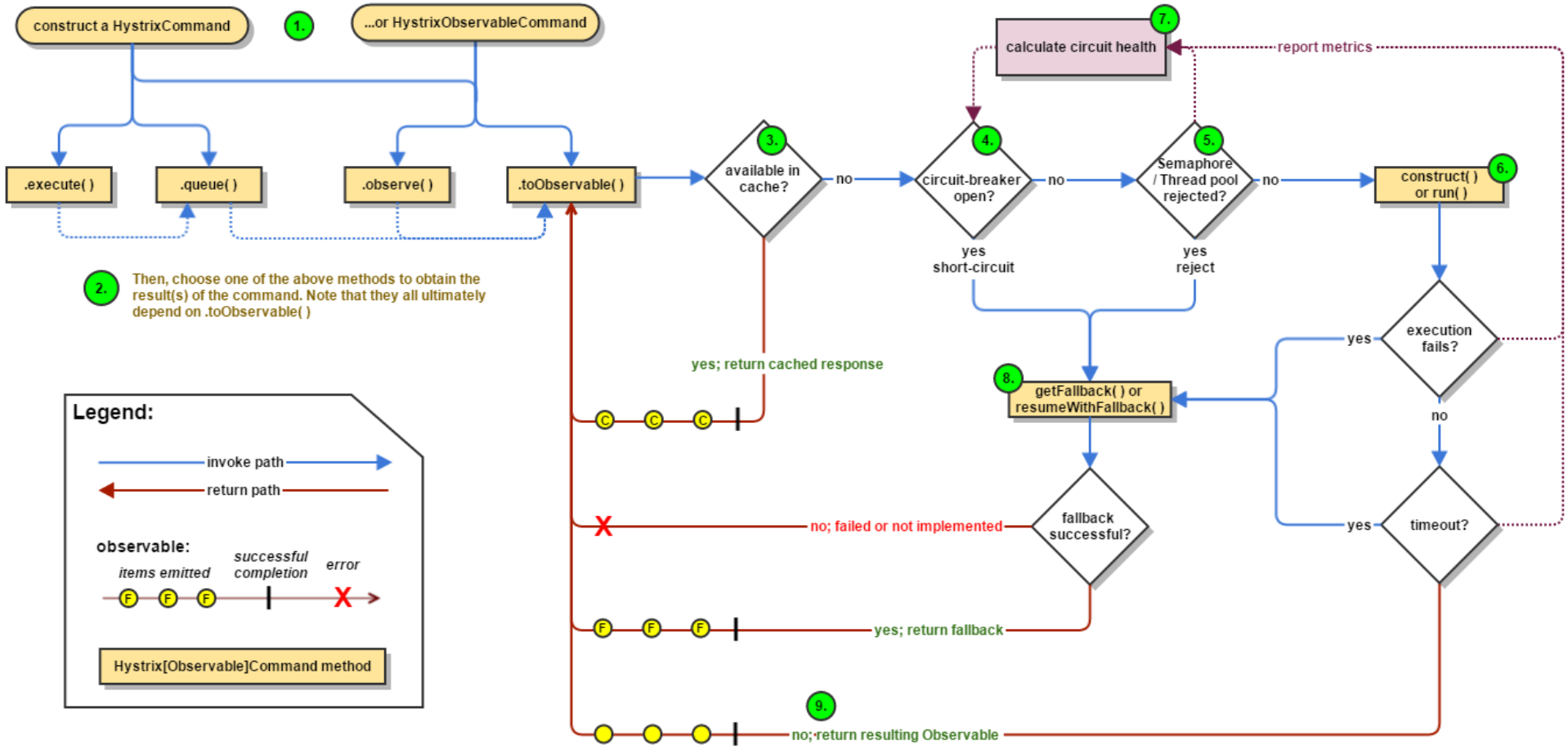
Netflix | 【翻译】Hystrix文档-实现原理

流程图

下图展示了当你使用 Hystrix 来包装你请求依赖服务时的流程：



接下来将详细介绍如下问题：

1. 构建 `HystrixCommand` 或者 `HystrixObservableCommand` 对象
2. 执行命令（即上述 Command 对象包装的逻辑）
3. 结果是否有缓存
4. 请求线路（类似电路）是否是开路
5. 线程池/请求队列/信号量占满时会发生什么
6. 使用 `HystrixObservableCommand.construct()` 还是 `HystrixCommand.run()`
7. 计算链路健康度
8. 失败回退逻辑
9. 返回正常回应

1. 构建 `HystrixCommand` 或者 `HystrixObservableCommand` 对象

使用 Hystrix 的第一步是创建一个 `HystrixCommand` 或者 `HystrixObservableCommand` 对象来表示你需要发给依赖服务的请求。你可以向构造器传递任意参数。

若只期望依赖服务每次返回单一的回应，按如下方式构造一个 `HystrixCommand` 即可：

```
HystrixCommand command = new HystrixCommand(arg1, arg2);
```

若期望依赖服务返回一个 Observable，并应用『Observer』模式监听依赖服务的回应，可按如下方式构造一个 `HystrixObservableCommand` ：

```
HystrixObservableCommand command = new HystrixObservableCommand(arg1, arg2);
```

2. 执行命令

Hystrix 命令提供四种方式（ `HystrixCommand` 支持所有四种方式，而 `HystrixObservableCommand` 仅支持后两种方式）来执行你包装的请求：

- `execute()` —— 阻塞，当依赖服务响应（或者抛出异常/超时）时，返回结果
- `queue()` —— 返回 `Future` 对象，通过该对象异步得到返回结果
- `observe()` —— 返回 `Observable` 对象，立即发出请求，在依赖服务响应（或者抛出异常/超时）时，通过注册的 `Subscriber` 得到返回结果
- `toObservable()` —— 返回 `Observable` 对象，但只有在订阅该对象时，才会发出请求，然后在依赖服务响应（或者抛出异常/超时）时，通过注册的 `Subscriber` 得到返回结果

```
K             value   = command.execute();
Future<K>     fValue  = command.queue();
Observable<K> ohValue = command.observe();         // hot observable（注：调用observe()方法时，请求立即发出）
Observable<K> ocValue = command.toObservable();    // cold observable（注：只有在返回的ocValue上调用subscribe时，才会发出请求）
```

在内部实现中，`execute()` 是同步调用，内部会调用 `queue().get()` 方法。`queue()` 内部会调用 `toObservable().toBlocking().toFuture()`。也就是说，`HystrixCommand` 内部均通过一个 `Observable` 的实现来执行请求，即使这些命令本来是用来执行同步返回回应这样的简单逻辑。

3. 结果是否有缓存

如果请求结果缓存这个特性被启用，并且缓存命中，则缓存的回应会立即通过一个 `Observable` 对象的形式返回。

4. 请求线路是否是开路

当执行一个命令时，Hystrix 会先检查熔断器状态，确定请求线路是否是开路

如果请求线路是开路，Hystrix 将不会执行这个命令，而是直接使用『失败回退逻辑』

5. 线程池/请求队列/信号量占满时会发生什么

如果和当前需要执行的命令相关联的线程池和请求队列（或者信号量，如果不使用线程池），Hystrix 将不会执行这个命令，而是直接使用『失败回退逻辑』

6. 使用 `HystrixObservableCommand.construct()` 还是 `HystrixCommand.run()`

Hystrix 将根据你使用类的不同，内部使用不同的方式来请求依赖服务：

- `HystrixCommand.run()` —— 返回回应或者抛出异常
- `HystrixObservableCommand.construct()` —— 返回 Observable 对象，并在回应到达时通知 observers，或者回调 `onError` 方法通知出现异常

若 `run()` 或者 `construct()` 方法耗时超过了给命令设置的超时阈值，执行请求的线程将抛出 `TimeoutException`（若命令本身并不在其调用线程内执行，则单独的定时器线程会抛出该异常）。在这种情况下，Hystrix 将会执行失败回退逻辑，并且会忽略最终（若执行命令的线程没有被中断）返回的回应。

若命令本身并不抛出异常，并正常返回回应，Hystrix 在添加一些日志和监控数据采集之后，直接返回回应。Hystrix 在使用 `run()` 方法时，Hystrix 内部还是会生成一个 `Observable` 对象，并返回单个请求，产生一个 `onCompleted` 通知；而在 Hystrix 使用 `construct()` 时，会直接返回由 `construct()` 产生的 `Observable` 对象

7. 计算线路健康度

Hystrix 会将请求成功，失败，被拒绝或超时信息报告给熔断器，熔断器维护一些用于统计数据用的计数器。

这些计数器产生的统计数据使得熔断器在特定的时刻，能短路某个依赖服务的后续请求，直到恢复期结束，若恢复期结束根据统计数据熔断器判定线路仍然未恢复健康，熔断器会再次关闭线路。

8. 失败回退逻辑

当命令执行失败时，Hystrix 将会执行失败回退逻辑，失败原因可能是：

1. `construct()` 或 `run()` 方法抛出异常
2. 当线路是开路，导致命令被短路时
3. 当命令对应的线程池或信号量被占满
4. 超时

失败回退逻辑包含了通用的回应信息，这些回应从内存缓存中或者其他固定逻辑中得到，而不应有任何的网络依赖。如果一定要在失败回退逻辑中包含网络请求，必须将这些网络请求包装在另一个 `HystrixCommand` 或 `HystrixObservableCommand` 中。

当使用 `HystrixCommand` 时，通过实现 `HystrixCommand.getFallback()` 返回失败回退时的回应。

当使用 `HystrixObservableCommand` 时，通过实现 `HystrixObservableCommand.resumeWithFallback()` 返回 `Observable` 对象来通知 observers 失败回退时的回应。

若失败回退方法返回回应，Hystrix 会将这个回应返回给命令的调用者。若 Hystrix 内部调用 `HystrixCommand.getFallback()` 时，会产生一个 Observable 对象，并包装用户实现的 `getFallback()` 方法返回的回应；若 Hystrix 内部调用 `HystrixObservableCommand.resumeWithFallback()` 时，会将用户实现的 `resumeWithFallback()` 返回的 Observable 对象直接返回。

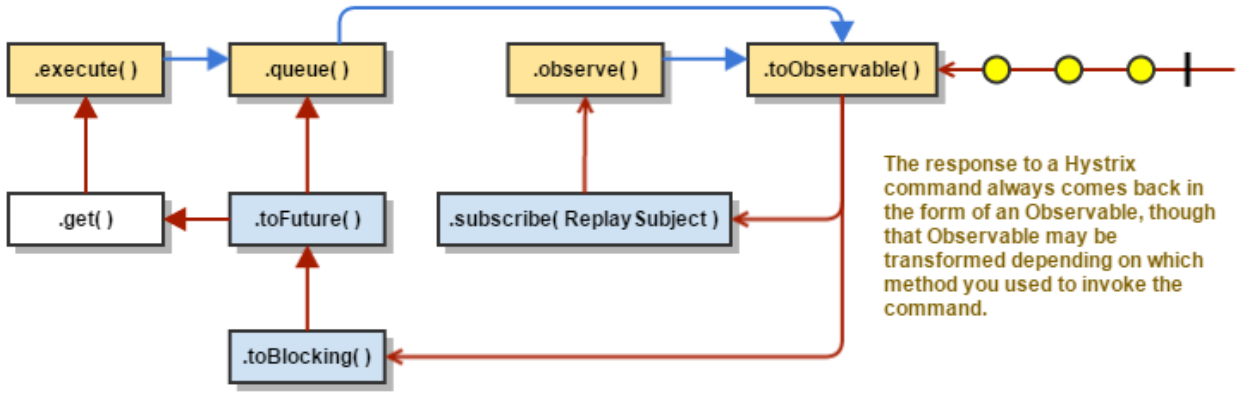
若你没有实现失败回退方法，或者失败回退方法抛出异常，Hystrix 内部还是会生成一个 Observable 对象，但它不会产生任何回应，并通过 `onError` 通知立即中止请求。Hystrix 默认会通过 `onError` 通知调用者发生了何种异常。你需要尽量避免失败回退方法执行失败，保持该方法尽可能的简单不易出错。

若失败回退方法执行失败，或者用户未提供失败回退方法，Hystrix 会根据调用执行命令的方法的不同而产生不同的行为：

- `execute()` —— 抛出异常
- `queue()` —— 成功返回 `Future` 对象，但其 `get()` 方法被调用时，会抛出异常
- `observe()` —— 返回 `Observable` 对象，当你订阅它的时候，会立即调用 subscriber 的 `onError` 方法中止请求
- `toObservable()` —— 返回 `Observable` 对象，当你订阅它的时候，会立即调用 subscriber 的 `onError` 方法中止请求

9. 返回正常回应

若命令成功被执行，Hystrix 将回应返回给调用方，或者通过 `Observable` 的形式返回。根据上述调用命令方式的不同（如第2条所示），`Observable` 对象会进行一些转换：

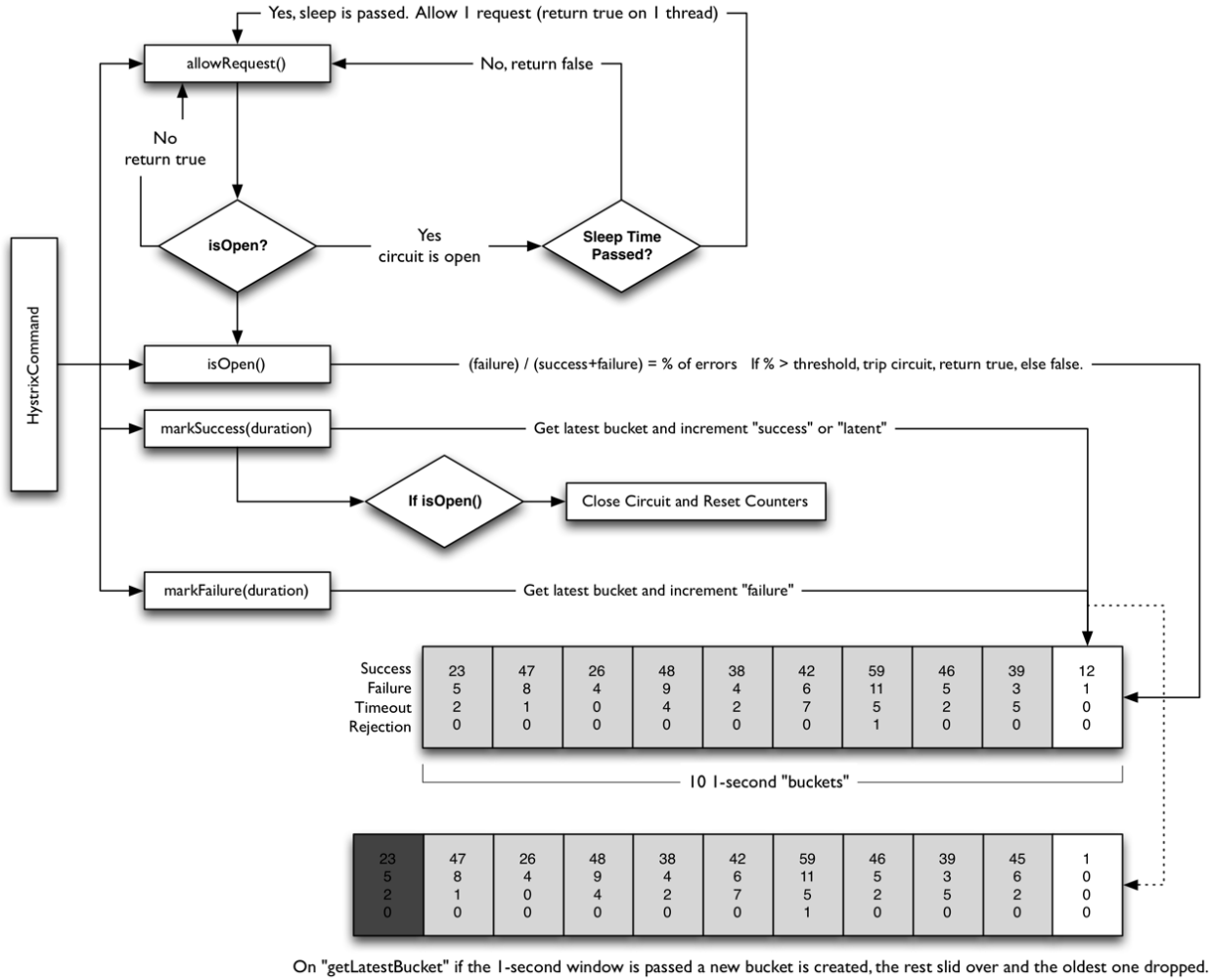


- `execute()` —— 产生一个 `Future` 对象，行为同 `.queue()` 产生的 `Future` 对象一样，接着调用其 `get()` 方法，生成由内部产生的 `Observable` 对象返回的回应
- `queue()` —— 将内部产生的 `Observable` 对象转换（Decorator模式）成 `BlockingObservable` 对象，以产生并返回 `Future` 对象

- `observe()` —— 产生 `Observable` 对象后，立即订阅（ReplaySubject）以使命令得以执行（异步），返回该 `Observable` 对象，当你调用其 `subscribe` 方法时，重放产生的回应信息和通知给用户提供的订阅者
- `toObservable()` —— 返回 `Observable` 对象，你必须调用其 subscribe 方法，以使命令得以执行。

熔断器

下图展示了 `HystrixCommand` 或 `HystrixObservableCommand` 如何与 `HystrixCircuitBreaker` 进行交互，以及 `HystrixCircuitBreaker` 的决策逻辑流程，包括熔断器内部计数器如何工作。



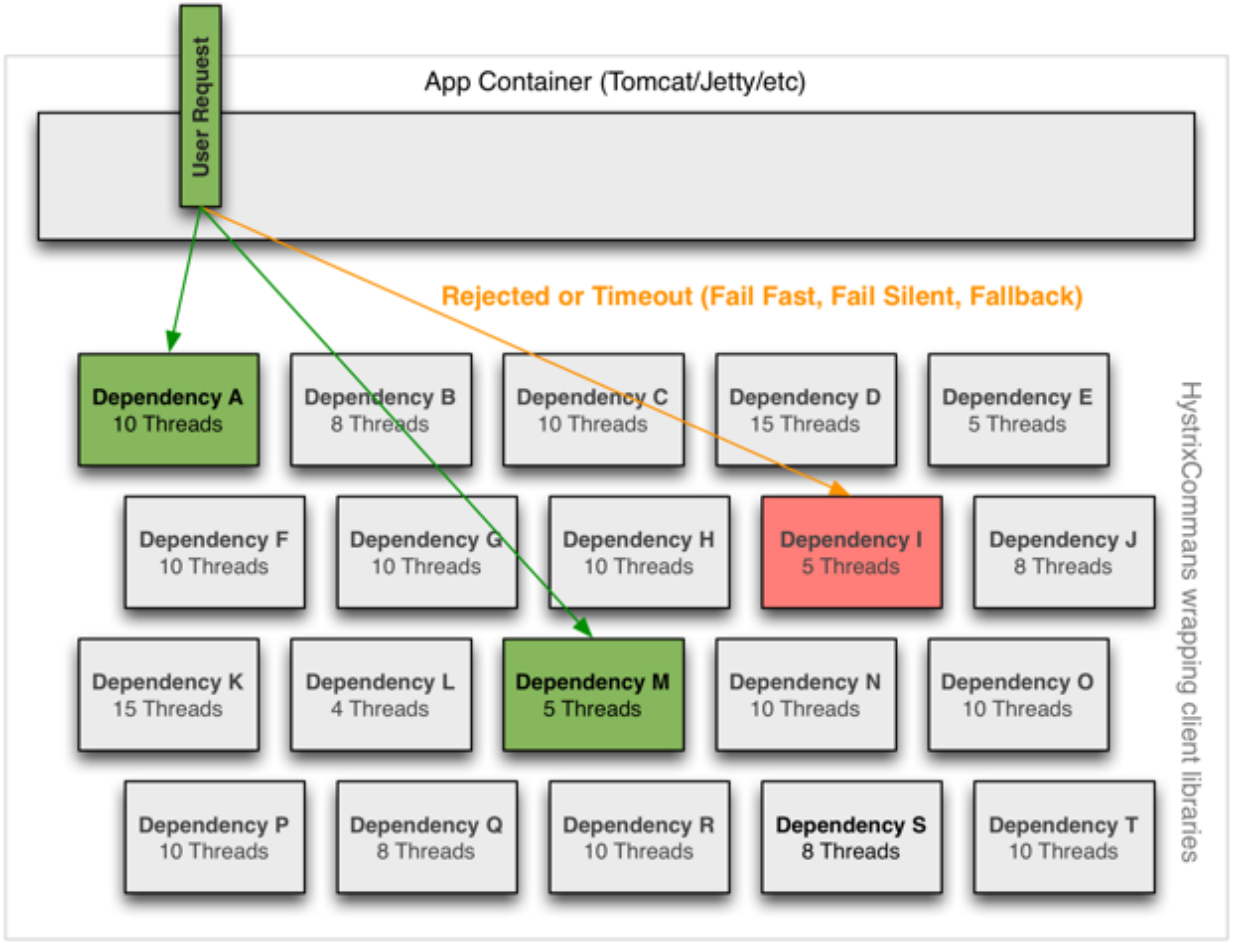
线路的开路闭路详细逻辑如下：

1. 假设线路内的容量（请求QPS）达到一定阈值（通过 `HystrixCommandProperties.circuitBreakerRequestVolumeThreshold()` 配置）
2. 同时，假设线路内的错误率达到一定阈值（通过`HystrixCommandProperties.circuitBreakerErrorThresholdPercentage()` 配置）
3. 熔断器将从『闭路』转换成『开路』

4. 若此时是『开路』状态，熔断器将短路后续所有经过该熔断器的请求，这些请求直接走『失败回退逻辑』
5. 经过一定时间（即『休眠窗口』，通过 `HystrixCommandProperties.circuitBreakerSleepWindowInMilliseconds()` 配置），后续第一个请求将会被允许通过熔断器（此时熔断器处于『半开』状态），若该请求失败，熔断器将又进入『开路』状态，且在休眠窗口内保持此状态；若该请求成功，熔断器将进入『闭路』状态，回到逻辑1循环往复。

依赖隔离

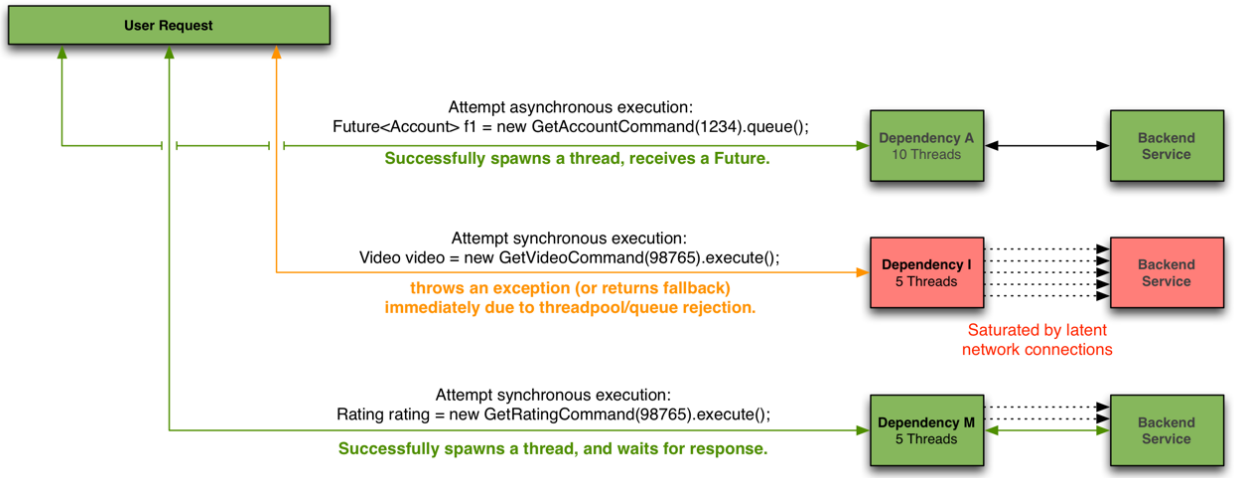
Hystrix 通过使用『舱壁模式』（注：将船的底部划分成一个个的舱室，这样一个舱室进水不会导致整艘船沉没。将系统所有依赖服务隔离起来，一个依赖延迟升高或者失败，不会导致整个系统失败）来隔离依赖服务，并限制访问这些依赖服务的并发度。



线程 & 线程池

通过将对依赖服务的访问执行放到单独的线程，将其与调用线程（例如 Tomcat 线程池中的线程）隔离开来，调用线程能空出来去做其他的工作而不至于被依赖服务的访问阻塞过长时间。

Hystrix 使用独立的，每个依赖服务对应一个线程池的方式，来隔离这些依赖服务，这样，某个依赖服务的高延迟只会拖慢这个依赖服务对应的线程池。

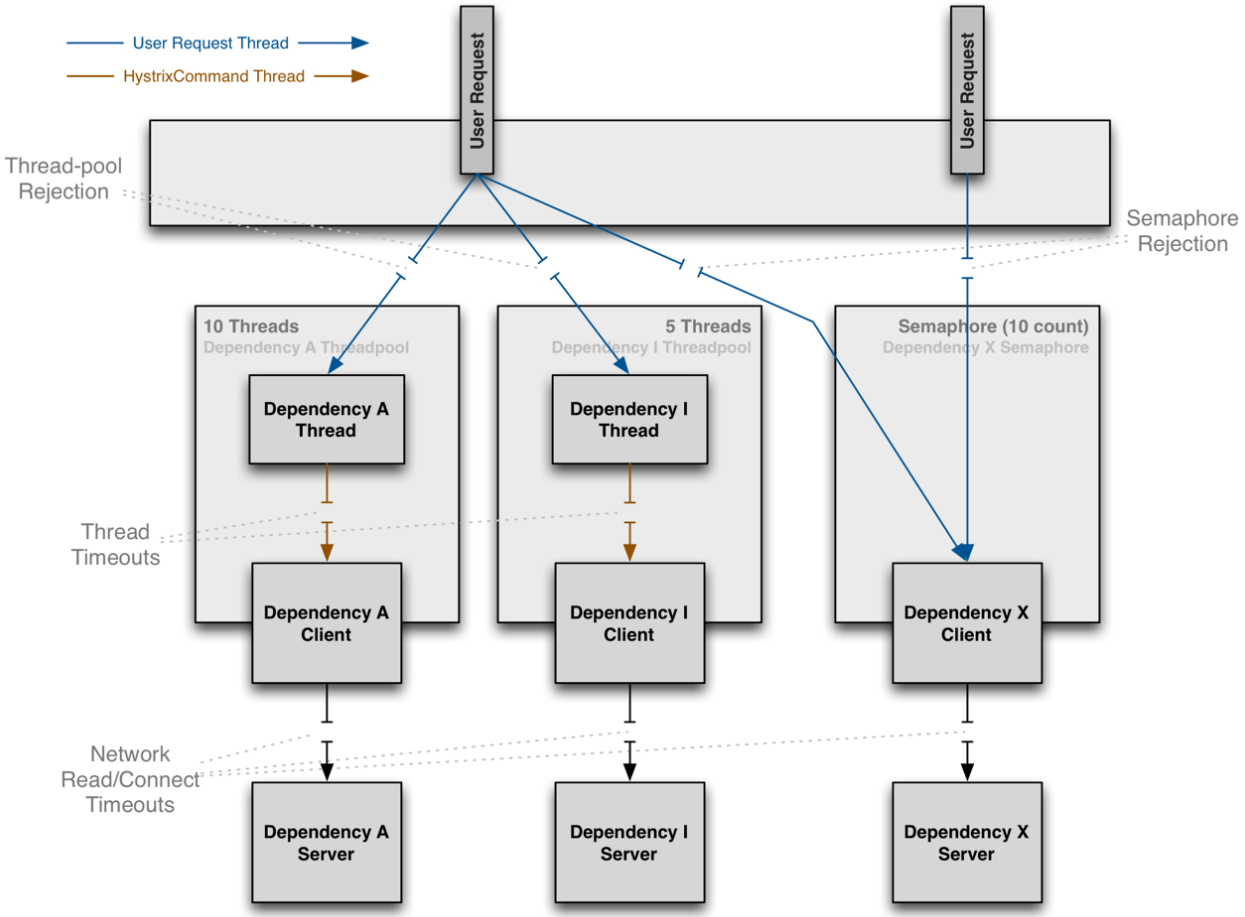


当然，也可以不使用线程池来使你的系统免受依赖服务失效的影响，这需要你小心的设置网络连接/读取超时时间和重试配置，并保证这些配置能正确正常的运作，以使这些依赖服务在失效时，能快速返回错误。

Netflix 在设计 Hystrix 时，使用线程/线程池来实现隔离，原因如下：

- 多数系统同时运行了（有时甚至多达数百个）不同的后端服务，这些服务由不同开发组开发
- 每个服务都提供了自己的客户端库
- 客户端库经常会发生变动
- 客户端库可能会改变逻辑，加入新的网络请求
- 客户端库可能会包含重试逻辑，数据解析，缓存（本地缓存或分布式缓存），或者其他类似逻辑
- 客户端库对于使用者来说，相当于『黑盒』，其实现细节，网络访问方式，默认配置等等均对使用者透明
- In several real-world production outages the determination was “oh, something changed and properties should be adjusted” or “the client library changed its behavior.”

- 即使客户端库本身未发生变化，服务自身发生变化，也可能会影响其性能，从而导致客户端配置不再可靠
- 中间依赖服务可能包含一些其依赖服务提供的客户端库，而这些库可能不受控且配置不合理
- 绝大多数网络访问都采用同步的方式进行
- 客户端代码可能也会有失效或者高延迟，而不仅仅是在网络访问时



线程池的优势

将依赖服务请求通过使用不同的线程池隔离，其优势如下：

- 系统完全与依赖服务请求隔离开来，即使依赖服务对应线程池耗尽，也不会影响系统其它请求
- 降低了系统接入新的依赖服务的风险，若新的依赖服务存在问题，也不会影响系统其它请求
- 当依赖服务失效后又恢复正常，其对应的线程池会被清理干净，相对于整个 Tomcat 容器的线程池被占满需要耗费更长时间以恢复可用来说，此时系统可以快速恢复
- 若依赖服务的配置有问题，线程池能迅速反映出来（通过失败次数的增加，高延迟，超时，拒绝访问等等），同时，你可以在不影响系统现有功能的情况下，处理这些问题（通常通过热配置等方式）
- 若依赖服务的实现发生变更，性能有了很大的变化（这种情况时常发生），需要进行配置调整（例如增加/减小超时阈值，调整重试策略等）时，也可以从线程池的监控信息上迅速反映出来（失败次数增加，高延迟，超时，拒绝访问等等），同时，你可以在不影响其他依赖服务，系统请求和用户的情况下，处理这些问题
- 线程池处理能起到隔离的作用以外，还能通过这种内置的并发特性，在客户端库同步网络IO上，建立一个异步的 Facade（类似 Netflix API 建立在 Hystrix 命令上的 Reactive、全异步化的那一套 Java API）

简而言之，通过线程池提供的依赖服务隔离，可以使得我们能在不停止服务的情况下，更加优雅地应对客户端库和子系统性能上的变化。

> 注：尽管线程池能提供隔离性，但你仍然需要对你的依赖服务客户端代码增加超时逻辑，并且/或者处理线程中断异常，以使这些代码不会无故地阻塞或者拖慢 Hystrix 线程池。

线程池的弊端

使用线程池的主要弊端是会增加系统 CPU 的负载，每个命令的执行，都包含了 CPU 任务的排队，调度，上下文切换。

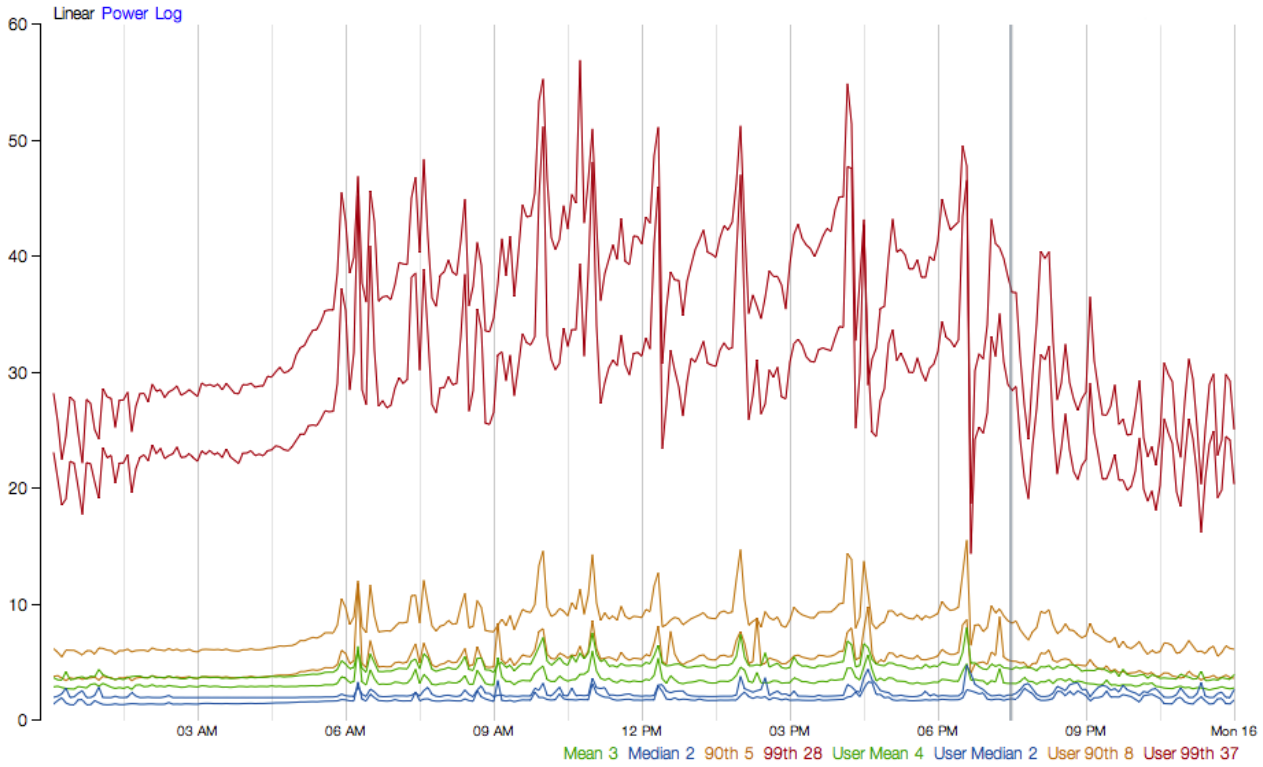
Netflix 在设计 Hystrix 时，认为相对于其带来的好处，其带来的负载的一点点升高对系统的影响是微乎其微的。

线程池的开销

Hystrix 的开发人员测试了在子线程中执行 `construct()` 或 `run()` 方法带来的额外时延，以及在父线程中整个请求的耗时，通过这个测试，你能直观了解 Hystrix 使用线程池带来的一点点系统负载的升高影响（线程，监控，日志，熔断器等）。

Netflix API 使用线程池来隔离依赖服务，每天可以处理超过 100 亿的 Hystrix 命令，每个 API 实例有超过 40 个线程池，每个线程池有 5 到 20 个工作线程（绝大部分设置为 10 个线程）。

下图展示了一个 HystrixCommand 以 60QPS 的速度，在一个 API 实例（每台服务器每秒运行的线程数峰值为 350）上被执行的耗时监控：



（注：有 User 的表示使用线程池来隔离依赖服务后的耗时）

中位数显示二者（未使用线程池和使用线程池）没有差别。

90% 的情况下，使用线程池有 3ms 的延迟

99% 的情况下，使用线程池有 9ms 的延迟，尽管如此，相对于请求的总时间（2ms$_{28ms}$），延迟（0ms9ms）基本可以忽略不计

90% 的情况下，这些延迟和在使用了熔断器之后更高的延迟，在绝大多数 Netflix 的需求来看，是微不足道的，更何况其能带来系统稳定性和鲁棒性上的巨大提升。

对于那些本来延迟就比较小的请求（例如访问本地缓存成功率很高的请求）来说，线程池带来的开销是非常高的，这时，你可以考虑采用其他方法，例如非阻塞信号量（不支持超时），来实现依赖服务的隔离，使用信号量的开销很小。但绝大多数情况下，Netflix 更偏向于使用线程池来隔离依赖服务，因为其带来的额外开销可以接受，并且能支持包括超时在内的所有功能。

信号量

除了线程池，队列之外，你可以使用信号量（或者叫计数器）来限制单个依赖服务的并发度。Hystrix 可以利用信号量，而不是线程池，来控制系统负载，但信号量不允许我们设置超时和异步化，如果你对客户端库有足够的信任（延迟不会过高），并且你只需要控制系统负载，那么你可以使用信号量。

`HystrixCommand` 和 `HystrixObservableCommand` 在两个地方支持使用信号量：

- 失败回退逻辑：当 Hystrix 需要执行失败回退逻辑时，其在调用线程（Tomcat 线程）中使用信号量
- 执行命令时：如果设置了 Hystrix 命令的 `execution.isolation.strategy` 属性为 `SEMAPHORE` ，则 Hystrix 会使用信号量而不是线程池来控制调用线程调用依赖服务的并发度

你可以通过动态配置（即热部署）来决定信号量的大小，以控制并发线程的数量，信号量大小的估计和使用线程池进行并发度估计一样（仅访问内存数据的请求，一般能达到耗时在 1ms 以内，且能达到 5000rps，这样的请求对应的信号量可以设置为 1 或者 2。默认值为 10）。

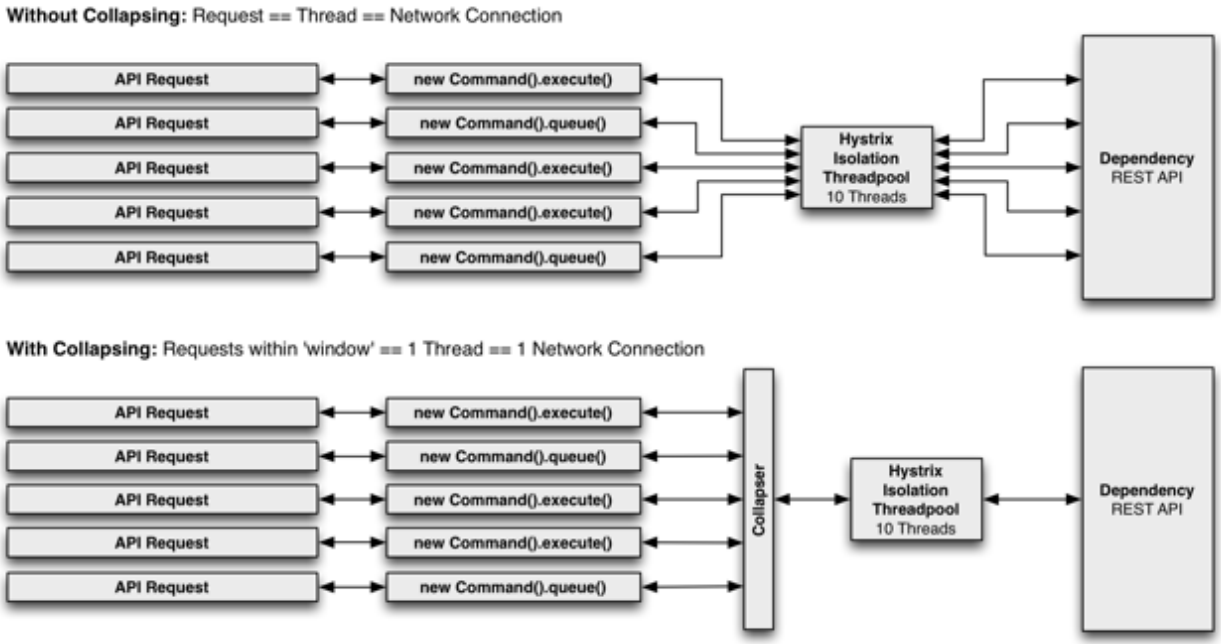
> 注意：如果依赖服务使用信号量来进行隔离，当依赖服务出现高延迟，其调用线程也会被阻塞，直到依赖服务的网络请求超时。

信号量在达到上限时，会拒绝后续请求的访问，同时，设置信号量的线程也无法异步化（即像线程池那样，实现『提交-做其他工作-得到结果』模式）

请求合并

你可以在 `HystrixCommand` 之前放置一个『请求合并器』（ `HystrixCollapser` 为请求合并器的抽象父类），该合并器可以将多个发往同一个后端依赖服务的请求合并成一个。

下图展示了在两种场景（未增加『请求合并器』和增加『请求合并器』）下，线程和网络连接数量（假设所有请求在一个很小的时间窗口内，例如 10ms，是『并发』的）：



为什么要使用请求合并？

在并发执行 `HystrixCommand` 时，利用请求合并能减少线程和网络连接数量。通过使用 `HystrixCollapser` ，Hystrix 能自动完成请求的合并，开发者不需要对现有代码做批量化的开发。

全局上下文（适用于所有 Tomcat 线程）

理想情况下，合并过程应该发生在系统全局层面，这样用户发起的，由 Tomcat 线程执行的所有请求都能被执行合并操作。

例如，有这样一个需求，用户需要获取电影评级，而这些数据需要系统请求依赖服务来获取，对依赖服务的请求使用 `HystrixCommand` 进行包装，并增加了请求合并的配置，这样，当同一个 JVM 中其他线程需要执行同样的请求时，Hystrix 会将这个请求同其他同样的请求合并，只产生一个网络请求。

> 注意：合并器会传递一个 `HystrixRequestContext` 对象到合并的网络请求中，因此，下游系统需要支持批量化，以使请求合并发挥其高效的特点。

用户请求上下文（适用于单个 Tomcat 线程）

如果给 `HystrixCommand` 只配置成针对单个用户进行请求合并，则 Hystrix 只会在单个 Tomcat 线程（即请求）中进行请求合并。

例如，如果用户想加载 300 个视频对象的书签，请求合并后，Hystrix 会将原本需要发起的 300 个网络请求合并到一个。

对象模型和代码复杂度

很多时候，当你创建一个对象模型，适用于对象的消费者逻辑，结果发现这个模型会导致生产者无法充分利用其拥有的资源。

例如，这里有一个包含 300 个视频对象的列表，需要遍历这个列表，并对每一个对象调用 `getSomeAttribute()` 方法，这是一个显而易见的对象模型，但如果简单处理的话，可能会导致 300 次的网络请求（假设 `getSomeAttribute()` 方法内需要发出网络请求），每一个网络请求可能都会花上几毫秒（显然，这种方式非常容易拖慢系统）。

当然，你也可以要求用户在调用 `getSomeAttribute()` 之前，先判断一下哪些视频对象真正需要请求其属性。

或者，你可以将对象模型进行拆分，从一个地方获取视频列表，然后从另一个地方获取视频的属性。

但这些实现会导致 API 非常丑陋，且实现的对象模型无法完全满足用户使用模式。 并且在企业级开发时，很容易因为开发者的疏忽导致错误或者不够高效，因为不同的开发者可能有不同的请求方式，这样一个地方的优化不足以保证在所有地方都会有优化。

通过将合并逻辑下沉到 Hystrix 层，不管你如何设计对象模型，或者以何种方式去调用依赖服务，又或者开发者是否意识到这些逻辑需要不需要进行优化，这些都不需要考虑，因为 Hystrix 能统一处理。

`getSomeAttribute()` 方法能放在它最适合的位置，并且能以最适合的方式被调用，Hystrix 的请求合并器会自动将请求合并到合并时间窗口内。

请求合并带来的额外开销

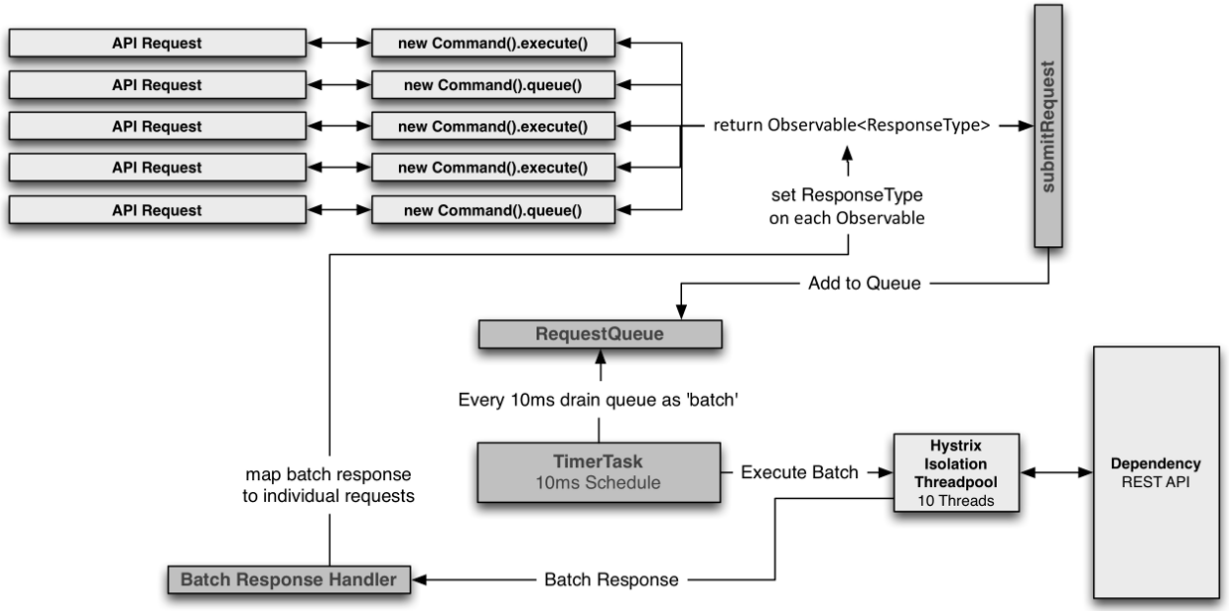
请求合并会导致依赖服务的请求延迟增高（该延迟为等待请求的延迟），延迟的最大值为合并时间窗口大小。

若某个请求耗时的中位数是 5ms，合并时间窗口为 10ms，那么在最坏情况下（注：合并时间窗口开启时发起请求），请求需要消耗 15ms 才能完成。通常情况下，请求不太可能恰好在合并时间窗口开启时发起，因此，请求合并带来的额外开销应该是合并时间窗口的一般，在此例中是 5ms。

请求合并带来的额外开销是否值得，取决于将要执行的命令，高延迟的命令相比较而言不会有太大的影响。同时，缓存 Key 的选择也决定了在一个合并时间窗口内能『并发』执行的命令数量：如果一个合并时间窗口内只有 1~2 个请求，将请求合并显然不是明智的选择。事实上，如果单线程循环调用同一个依赖服务的情况下，如果将请求合并，会导致这个循环成为系统性能的瓶颈，因为每一个请求都需要等待 10ms 的合并时间周期。

然而，如果一个命令具有高并发度，并且能批量处理多个，甚至上百个的话，请求合并带来的性能开销会因为吞吐量的极大提升而基本可以忽略，因为 Hystrix 会减少这些请求所需的线程和网络连接数量。

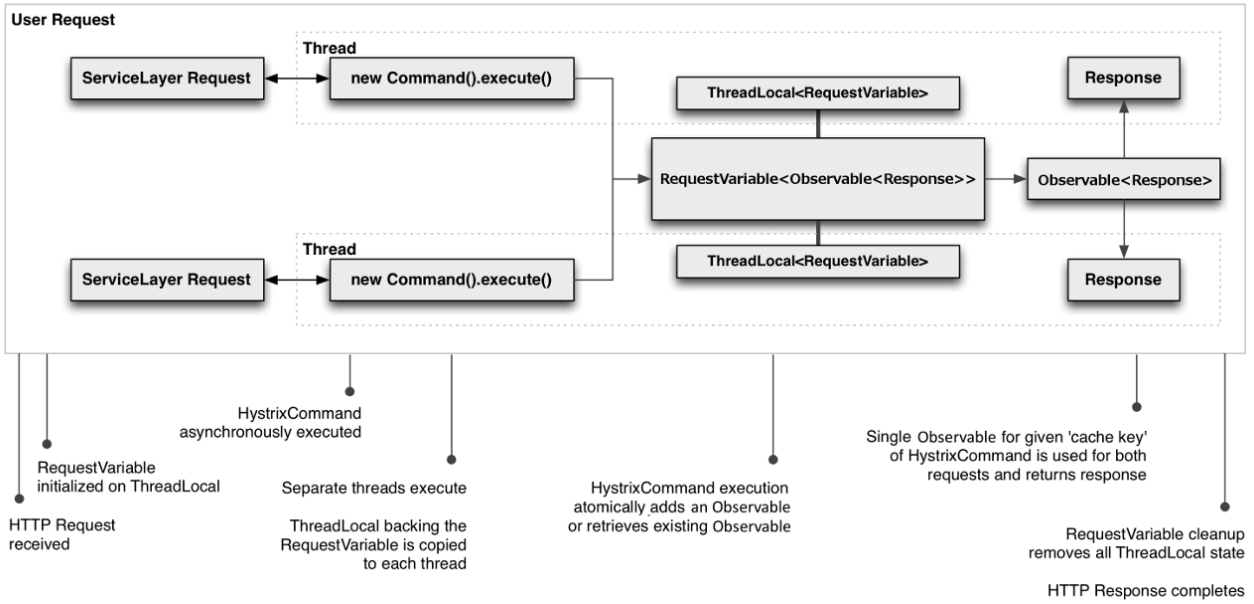
请求合并器的执行流程



请求缓存

在 `HystrixCommand` 和 `HystrixObservableCommand` 的实现中，你可以定义一个缓存的 Key，这个 Key 用于在同一个请求上下文（全局或者用户级）中标识缓存的请求结果，当然，该缓存是线程安全的。

下例展示了在一个完整 HTTP 请求周期内，两个线程执行命令的流程：



请求缓存有如下好处：

- 不同请求路径上针对同一个依赖服务进行的重复请求（有同一个缓存 Key），不会真实请求多次

这个特性在企业级系统中非常有用，在这些系统中，开发者往往开发的只是系统功能的一部分。（注：这样，开发者彼此隔离，不太可能使用同样的方法或者策略去请求同一个依赖服务提供的资源）

例如，请求一个用户的 `Account` 的逻辑如下所示，这个逻辑往往在系统不同地方被用到：

```
Account account = new UserGetAccount(accountId).execute();
//or
Observable<Account> accountObservable = new UserGetAccount(accountId).observe();
```

Hystrix 的 `RequestCache` 只会在内部执行 run() 方法一次，上面两个线程在执行 `HystrixCommand` 命令时，会得到相同的结果，即使这两个命令是两个不同的实例。

- 数据获取具有一致性

因为缓存的存在，除了第一次请求需要真正访问依赖服务以外，后续请求全部从缓存中获取，可以保证在同一个用户请求内，不会出现依赖服务返回不同的回应的情况。

- 避免不必要的线程执行

在 `construct()` 或 `run()` 方法执行之前，会先从请求缓存中获取数据，因此，Hystrix 能利用这个特性避免不必要的线程执行，减小系统开销。

若 Hystrix 没有实现请求缓存，那么 `HystrixCommand` 和 `HystrixObservableCommand` 的实现者需要自己在 `construct()` 或 `run()` 方法中实现缓存，这种方式无法避免不必要的线程执行开销。