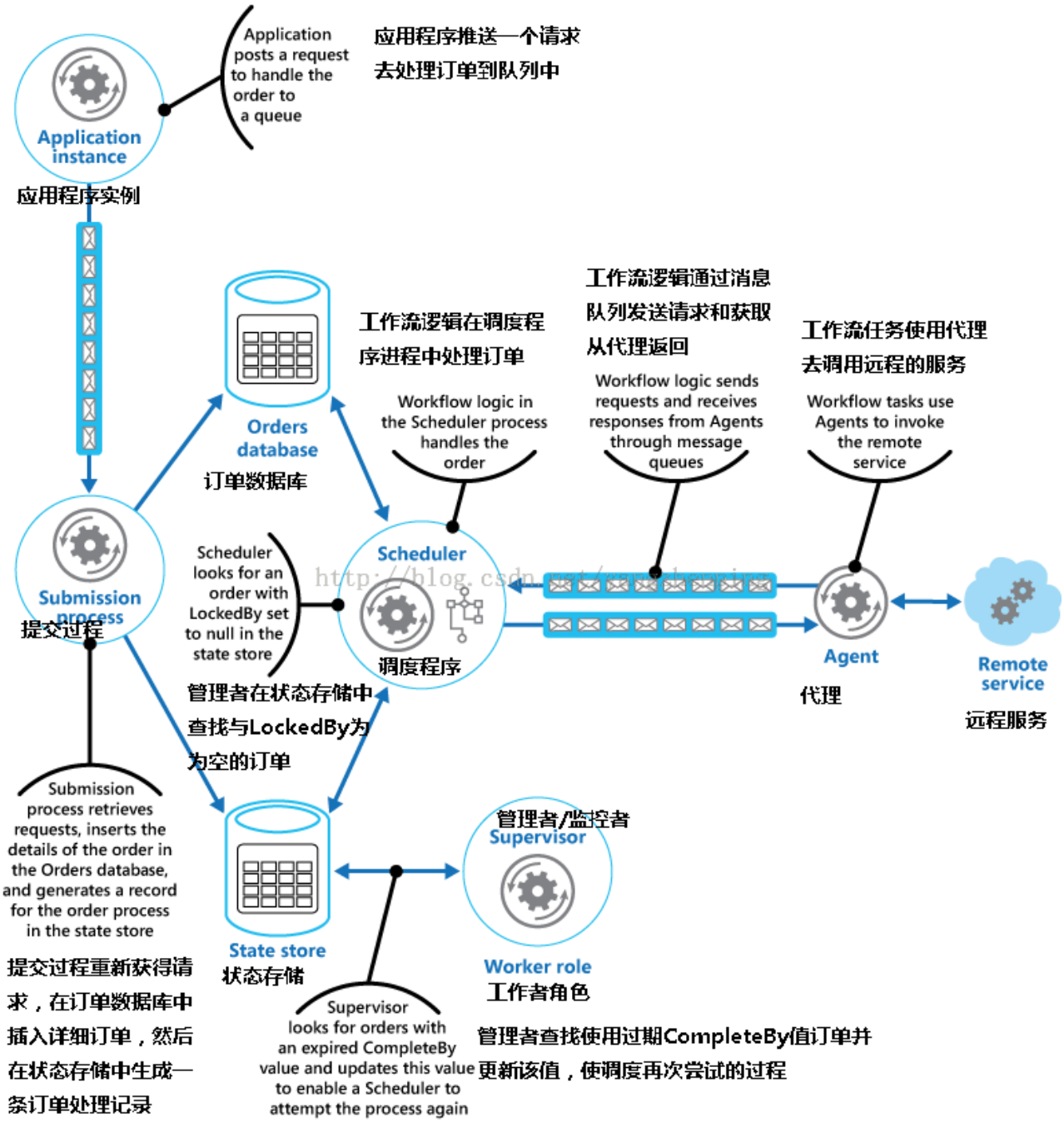


图2 – 使用调度程序代理管理者模式在 Azure 的解决方案来处理订单

从一个工作流步骤发送到代理的消息描述的顺序，并包括 CompleteBy 时间。如果代理接收来自远程服务的响应的 CompleteBy 时间到期之前，构建其上的服务总线队列在其上的工作流是听张贴答复消息。当工作流步骤接收到有效的应答消息，它完成它的处理和调度台的订单状态 ProcessState 场处理。在这一点上，为了处理已成功完成。

如果 CompleteBy 时间到期的代理接收来自远程服务的响应之前，代理简单地停止其处理，并终止处理顺序。类似地，如果工作流处理的顺序超过了 CompleteBy 时，它也将终止。在这两种情况下，在该状态存储在顺序的状

态保持给定处理中，但 CompleteBy 时间指示的时间用于处理订单已经过去，该处理判定为不合格。请注意，如果正在访问远程服务，或者正在处理的顺序工作流（或两者）的代理意外终止，在状态存储的信息将再次保持设置为处理，最终将有过期 CompleteBy 值。

如果代理检测到不可恢复的非瞬时性故障当它正在尝试联系远程服务，它可以发送一个错误响应返回到工作流。该调度程序可以设置为错误，并提出了警示操作者的事件的状态。然后，操作者可以尝试手动解决失败的原因，并重新提交失败的处理步骤。

主管定期检查状态存储在寻找订单，过期 CompleteBy 值。如果管理者发现这样的记录，它增加了 FailureCount 领域。如果 FailureCount 值低于规定的阈值时，所述管理者复位 LockedBy 字段为空，更新 CompleteBy 场与新的到期时间，并设置 ProcessState 字段待定。调度程序的一个实例可以拿起这个命令并执行其处理如前。如果 FailureCount 值超过特定阈值时，对故障的原因被假定为非瞬态。监事设置为错误的状态，并引发了警报操作，如前所述的事件。

注意： 在这个例子中，管理者是在一个单独的工作任务落实。您可以使用各种策略来安排监理任务运行，包括使用 Azure 的计划程序服务（不要与计划程序组件在此模式相混淆）。关于Azure的计划程序服务的更多信息，请访问调度程序页面。

虽然未在本实施例中所示，该调度程序可能需要保留在通知关于单的进度及状态首位提交的顺序应用。应用程序和调度程序彼此分离，以消除它们之间的任何相关性。该应用程序并不知道哪个调度的实例处理的顺序，调度不知道它的具体应用实例发布的顺序。

为使订单状态予以报道，该应用程序可以使用自己的私人响应队列。这个响应队列的详细信息将被纳入送往提交过程的要求，其中包括在状态存储这些信息的一部分。该调度程序随后将邮件投递到该队列表示订单的状态（“接收到的请求”，“为了完成”，“订单失败”，等等）。它应包括订单ID在这些消息中，以便它们可以与由该应用程序的原始请求相关联。

21

Sharding 分片模式

将一个数据存储到一组水平分区或碎片。存储和访问大量数据时，这个模式可以提高可扩展性。

背景和问题

由一个单一的服务器托管的数据存储区可能会受到以下限制：

- 存储空间。一种数据存储为一个大型云应用可以预期含有数据量巨大，可以随着时间的推移显著增加。服务器通常提供的磁盘存储仅是有限的，但它可以是能与较大的取代现有的磁盘，或者添加另外的磁盘的机器作为数据量的增加。然而，由此，不能够容易地增加一个给定的服务器上的存储容量的系统最终将达到一个硬限制。
- 计算资源。云应用程序可能需要支持大量并发用户，每一个运行检索的数据存储信息的查询。一个单一的服务器托管的数据存储可能无法提供所需的计算能力，以支持该负载，从而延长反应时间，为用户和故障频作为应用程序试图存储和检索数据超时。它可能会增加存储器或升级的处理器，但是当其不能够提高计算资源的任何进一步的系统将达到极限。
- 网络带宽。最后，在单个服务器上运行的数据存储区的性能是通过在该服务器可以接收请求并发送回复率的约束。这是可能的网络流量的量可能会超过用于连接到该服务器，从而导致失败的请求的网络的容量。
- 地理。可能需要为存储由特定的用户在同一个区域中产生的那些用户为合法，合规性，或性能原因，数据，或减少数据访问延滞。如果用户在不同的国家或地区的分散，也未必能够存储整个数据为在一个单一的数据存储区中的应用。

垂直缩放通过添加更多的磁盘容量，处理能力，内存和网络连接可能会推迟一些这些限制的效果，但它很可能是只是一个临时的解决方案。能够支持大量用户和大量数据的商业云应用程序必须能够扩展几乎无限，所以垂直缩放不一定是最好的解决方案。

解决方案

划分数据存储到水平分区或碎片。每个碎片都有相同的模式，但保存的数据其独特的子集。甲碎片是在自己的权利（它可以包含许多不同类型的实体的数据）的数据存储器，用作存储节点的服务器上运行。

这种模式具有以下优点：

- 您可扩展系统，通过添加额外的存储节点上运行的进一步碎片。
- 系统可以使用现成的商品硬件，而不是专门的（和昂贵）的计算机为每个存储节点。
- 您可以通过平衡跨越碎片的工作量减少争用和改进的性能。
- 在云中，碎片可以位于物理上接近将要访问数据的用户。

何时将一个数据存储成碎片，决定哪些数据应该被放置在每个碎片。甲碎片通常包含倒在数据中的一个或多个属性所确定的指定范围内的物品。这些属性形成的分片密钥（有时也被称为分区键）。分片的关键应该是静态的。它不应该根据可能发生变化的数据。

分片身体组织的数据。何时一个应用程序存储和检索数据，该分片的逻辑指示应用到相应的碎片。此拆分逻辑可在应用程序中被实现为数据访问代码的一部分，或者它可以由数据存储系统中实现，如果它透明地支持分片。

抽象的拆分逻辑的数据的物理位置提供了一个高层次的控制在其上的碎片包含的数据，并且使数据分片之间迁移，而再加工的应用程序的业务逻辑应该需要在碎片的数据后面将要引入的（例如，如果碎片变得不平衡）。的折衷是在确定的每个数据项的位置，因为它被检索所需的附加数据存取的开销。

为了确保最佳的性能和可扩展性，分裂的数据的方式，适合的查询类型的应用程序执行是重要的。在许多情况下，这是不可能的拆分计划将完全符合每个查询的要求。例如，在一个多用户系统中的应用，可能需要通过使用承租者ID来检索租户数据，但它也可能需要基于其它属性这个数据查找，如承租人的名称或位置。为了处理这些情况，实行分片策略，即支持最常用的查询执行的一个子库的关键。

如果查询通过使用属性值的组合来定期检索数据，这可能是可以定义通过将属性一起的复合分片键。另外，使用模式，如索引表，以提供快速的查找到未覆盖的碎片关键基础属性数据。

分片策略

选择所述分片密钥，并决定如何跨碎片分发数据时的三种策略是常用的。注意，并不需要成为碎片和承载它们 – 在单个服务器可以承载多个分块中的服务器之间的一对一的对应关系。这些战略包括：

- 查找策略。在上述策略中，分片的逻辑实现了一个图，路由对数据的请求到包含碎片，通过使用分片密钥数据。在一个多用户应用将租户的所有数据可能通过使用租户 ID 作为分片密钥一起存储在一个碎片。多个租户可能共享相同的碎片，但对于单个租户的数据将不被跨越多个碎片散布。图1示出了这种策略的一个例子。

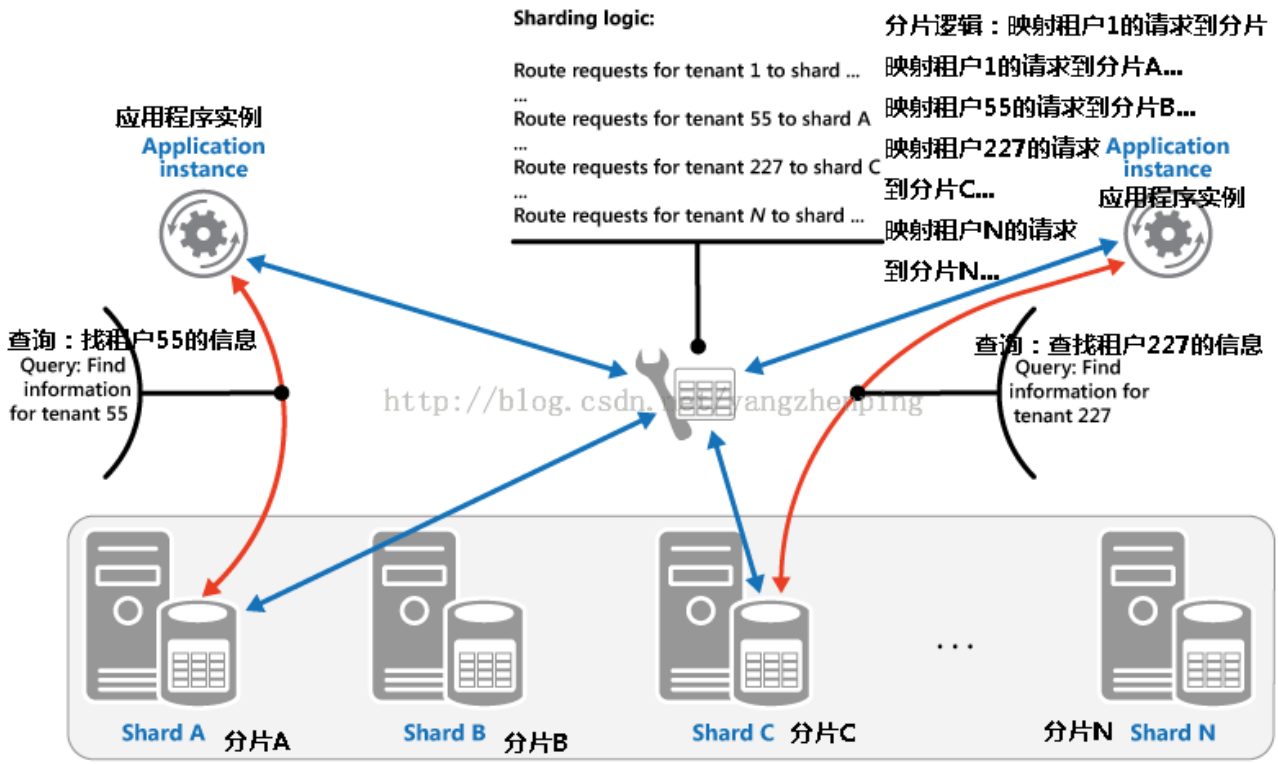


图1 – 基于租户ID的分片租户数据

分片键和物理存储的映射关系可以基于物理分块，每个分片键映射到一个物理分区。可替换地，这种技术提供了重新平衡碎片时更大的灵活性是使用虚拟分区方法，其中分片键映射到虚拟碎片的数量相同，这又映射到较少的物理分区。在这种方法中，一个应用程序通过使用指的是一个虚拟碎片一个碎片键定位数据，并在系统的虚拟分片透明地映射到物理分区。进行修改，以使用一组不同的碎片的键的虚拟碎片和物理分区可以改变，而不需要对应用程序代码之间的映射。

范围的策略。这在同一个分片相关的项目一起策略组，并把它们的订单通过分片密钥分片键是连续的。它是用于应用程序通过使用范围查询（即返回一组数据项的为落在给定范围内的碎片键查询）经常检索项集有用的。例如，如果应用程序经常需要找到放置在给定月份所有的订单，该数据可被检索更快如果一个月的所有命令被存储在日期和时间的顺序在同一个分片。如果每个订单被存储在不同的碎片，它们将必须通过进行大量的点查询（返回单个数据项的查询）的单独取出。图2示出了这种策略的一个例子。

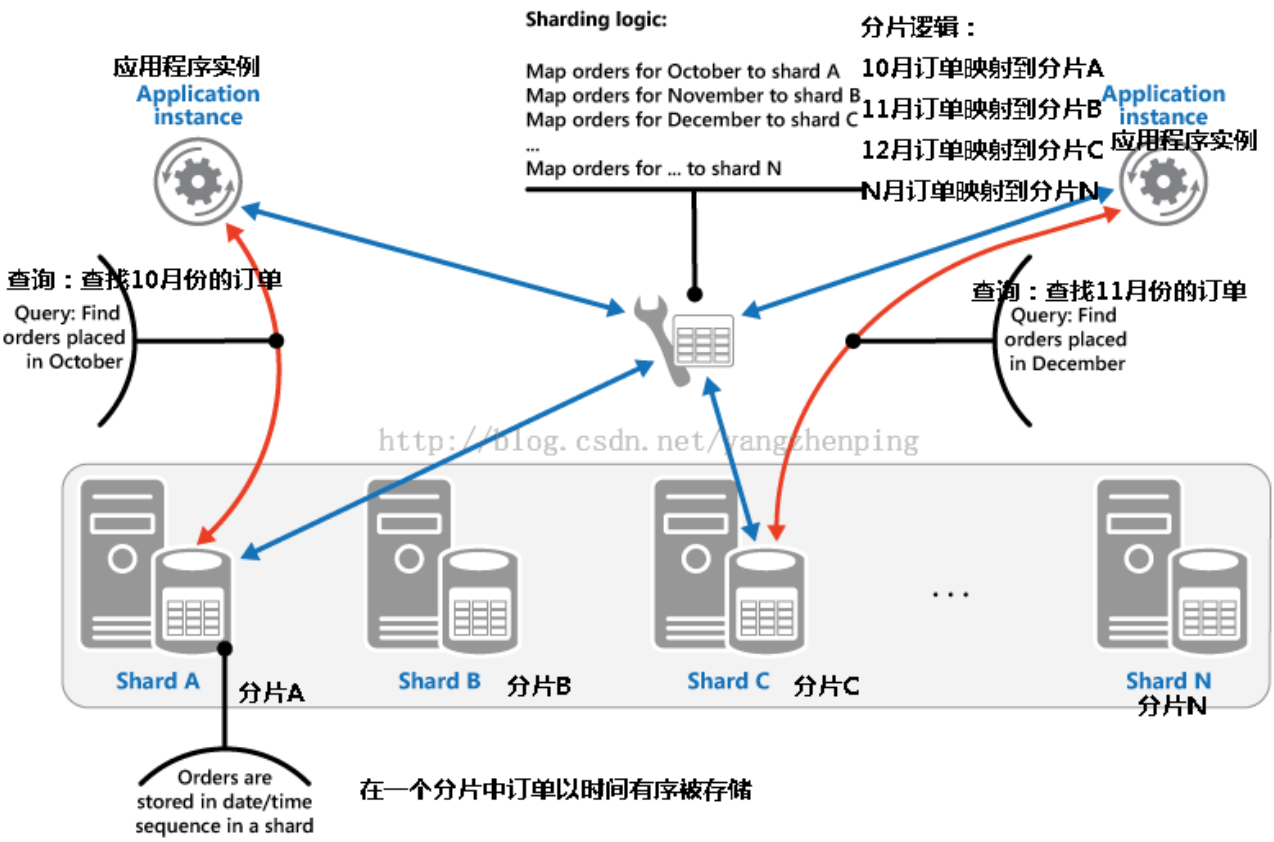


图2 – 数据中的碎片存储顺序集合（范围）

在这个例子中，分片键是一个组合键，包括订单月作为最显著元件，其次是为了日和时间。创建和附加到一个碎片新订单时，订单中的数据自然排序。一些数据存储支持包括识别所述碎片和行密钥唯一地标识该子库中的项分区键元件的两部分分片密钥。数据通常是在碎片中排键顺序举行。该受的范围内的查询和需要的项目组合在一起可以使用一个分片键具有用于分区键但该行键的唯一值相同的值。

哈希策略。这种策略的目的是减少在数据热点的机会。它的目的是分配在实现每个碎片的大小和平均负载，每个碎片会遇到之间的平衡的方式在整个碎片中的数据。分片的逻辑计算，其中基于所述数据的一个或多个属性的散列来存储中的项目的子库。所选择的散列函数应该均匀地分布在整个数据碎片，可能通过引入一些随机元素插入的计算。图2示出了这种策略的一个例子。

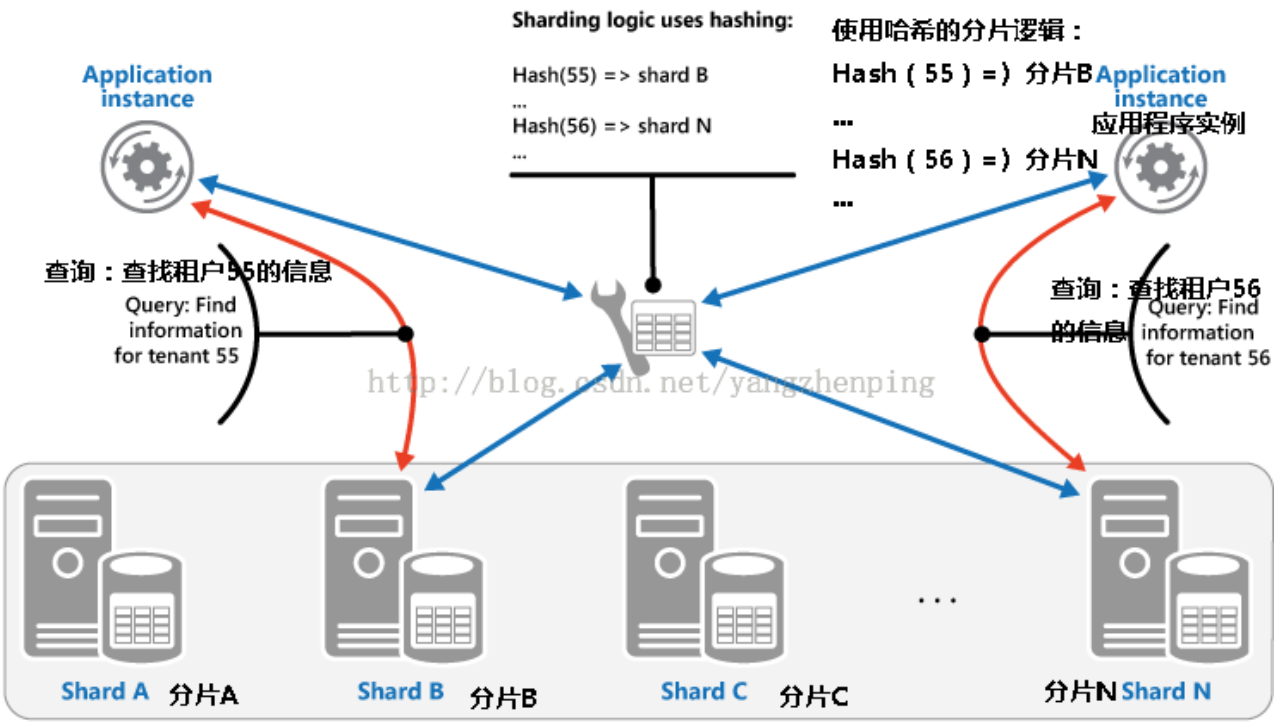


图3 – 基于租户ID的哈希分片租户数据

了解超过其他分片策略哈希策略的优势，考虑如何依序录取新租户多租户应用程序可能分配租户碎片中的数据存储。当使用范围的策略，租户1到n的数据都将存储在分片 A 中，数据为住户的 n+1 到 m 都将存储在分片 B，依此类推。如果最近登记的租户也是最活跃，最数据活动将发生在少数碎片，这可能会导致热点。与此相反，哈希策略分配租户基于对其租户ID的散列碎片。这意味着顺序租户是最有可能被分配到不同的碎片，如图 3 所示为住户 55 和 56，这将在这些碎片分配负载。

下表列出的主要优点和考虑这三个分片策略。

Lookup 查找

更好地控制碎片的配置和使用方式。

重新平衡数据时，因为新的物理分区可以被添加到拉平工作量使用虚拟碎片减少的影响。可以在不影响使用一个分片键来存储和检索数据的应用程序代码被修改的虚拟碎片和实现该分片的物理分区之间的映射。

Range 范围

易于实现和使用范围查询工作得很好，因为它们通常可以取在单个操作中从单个分片的多个数据项。

更简便的数据管理。例如，如果用户在相同的区域是在相同的子库，更新可以安排在基于本地负载和需求模式的每个时区。

Hash 哈希

一个甚至更多的数据和负荷分布的更好的机会。

请求路由可以直接通过使用哈希函数来实现。没有必要来维护一个地图。

最常见的拆分方案实现上述方法之一，但你也应该考虑你的应用程序的业务需求和他们的数据使用模式。例如，在一个多用户应用：

- 可以分片根据工作负载数据。你可以分开在不同的碎片极易挥发租户的数据。对于其他租户的数据访问的速度可以提高作为结果。
- 可以分片根据租户的位置数据。它可能会采取对租户的数据在一个特定的地理区域离线期间在该地区非高峰期备份和维护，而对于住户在其他地区的数据仍然在他们上班时间上网和访问。
- 高价值的住户能分到自己的私人高性能，轻载碎片，而价值较低的住户可能有望分享更密集堆积，忙碎片。
- 对于需要数据隔离和隐私的高度可以存储一个完全独立的服务器上租户的数据。

缩放和数据移动操作

每个分片策略意味着不同的功能和复杂性管理的规模，向外扩展，数据移动，并保持水平状态。

查找策略允许缩放和数据移动操作来进行，在用户层面，无论是在线还是离线。该技术暂停部分或全部用户活动（也许是在非高峰时段），移动数据到新的虚拟分区或物理碎片，改变映射，无效或刷新持有该数据的缓存，然后让用户活动恢复。通常这种类型的操作可以进行集中管理。查找战略要求的状态是高度可缓存和副本友好。

的范围的策略规定了结垢和数据移动操作，这通常必须进行时的一部分或全部的数据存储为脱机，因为数据必须被分割和整个碎片合并的一些限制。移动的数据，以重新平衡碎片可能无法解决不均匀负荷的问题，如果大多数的活性是对相邻分片密钥或数据标识符是相同的范围之内的。范围的策略可能也需要进行维护，以图范围内的物理分区的一些状态。

哈希策略使得扩展和数据移动操作更为复杂，因为分区键是碎片密钥或数据标识符的哈希值。每个碎片的新位置，必须从散列函数来确定，或者该函数修改，以提供正确的映射。然而，哈希策略不需要维护状态。

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 分片是互补的其他形式的分区，如垂直分区和功能划分。例如，单个子库可以包含已垂直划分实体和功能划分可以被实现为多个碎片。有关分区的详细信息，请参阅数据分区指导。

- 保持平衡碎片，使他们都处理的I / O相似的体积。作为数据被插入和删除，它可能有必要定期地重新平衡的碎片，以保证均匀分布，并减少热点的机会。再平衡可能是一个昂贵的操作。为了降低频率与再平衡成为必要，你应该确保每个碎片中含有足够的可用空间来处理变化的预期货量计划增长。你还应该制定战略和脚本，你可以用它来快速重新平衡碎片这应该成为必要。
- 使用稳定的数据分片的关键。如果分片键的变化，相应的数据项可能需要碎片之间移动，增加工作通过更新操作执行的工作量。出于这个原因，避免立足于潜在的波动信息的碎片关键。相反，寻找那些不变的或自然形成的关键属性。
- 确保碎片钥匙都是独一无二的。例如，应避免使用自动递增的字段作为分片的关键。在一些系统中，自动递增字段可以不横跨碎片协调，从而可能导致在具有相同分片键不同的碎片的项目。

注意： 在不包括分片键也可能导致问题字段自动递增值。例如，如果您使用自增字段来生成唯一标识，并分布在不同的碎片两个不同的项目可能被分配相同的ID。

- 它可能无法设计出符合针对数据的每个可能的查询要求一个分片键。分片的数据，以支持最经常进行的查询，并在必要时创建二级索引表，以支持通过使用基于不属于分片键的一部分的属性标准检索数据的查询。欲了解更多信息，请参见索引表模式。
- 查询访问仅单个碎片会比那些来自多个分块中检索数据的效率，从而避免执行一个分片方案，该方案导致在执行该连接在不同碎片保持的数据的查询的大量应用程序。请记住，碎瓷片可以包含的数据为多个类型的实体。考虑非规范化的数据保持相同的碎片常常被认为是一起查询（如客户的详细信息和订单，他们已经把）相关实体，以减少单独读取数的应用程序执行。

注意： 如果在一个子库的实体引用存储在另一个分片的一个实体，包括分片键用于第二实体，作为第一实体的架构的一部分。这可以帮助提高引用跨碎片相关数据查询的性能。

- 如果应用程序必须执行检索来自多个分块数据的查询，有可能通过使用并行任务来获取这些数据。例子包括扇出查询，其中来自多个分片的数据检索平行，然后汇总到一个结果。然而，这种方法不可避免地增加了一些复杂性的溶液的数据访问逻辑。
- 对于许多应用来说，产生的小碎片更大数目可以比具有少量的大碎片，因为它们可以提供用于负载平衡的机会增加更加有效。如果预计需要碎片从一个物理位置移动到另一个这样的方法也可以是有用的。移动小碎片比移动一个大的更快。
- 确保提供给每个分片存储节点的资源足以处理数据的规模和吞吐量方面的可扩展性要求。欲了解更多信息，请参阅数据分区引导部分“设计分区的可扩展性”。
- 考虑复制参考数据所有碎片。如果从一个子库中检索数据的操作还引用静态或缓慢移动的数据作为同一查询的一部分，这个数据添加到碎片。然后，应用程序可以读取所有用于容易地查询中的数据，而无需进行额外的往返行程到一个单独的数据存储中。

注意： 如果在多个分块的变化保持的基准数据，该系统必须同步所有碎片这些变化。而此同步发生时，系统可能会出现一定程度的混乱。如果你按照这种方法，你应该设计自己的应用程序能够处理这个矛盾。

- 它可以是难以维持的碎片之间的参照完整性和一致性，所以你应该尽量减少影响在多个碎片数据操作。如果应用程序必须通过碎片修改数据，评估完整的数据一致性是否实际上是一个要求。相反，在云中一个常用的方法是实现最终一致性。每个分区中的数据分别进行更新，并在应用程序逻辑必须承担保证责任的更新都成功完成，以及处理，可以从查询数据的最终一致的运行操作时产生的不一致。有关实现最终一致性的更多信息，请参阅数据一致性底漆。
- 配置和管理大量碎片可能是一个挑战。任务，例如监控，备份，检查一致性，并记录或审计必须完成对多个碎片和服务器，在多个位置有可能保持。这些任务可能会通过使用脚本或其他自动化解决方案，但脚本和自动化可能无法完全消除额外的行政要求执行。
- 碎片可以是地理定位的，使得它们包含的数据是靠近使用它的应用程序的实例。这种方法可以显着改善的性能，但是需要额外考虑为必须访问多个分块中的不同位置的任务。

何时使用这个模式

使用这种模式：

- 当数据存储可能需要扩展超越了一单个存储节点的资源的限制。
- 通过减少争用的数据存储来提高性能。

注意： 分片的主要焦点是改进系统的性能和可扩展性，而作为副产物，也可以借助于其中数据被划分成单独的分区的方式提高可用性。在一个分区中的故障不一定阻止应用程序访问的其他分区中保存的数据，并且操作者无需使得整个数据为应用程序无法访问的可以执行的一个或多个分区的维护或复原。欲了解更多信息，请参阅数据分区指导。

例子

下面的示例使用了一组充当碎片的 SQL Server 数据库。每个数据库包含一个应用程序使用的数据的一个子集。应用程序检索该被分布在整个碎片通过使用它自己的分片逻辑（这是一个扇出查询的一个例子）的数据。将位于每个子库中的数据的细节是通过这样的方法称为 GetShards 返回。此方法返回 ShardInformation 对象，其中 ShardInformation 类型包含一个标识符为每个碎片和 SQL Server 的连接字符串，应用程序应该使用连接到碎片的枚举列表（在连接字符串中没有代码示例所示）。

```
private IEnumerable<ShardInformation> GetShards()
{
```

```
  // This retrieves the connection information from a shard store
  // (commonly a root database).
  return new[]
  {
    new ShardInformation
    {
      Id = 1,
      ConnectionString = ...
    },
    new ShardInformation
    {
      Id = 2,
      ConnectionString = ...
    }
  };
}
```

下面的代码显示了如何在应用程序使用 ShardInformation 对象名单进行了从并行每个碎片获取数据的查询。查询的细节没有示出，但在本实施例中所检索的数据包括可以存放信息，如客户的名称，如果碎片包含客户的细节的字符串。该结果由应用聚集成 ConcurrentBag 集合进行处理。

```
// Retrieve the shards as a ShardInformation[] instance.
var shards = GetShards();

var results = new ConcurrentBag<string>();

// Execute the query against each shard in the shard list.
// This list would typically be retrieved from configuration
// or from a root/master shard store.
Parallel.ForEach(shards, shard =>
{
  // NOTE: Transient fault handling is not included,
  // but should be incorporated when used in a real world application.
  using (var con = new SqlConnection(shard.ConnectionString))
  {
    con.Open();
    var cmd = new SqlCommand("SELECT ... FROM ...", con);

    Trace.TraceInformation("Executing command against shard: {0}", shard.Id);

    var reader = cmd.ExecuteReader();
    // Read the results in to a thread-safe data structure.
    while (reader.Read())
    {
      results.Add(reader.GetString(0));
```

```
    }
  }
});

Trace.TraceInformation("Fanout query complete - Record Count: {0}",
                       results.Count);
```

22

静态内容托管模式

部署静态内容到一个基于云的存储服务，可以直接向客户提供这些。这个模式可以减少潜在的昂贵的计算实例的需求。

背景和问题

Web应用程序通常包括静态内容的一些元素。此静态内容可以包括HTML页面和诸如图像和可用到客户端的文件的其他资源，无论是作为一个 HTML 页的一部分（如嵌入式图像，样式表和客户端 JavaScript 文件）或作为单独的下载（如PDF文档）。

尽管Web服务器以及调整通过有效的动态执行页代码和输出缓存优化的要求，他们仍然必须处理请求下载静态内容。这种吸收，可以经常得到更好的利用处理周期。

解决方案

在大多数云托管环境中，它可以最小化用于计算实例的要求（例如，使用较小的实例或更少的情况下），通过定位部分的应用程序的资源和静态网页中的存储服务。费用为云托管的存储通常比计算实例少得多。

何时主机在一个存储服务的应用的某些部分，主要考虑的是与应用程序的部署以及确保其不旨在提供给匿名用户的资源。

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 托管存储服务必须公开，用户可以访问下载静态资源的 HTTP 端点。一些存储服务还支持 HTTPS，这意味着它能够承载需要使用SSL在存储业务资源。
- 为了获得最高的性能和可用性，可以考虑使用内容分发网络（如果有的话）来缓存在世界各地的多个数据中心的存储容器中的内容。但是，这将产生额外费用的使用内容交付网络。
- 存储账户往往 GEO-复制默认情况下，提供弹性对可能影响数据中心的事件。这意味着它们的 IP 地址可能会改变，但该URL将保持不变。
- 当一些内容位于一个存储账户等内容的托管计算实例变得更具挑战性来部署应用程序并对其进行更新。这可能是必要的，以便当所述静态内容包括脚本文件或用户界面组件来管理它更容易，尤其是分别执行的部署，以及版本的应用程序和内容。然而，如果仅仅静态资源要更新他们可以简单地被上传到存储帐户，而无需重新部署应用程序包。

- 存储服务可能不支持使用自定义域名。在这种情况下，有必要在链接指定的资源的完整 URL，因为它们将在从含有链接动态地生成的内容不同的域。
- 存储容器必须为公共读取权限进行配置，但它是至关重要的，以确保它们没有被配置为市民写访问权限，以防止用户能够上传内容。请考虑使用代客钥匙或令牌控制对资源的访问不应该用匿名，看到代客主要模式的更多信息。

何时使用这个模式

这种模式非常适合于：

- 最小化的网站，并包含一些静态资源应用的托管费用。
- 最小化的网站只包含静态内容和资源的托管费用。根据不同的托管服务提供商的存储系统的功能，有可能承载全静态网页的全部内容存储帐户内。
- 暴露的静态资源和其他宿主环境或本地服务器上运行的应用程序的内容。
- 通过使用缓存的存储账户中的内容在世界各地的多个数据中心的内容分发网络定位在多个地理区域中的内容。
- 监测成本和带宽的使用。使用一段静态内容的部分或全部单独的存储帐户允许的成本更容易分辨从承载和运行成本。

这种模式可能不适合于下列情况：

- 应用程序需要将它传递给客户端之前对静态内容进行一些处理。例如，它可能是必要的时间戳添加到文档中。
- 静态内容的数量是非常小的。检索从单独的存储该内容的开销可能会超过它的计算资源中分离出来的成本效益。

注意： 它有时是可以存储一个完整的网站只包含静态内容，如 HTML 页面，图片，样式表，客户端 JavaScript 文件，下载的文件，如在云中托管的存储 PDF 文件。欲了解更多信息，请参阅在 Infosys 的博客部署静态网站在微软 Azure 的有效途径。

例子

位于 Azure 的 Blob 存储静态内容，可直接通过 Web 浏览器进行访问。 Azure 提供一个基于 HTTP 的接口上的存储，可以公开暴露给客户。例如，在一个天青 Blob 存储容器内容使用形式的 URL 被公开：

```
HTTP：// [存储帐户名称].blob.core.windows.net/[容器名称]/[文件名]
```

何时上载该应用程序的内容，必须创建一个或多个斑点的容器来保存文件和文档。请注意，对于一个新的容器的默认权限是私有的，你必须改变这种公共允许客户端访问的内容。如果有必要，以防止匿名访问的内容，您可以实现代客主要模式，因此用户必须按顺序下载资源出示有效的令牌。

注意： 在 Azure 网站上的页面 Blob 服务的概念包含了 Blob 存储信息，并且您可以访问它，并用它的方式。

在每个页面中的链接将指定的资源的 URL，客户端将直接从存储服务访问该资源。图1示出了这种方法。

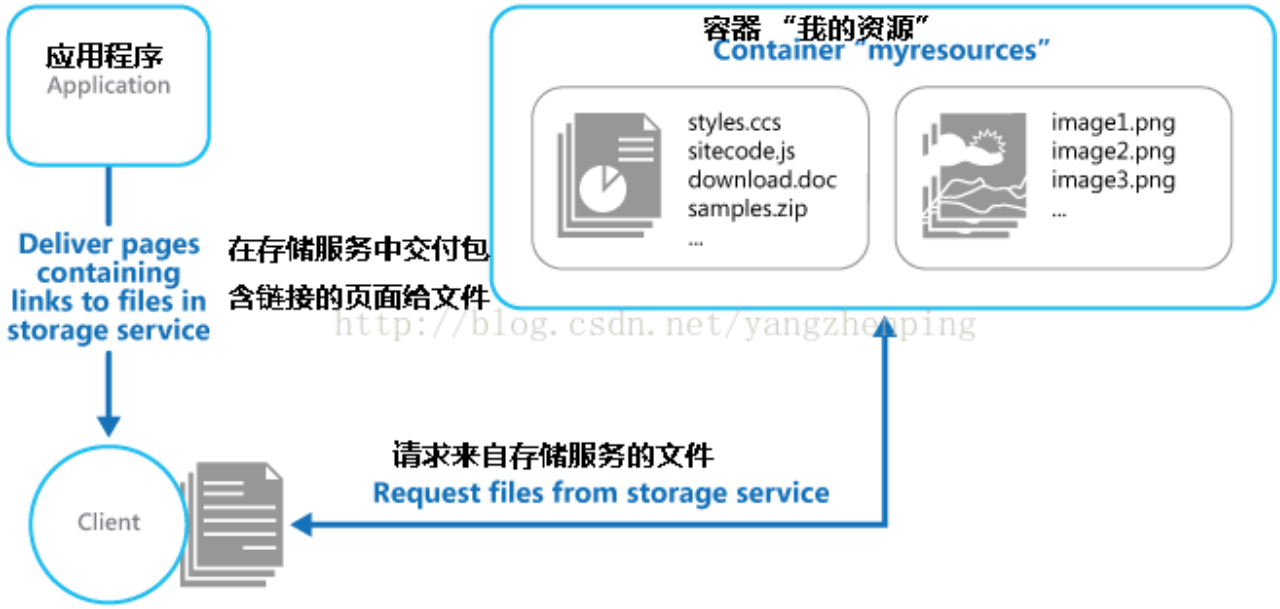


图1 – 从存储服务交付的应用程序的静态部分，直接

在传送到客户端的页面的链接必须指定的 blob 容器和资源的完整 URL。例如，包含在一个公共容器的链接的图像的页面可能包含以下内容。

```
<img src="http://mystorageaccount.blob.core.windows.net/myresources/image1.png"
    alt="My image" />
```

注意：

如果该资源是通过使用代客密钥进行保护，如天青共享访问签名（SAS），该签名必须被包含在链接的 URL。

适用于本指南中的示例包含一个名为 StaticContentHosting，演示了使用外部存储静态资源的解决方案。该 StaticContentHosting.Cloud 项目包含指定保存静态内容的存储帐户和容器的配置文件。

```
<Setting name="StaticContent.StorageConnectionString"
        value="UseDevelopmentStorage=true" />
<Setting name="StaticContent.Container" value="static-content" />
```

在 StaticContentHosting.Web 项目的文件 Settings.cs Settings（设置）类包含的方法来提取这些值，并建立一个字符串值，包含云存储帐户的容器的 URL。

```
public class Settings
{
  public static string StaticContentStorageConnectionString {
    get
    {
      return RoleEnvironment.GetConfigurationSettingValue(
                        "StaticContent.StorageConnectionString");
    }
  }

  public static string StaticContentContainer
  {
    get
    {
      return RoleEnvironment.GetConfigurationSettingValue("StaticContent.Container");
    }
  }

  public static string StaticContentBaseUrl
  {
    get
    {
      var account = CloudStorageAccount.Parse(StaticContentStorageConnectionString);

      return string.Format("{0}/{1}", account.BlobEndpoint.ToString().TrimEnd('/'),
                          StaticContentContainer.TrimStart('/'));
    }
  }
}
```

在文件 StaticContentUrlHtmlHelper.cs 的 StaticContentUrlHtmlHelper 类公开命名 StaticContentUrl，如果传递给它的URL与ASP.NET根路径字符（？）开始生成包含路径的云存储帐户的 URL 的方法。

```
public static class StaticContentUrlHtmlHelper
{
  public static string StaticContentUrl(this HtmlHelper helper, string contentPath)
  {
    if (contentPath.StartsWith("~"))
    {
      contentPath = contentPath.Substring(1);
    }
```

```
    contentPath = string.Format("{0}/{1}", Settings.StaticContentBaseUrl.TrimEnd('/'),
                                contentPath.TrimStart('/'));

    var url = new UrlHelper(helper.ViewContext.RequestContext);

    return url.Content(contentPath);
  }
}
```

在浏览文件 Index.cshtml\ Home 文件夹包含使用 StaticContentUrl 方法创建的 URL，它的 src 属性的图像元素。

```
<img src="@Html.StaticContentUrl("~/Images/orderedList1.png")" alt="Test Image" />
```

23

Throttling 节流模式

控制由应用程序使用，一个单独的租户或整个服务的一个实例的资源的消耗。这种模式可以允许系统继续运行并满足服务水平协议，即使当增加需求的资源放置一个极端载荷。

背景和问题

在云应用负载通常上变化的基础上的活动用户的数量或他们正在执行的活动类型的时间。例如，多个用户可能会在工作时间被激活，否则系统可能被要求在每月结束时执行计算昂贵的分析。也有可能是突然和意外的突发活动。如果系统的处理要求超过了可用的资源的能力，其将遭受性能不佳，甚至会失败。该系统可能必须满足的服务约定的水平，并且这种故障可能是不可接受的。

有许多策略可用于处理可变负载在云中，根据业务目标的应用程序。一种策略是使用自动缩放来在任何给定时间相匹配的供应资源给用户的需要。这具有始终如一地满足用户需求，同时优化运行费用的潜力。然而，尽管自动缩放可能会引发更多的资源配置，这配置是不是瞬间。如果需求快速增长，有可能是一个时间窗口，那里是一个资源赤字。

解决方案

另一种策略来自动缩放是为了让应用程序能够使用的资源最多只有一些软限位，然后油门当他们达到此限制。该系统应监测它是如何使用的资源，使得当使用量超过一些系统定义的阈值时，它可以调节来自一个或多个用户的请求，以使系统继续工作，并满足任何服务级别协议（SLA），该已到位。有关监控资源使用情况的详细信息，请参阅仪器和遥测指导。

该系统可以实现多种限制策略，其中包括：

- 从谁已经访问系统API超过每秒n次超过给定时间内的个人用户拒绝请求。这就要求系统米利用资源用于运行应用程序的每个租户或用户。欲了解更多信息，请参阅服务计量指引。
- 禁用或有辱人格的选择不必要的服务的功能，以便必要的服务可以提供足够的资源运行畅通。例如，如果应用程序是视频流输出，它可以切换到一个较低的分辨率。
- 使用负载均衡来平滑活动量（这种方法是覆盖在由基于队列的负载均衡模式的更多细节）。在多租户环境中，这种方法将减少为每一个租户的性能。如果系统必须支持的住户有不同的SLA的组合，为高价值租户的工作可能会被立即执行。请求其他住户可以忍住，以及时处理积压有所缓解。优先级队列模式，可以用来帮助实现此方法。
- 代表低优先级的应用程序或租户被推迟执行的操作。这些操作可以暂停或削减，异常生成的通知，该系统正忙，该操作应该稍后重试房客。

图1示出一个区域图进行资源利用率（存储器，CPU，带宽，以及其它因素的组合）对时间对于正在使用的三个特征的应用程序。一个特征是功能性的区域，例如，执行特定的任务集，一个代码段，执行一个复杂的计算，或者，提供了一个服务，例如在内存中缓存的元素的组分。这些特征被标记为 A，B 和 C。

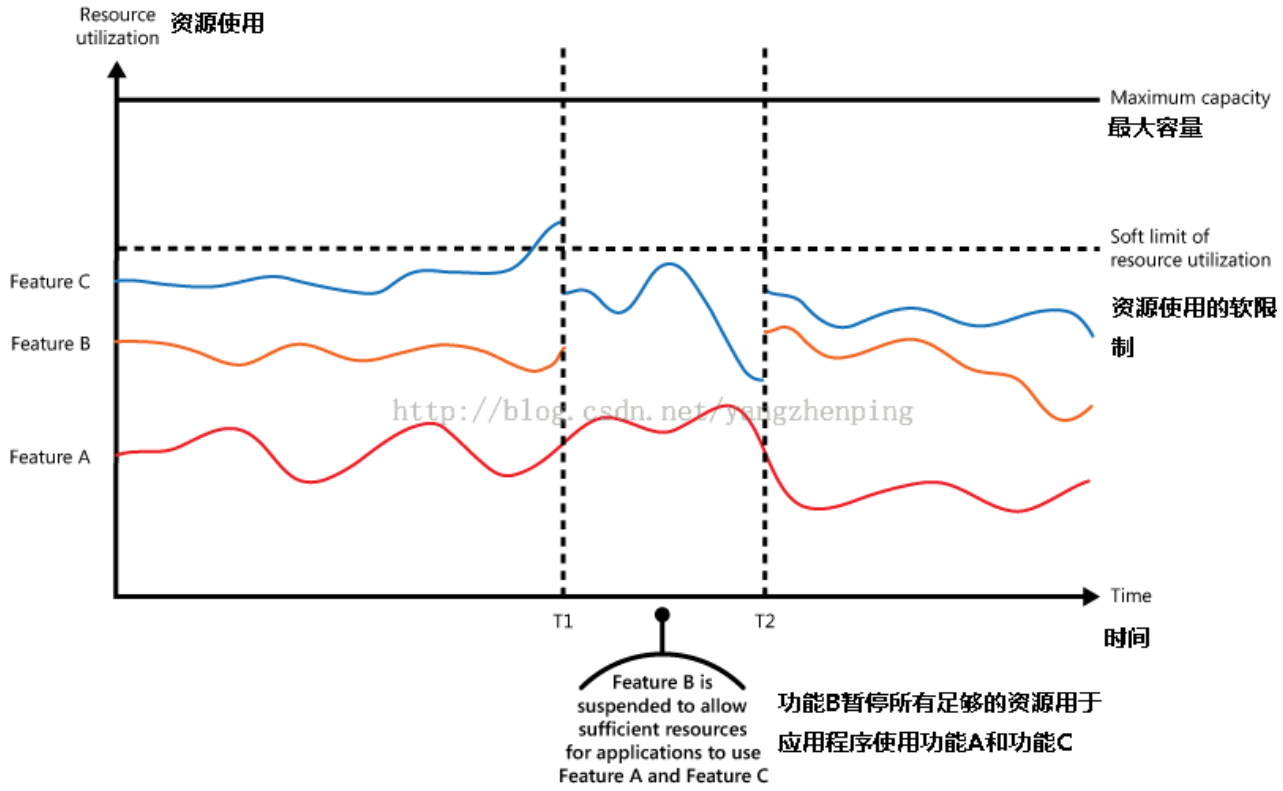


图1 – 对时间的曲线图的资源利用率代表三个用户运行的应用程序

注意：

立即行功能下的区域表示应用程序中使用时，调用此功能的资源。例如，下面的线为特色的一个区域显示使用的是正在使用的功能 A 的应用资源，并为特征 A 和特征 B 线之间的区域被使用的应用程序调用功能 B.汇总的指示资源对于每个特征区域显示了系统的总的资源利用率。

在图1中的曲线示出了延迟操作的效果。只是之前的时间 T1，分配给使用这些功能的所有应用程序的总资源达到一个阈值（资源利用的软限制）。在这一点上，应用程序是在用尽可用的资源的危险。在这个系统中，特征 B 比特点 A 或特征 Ç 不太重要，所以它是暂时禁用，并且它被使用的资源被释放。之间的时间 T1，T2，使用功能 A 和功能 C 中的应用程序继续运行正常。最后，资源利用这两个功能减退的点时，在时间 T2 时，有足够的容量，以再次启用功能 B 中。

该自动缩放和调节方法也可以结合，以帮助保持应用程序响应和 SLA 之内。如果需求预计将保持高位，节流可以提供一个临时的解决方案，同时在系统扩展了。在这一点上，该系统的全部功能可以恢复。

图 2 示出了整体的资源利用通过在与时间的系统中运行的所有应用程序的区域图，并示出了如何限制可与自动缩放组合。

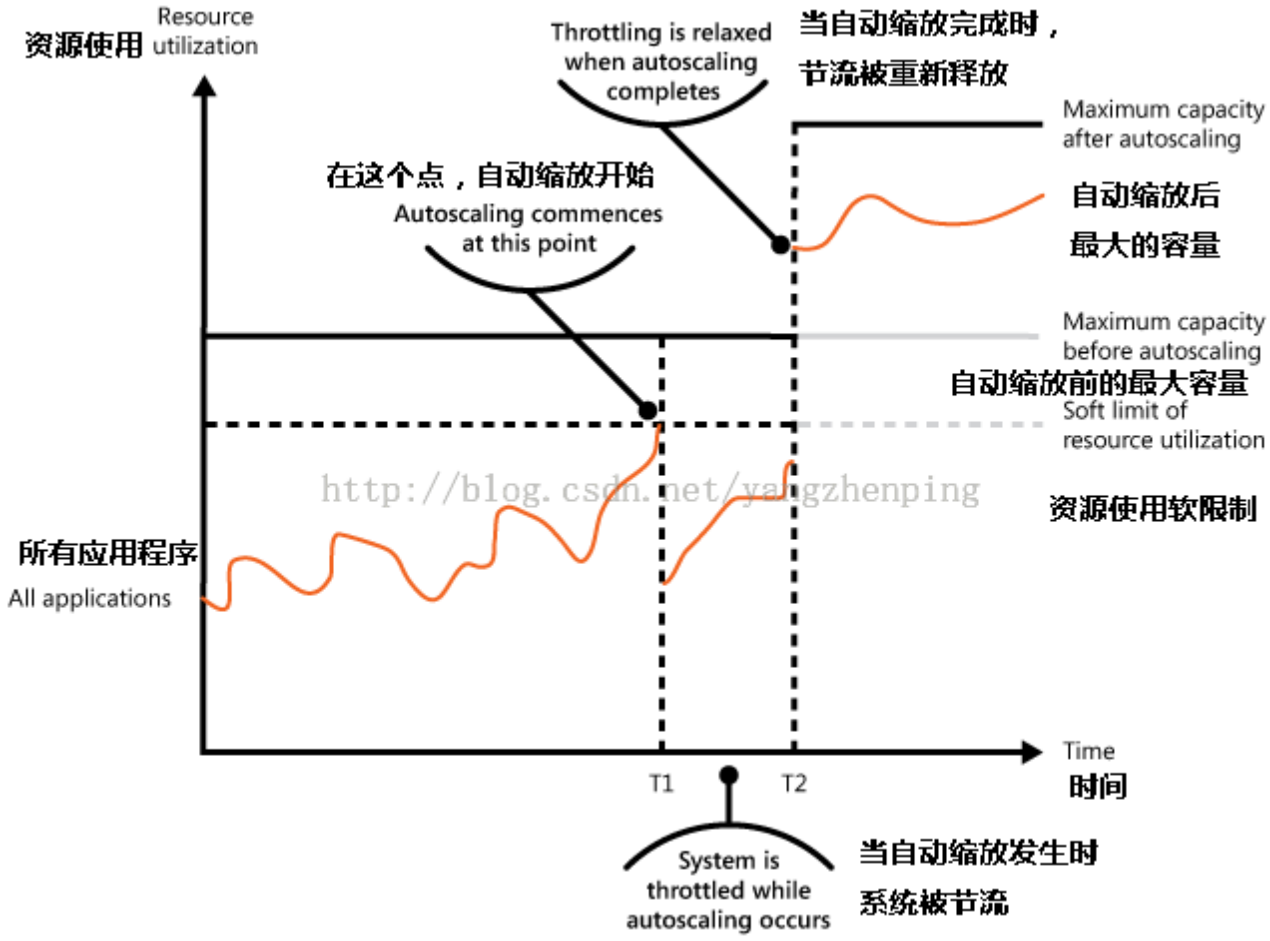


图2 – 图表显示自动缩放与节流相结合的效应

在时间T1，门槛指定资源利用的软限制为止。在这一点上，系统可以开始向外扩展。然而，如果新的资源不成为可用的足够快地再现有的资源可能被耗尽，并且系统可能会失败。为了防止这种情况发生，系统被暂时限制，如前面所述。何时自动缩放已完成和额外资源，限制可以放宽。

问题和注意事项

在决定如何实现这个模式时，您应考虑以下几点：

- 节流的应用程序，并使用策略，是一个建筑的决定，影响系统的整体设计。节流应在应用设计之初就被考虑，因为这是不容易的添加它一旦系统已经实施。
- 节流必须迅速执行。系统必须能够检测活性增加，并相应地作出反应。该系统还必须能够恢复到原来的状态后快速负载有所缓和。这需要相应的性能数据是不断捕获和监测。

- 如果一个服务需要暂时拒绝用户的请求，则它应该返回一个特定的错误代码，以使客户端应用程序理解为拒绝执行某种操作的原因是由于节流。客户端应用程序可以等待一段时间，然后重试该请求。
- 节流可作为而系统 autoscales一项临时措施。在某些情况下它可能是更好的简单节流，而不是按比例，如果在活动突发是突然的并且预计不会被长寿命，因为结垢可显着增加了运行成本。
- 如果节流正在使用的临时措施，而一个系统 autoscales，并且如果资源需求迅速增长，系统可能无法继续运作，即使在节流模式中操作时。如果这是不能接受的，考虑维护大容量的储备和配置更积极自动缩放。

何时使用这个模式

使用这种模式：

- 为了确保系统持续满足服务水平协议。
- 为了防止单一租户独占由应用程序所提供的资源。
- 为了处理突发活动。
- 为了帮助限制需要保持它运转的最大资源水平的成本优化的系统。

例子

图3示出了如何限制可以在多租户系统来实现。从每个租户组织的用户访问一个云托管的应用程序，他们填写并提交调查。应用程序中包含的仪器，用于监视在其中这些用户提交请求给应用程序的速度。

为了防止用户从一个租户影响应用的所有其他用户的响应性和可用性，限制施加到每秒从任何一个租户的用户可以提交请求的数目。该应用程序块请求超过此限制

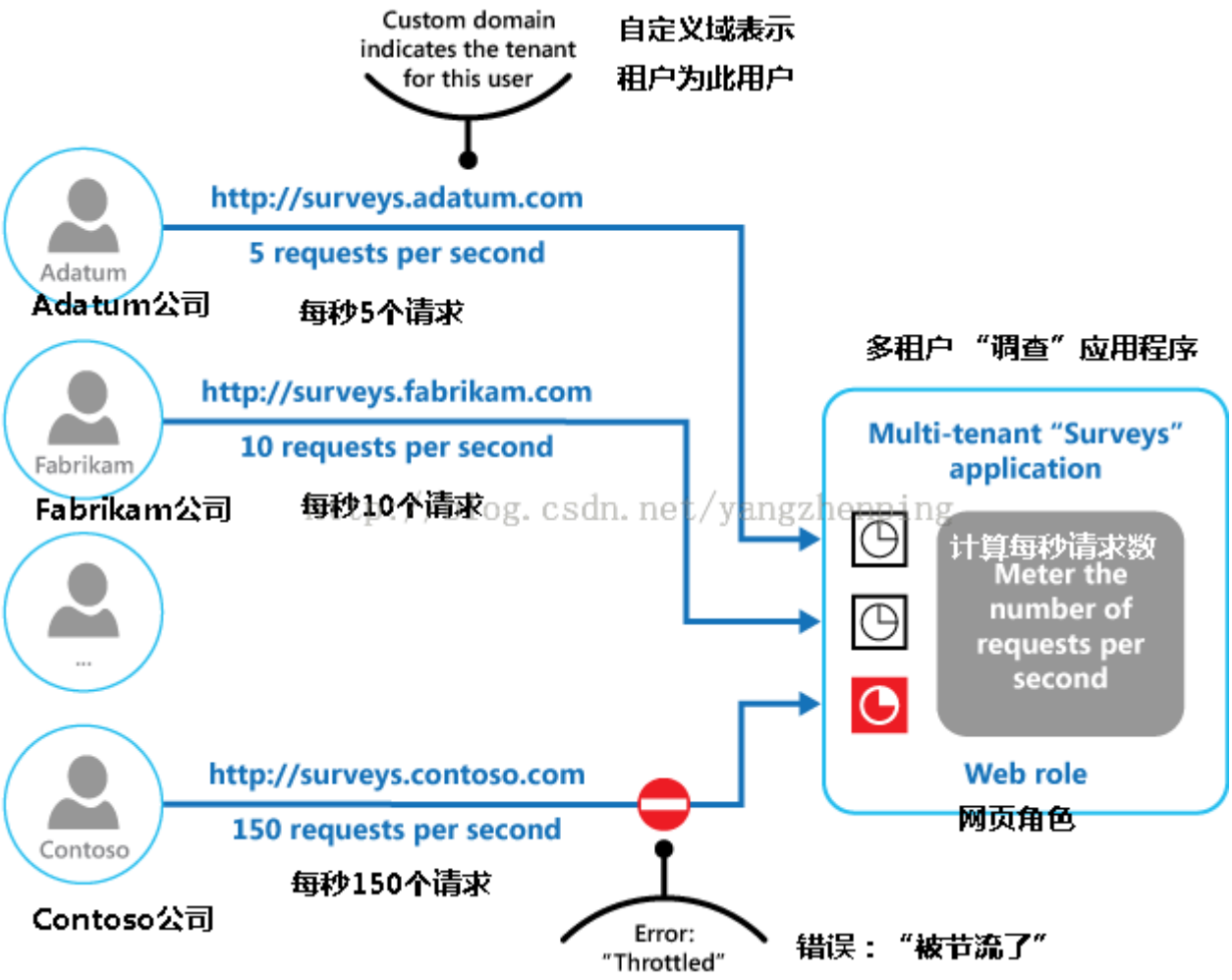


图3 – 在一个多租户应用程序中实现节流

24

仆人键模式

使用一个令牌或密钥，向客户提供受限制的直接访问特定的资源或服务，以便由应用程序代码卸载数据传输操作。这个模式是在使用云托管的存储系统或队列的应用中特别有用，并且可以最大限度地降低成本，最大限度地提高可扩展性和性能。

背景和问题

客户端程序和网络浏览器经常需要读取和写入文件或数据流，并从一个应用程序的存储空间。通常，应用程序将处理的运动数据，或者通过从存储读取它，并将其传输到客户端，或通过从客户机读取该载流并将其存储在数据存储中。然而，这种方法吸收了宝贵的资源，如计算，存储和带宽。

数据存储要处理的上载和直接数据的下载，而不需要对应用程序执行任何处理移动至该数据的能力，但是这通常需要在客户端能够访问该存储区中的安全凭证。虽然这可能是一种有用的技术来减少数据传送费用的要求进行扩展的应用，并以最大化性能，这意味着应用程序不再能够管理的数据的安全性。一旦客户端已到数据存储器进行直接访问的连接，应用程序不能充当看门人。它不再是在该方法的控制，并且不能防止随后上载或下载的数据存储中。

这不是，可能需要使用不受信任的客户的现代分布式系统实事求是的态度。相反，应用程序必须能安全地控制对数据的访问是粒状的方法，但仍然通过设置此连接，然后使客户端能够直接与数据存储来执行所需的读或写操作的通信降低服务器上的负载。

解决方案

要解决控制访问的数据存储在那里的商店本身无法管理身份验证和客户授权的问题，一个典型的解决方案是限制访问的数据存储的公共连接，并提供客户端用钥匙或令牌数据存储本身可以验证。

这个密钥或令牌通常被称为仆人键。它提供了对特定资源的时间限制的访问中，仅允许预定的操作，例如读取和写入到存储或队列，或上载和下载的Web浏览器。应用程序可以创建和发行代客键客户快速，方便地设备和网络浏览器，允许客户端，而无需应用程序直接处理数据传送执行所需的操作。这消除了处理开销，并且在性能和可扩展性所造成的影响，从该应用程序和该服务器。

客户端使用该令牌来访问特定资源中的数据存储为只有特定的时期，并与访问权限的特定的限制，如示于图1中指定的时间后，将键变为无效并且不会允许后续访问该资源。

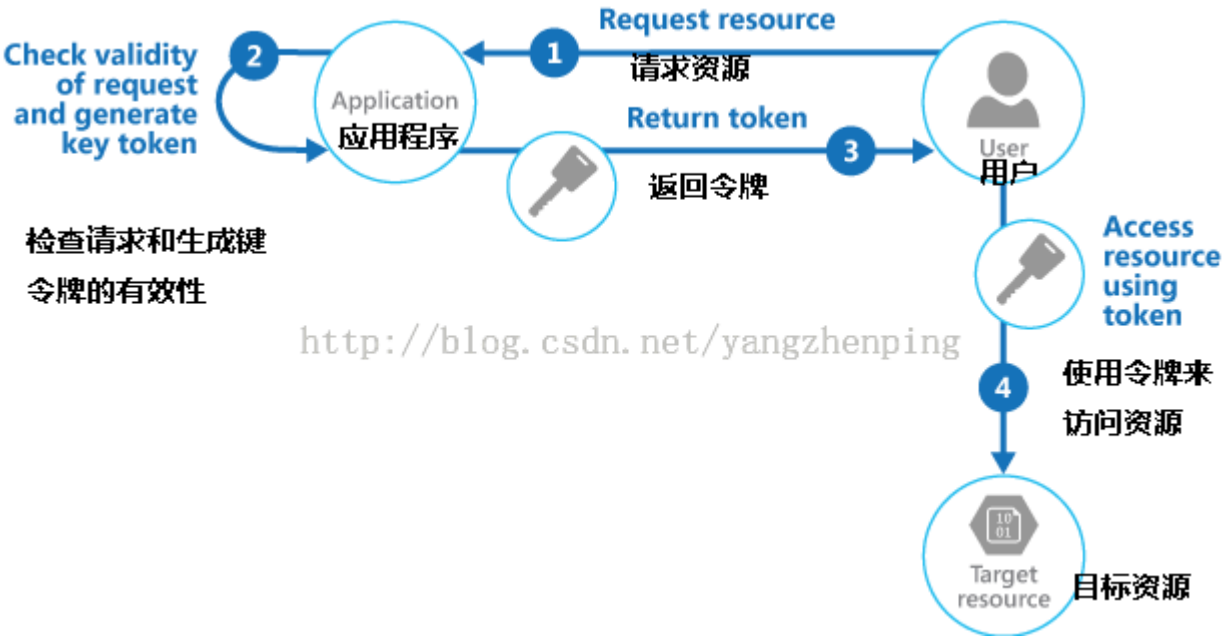


图1 – 模式概述

另外，也可以配置具有其他依赖关系，如该数据的位置的范围的一个关键。例如，根据不同的数据存储能力，所述键可在数据存储区指定一个完整的表格，或在表中仅特定的行。在云存储系统中的密钥可以指定一个容器，或只是一个特定项目的容器内。

键，也可以由应用程序无效。这是一种有用的方法，如果客户端通知该数据传送操作完成的服务器。然后，服务器可以是无效键，以防止将其用于任何后续访问的数据存储中。

使用这种模式可以简化管理对资源的访问，因为没有要求创建和验证用户，授予权限，然后再删除用户。它也可以很容易地限制的位置，允许，和有效期，所有通过简单地产生一个合适的键在运行时。的重要因素是限制的有效期，以及资源的特别的位置，尽可能紧，以使接收方可以将其用于仅在预定的目的。

问题和注意事项

在决定如何实现这个模式时，请考虑以下几点：

- 管理密钥的有效性状态和时期。最关键的是不记名票据，如果泄露或泄露，有效地解除锁定目标项目，并使其可用于在有效期内恶意使用。一键通常可撤销或无效，这取决于它是如何发出的。服务器端的策略可以被改变，或者在最终的情况下，服务器键入其用可被无效签名。指定一个短有效期，尽量减少使后续的无端操作来发生对数据存储的风险。然而，如果在有效期太短时，客户端可能无法在密钥过期之前完成该操作。允许授权用户如果多次访问所需的受保护资源的有效期间过期之前更新的关键。

- 访问控制的关键将提供的水平。典型地，该键应允许用户执行仅需要完成操作的行动，诸如只读访问，如果客户端不应该能够将数据上传到数据存储器。文件上传时，通常指定一个键，它提供只写权限，以及位置和有效期。至关重要的是要精确地指定资源或资源集到的关键应用。
- 考虑如何控制用户的行为。实施这种模式是指控制一定的损失转移到哪些用户有权访问的资源。控制的可施加的电平是由可用于该服务或目标数据存储区中的策略和许可的能力的限制。例如，它通常是不可能创建密钥，限制数据的大小将被写入到存储，或者次数的密钥可用于访问文件的数目。这可导致由预期客户所使用，即使和可能是由在可能会导致重复上载或下载的代码的错误将导致数据传输巨大意想不到的成本。限制次数的文件可以被上载或下载的，可能有必要在可能情况下的数量，能够强制客户端，当一个操作完成后，通知该应用程序。例如，某些数据存储引发事件的应用程序代码，可用于监视操作和控制的用户行为。然而，它可能是很难执行对个人用户的配额在多租户场景，其中相同的密钥从一个租户使用的所有用户。
- 验证和可选消毒，所有上传的数据。该收益的关键访问一个恶意用户可以上传，旨在进一步降低了系统的数据。可替换地，授权用户可上载的数据是无效的，并在处理时，可能会导致错误或系统故障。为了防止这一点，确保所有上传的数据进行验证，并检查使用前恶意内容。
- 审核所有操作。许多基于密钥的机制可以登录的操作，如上传，下载，和失败。这些日志通常可以并入一个审核过程，并且还用于计费，如果基于文件大小或数据量的用户被收取费用。使用日志来检测可能是由与键提供的一个存储的访问策略，或者意外移开问题导致验证失败。
- 提供关键安全。它可以被嵌入在用户激活在web页面的URL，或者可以在一个服务器重定向操作中使用，以便自动进行下载。始终使用HTTPS传送的关键在一个安全通道。
- 保护敏感数据在传输过程中。通过应用程序发送的敏感数据通常会发生使用SSL或TLS，这应该被强制执行的客户端直接访问数据存储。

其他问题要注意实现这个模式的时候是：

- 如果客户端没有，或者无法通知操作完成的服务器，唯一的限制是关键的到期期限，该应用程序将无法执行审计业务，如计算上载或下载的数量，或防止多个上传或下载。
- 可以生成可能是有限的关键策略的灵活性。例如，一些机制可以允许只使用一种定时期满期。其他人可能无法以指定读/写权限的足够的粒度。
- 如果指定的开始时间的键或令牌有效期限，确保它是比当前的服务器时间，以允许客户端时钟，可能会稍微超出同步早一点。如果没有指定，则默认通常是当前服务器时间。
- 包含密钥的URL将被记录在服务器日志文件中。而键通常已过期的日志文件进行分析之前，请确保您限制对它们的访问。如果日志数据发送到监控系统或存储在另一位置中，需要考虑的延迟，以防止密钥的泄漏，直至经过其有效期限已经过期。

- 如果客户端代码，在Web浏览器中运行，浏览器可能需要支持跨域资源共享（CORS），使Web浏览器中执行访问的数据从该服务的网页源域不同的域代码。一些旧的浏览器和一些数据存储不支持CORS，和代码运行在这些浏览器可能无法使用仆人键，提供对数据的访问在不同的领域，比如云存储帐户。

何时使用这个模式

这种模式非常适合于以下几种情况：

- 为了最大限度地减少资源负荷，最大限度地提高性能和可扩展性。使用仆人键不要求资源被锁定，没有远程服务器呼叫是必需的，有对可发出仆人键的数目没有限制，并且其避免了单点，将出现从执行数据失败的传送通过应用程序代码。创建仆人键通常是签订一个字符串键的简单加密操作。
- 为了最大限度地降低运营成本。支持直接访问存储和队列是资源与成本效率，可以导致更少的网络往返，并且可以允许在所需的计算资源的数量的减少。
- 当客户定期上载或下载数据，特别是在有一个大的体积，或当每个操作涉及大量的文件。
- 当应用程序具有有限的计算资源可用，或者是由于托管限制或成本的考虑。在这种情况下，该模式是更有利的，如果有很多并发的数据上传或下载，因为它会减轻，从处理数据传输的应用。
- 当数据被存储在远程数据存储装置或不同的数据中心。如果该应用程序被要求作为一个看门者，有可能是因为数据中心之间传送数据的额外的带宽，或通过客户端和应用程序之间的公共或专用网络的电荷，然后将应用程序和数据存储之间。

这种模式可能不适合于下列情况：

- 如果应用程序必须对数据执行一些任务之前将其存储或将其发送到客户端之前。例如，应用程序可能需要执行验证，登录访问成功，或者对数据进行变换。然而，一些数据存储和客户端都能够进行谈判，并进行简单的变换，如压缩和解压（例如，Web 浏览器通常可以处理 gzip 等格式）。
- 如果现有的应用程序的设计和实施，使得它难以和昂贵的工具。使用这种模式通常需要用于传送和接收数据的一个不同的体系结构方法。
- 如果有必要保持审核跟踪或控制的执行数据传送操作的次数，并在使用代客关键机制不支持该服务器可用于管理这些操作的通知。
- 如果有必要，以限制该数据的大小，特别是在上载操作。唯一的解决方法是为应用程序来检查数据的尺寸后，在操作完成后，或在指定时间或按计划检查上传的大小。

例子

微软 Azure 支持共享访问签名（SAS）对 Azure 存储的细粒度的访问控制，数据的blob，表和队列，并为服务总线队列和主题。一个 SAS 令牌可配置为提供特定的访问权限，如读，写，更新和删除特定表;一个键范围的表内;队列;一个 blob;或 BLOB 容器。有效期可以是一个指定的时间段，或没有时间限制。

天青 SAS 还支持可与特定资源相关联，如表或斑点服务器存储访问策略。这个特征提供了额外的控制和灵活性相比，应用程序生成的 SAS 令牌，并且应该尽可能使用。在服务器中存储策略中定义的设置可以在不发出新的令牌来改变，并反映在无需将发行新令牌的令牌，但在令牌本身定义的设置不能被改变。这种方法还使得有可能撤销有效的 SAS 令牌之前它已经过期。

注意： 欲了解更多信息，请参阅表介绍 SAS（共享访问签名），队列 SAS 和更新的 Blob SAS 在 Azure 存储团队博客和共享访问签名，第 1 部分：了解 SAS 型号 MSDN 上。

下面的代码演示了如何创建一个 SAS 的有效期为 5 分钟。该 GetSharedAccessReferenceForUpload 方法返回一个SAS可用于将文件上传到 Azure 的 Blob 存储。

```
public class ValuesController : ApiController
{
  private readonly CloudStorageAccount account;
  private readonly string blobContainer;
  ...
  /// <summary>
  /// Return a limited access key that allows the caller to upload a file
  /// to this specific destination for a defined period of time.
  /// </summary>
  private StorageEntitySas GetSharedAccessReferenceForUpload(string blobName)
  {
    var blobClient = this.account.CreateCloudBlobClient();
    var container = blobClient.GetContainerReference(this.blobContainer);

    var blob = container.GetBlockBlobReference(blobName);

    var policy = new SharedAccessBlobPolicy
    {
      Permissions = SharedAccessBlobPermissions.Write,

      // Specify a start time five minutes earlier to allow for client clock skew.
      SharedAccessStartTime = DateTime.UtcNow.AddMinutes(-5),

      // Specify a validity period of five minutes starting from now.
      SharedAccessExpiryTime = DateTime.UtcNow.AddMinutes(5)
```

```
    };

    // Create the signature.
    var sas = blob.GetSharedAccessSignature(policy);

    return new StorageEntitySas
    {
      BlobUri = blob.Uri,
      Credentials = sas,
      Name = blobName
    };
  }

  public struct StorageEntitySas
  {
    public string Credentials;
    public Uri BlobUri;
    public string Name;
  }
}
```

注意：

在 ValetKey 解决方案提供下载本指导意见提供包含此代码的完整样本。在此溶液中 ValetKey.Web 项目包含一个 Web 应用程序，包括如上所示的 ValuesController 类。使用该 Web 应用程序检索 SAS 键，将文件上传到 Blob 存储的样本客户端应用程序是在 ValetKey.Client 项目中可用。