云原生社区
Cloud Native Community

+

2021

死生之地不可不察：

论API标准化对Dapr的重要性

演讲人：敖小剑 阿里云

云原生社区Meetup
第四期·广州站

目录

CONTENTS

零、快速回顾：什么是Dapr?

Distributed Application Runtime

分布式应用运行时

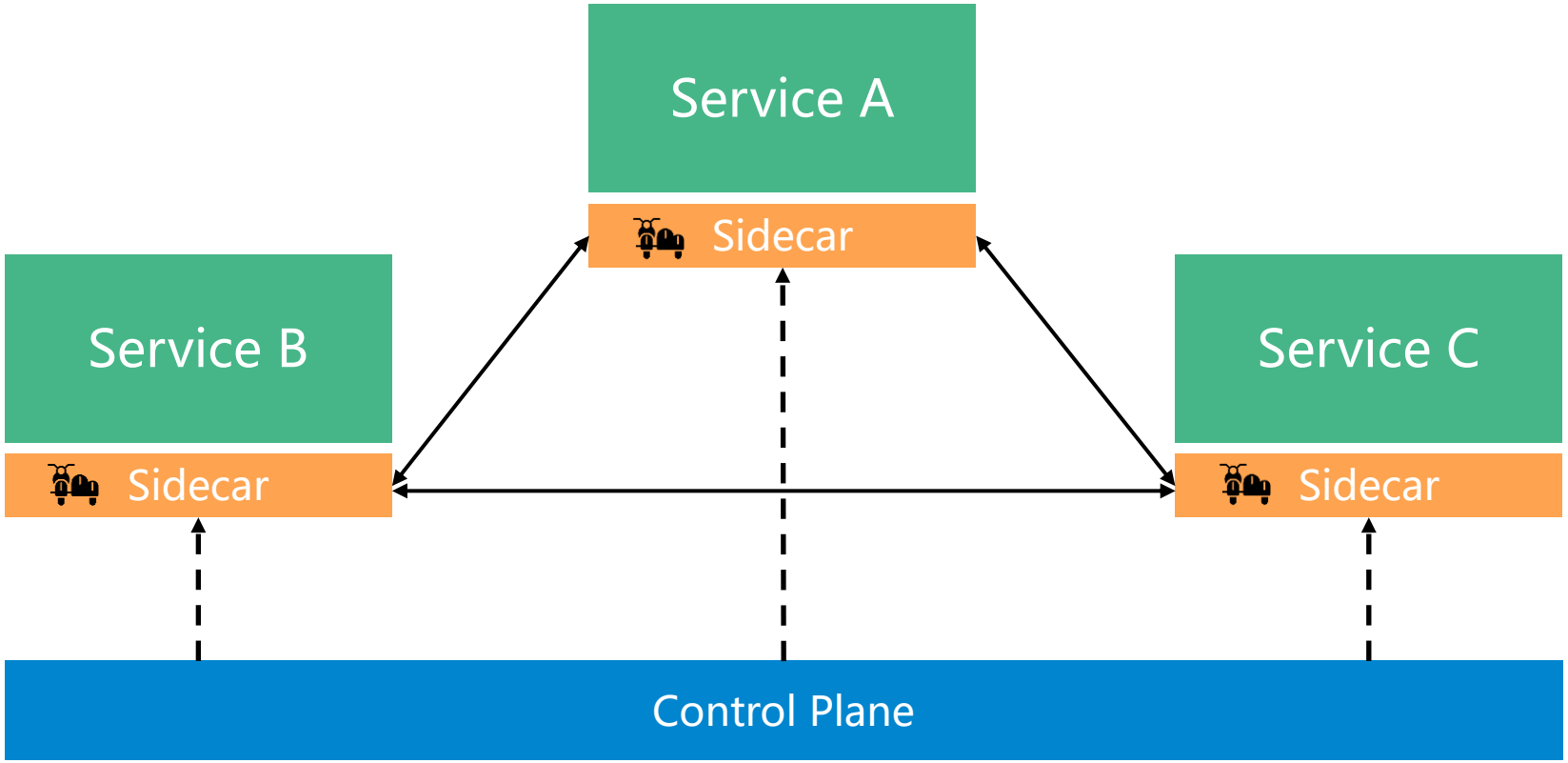
假定：大家对Dapr已有初步的了解，或参阅前次演讲

云原生社区 Cloud Native Community

或参阅前次演讲：

Dapr v1.0展望：从servicemesh到云原生

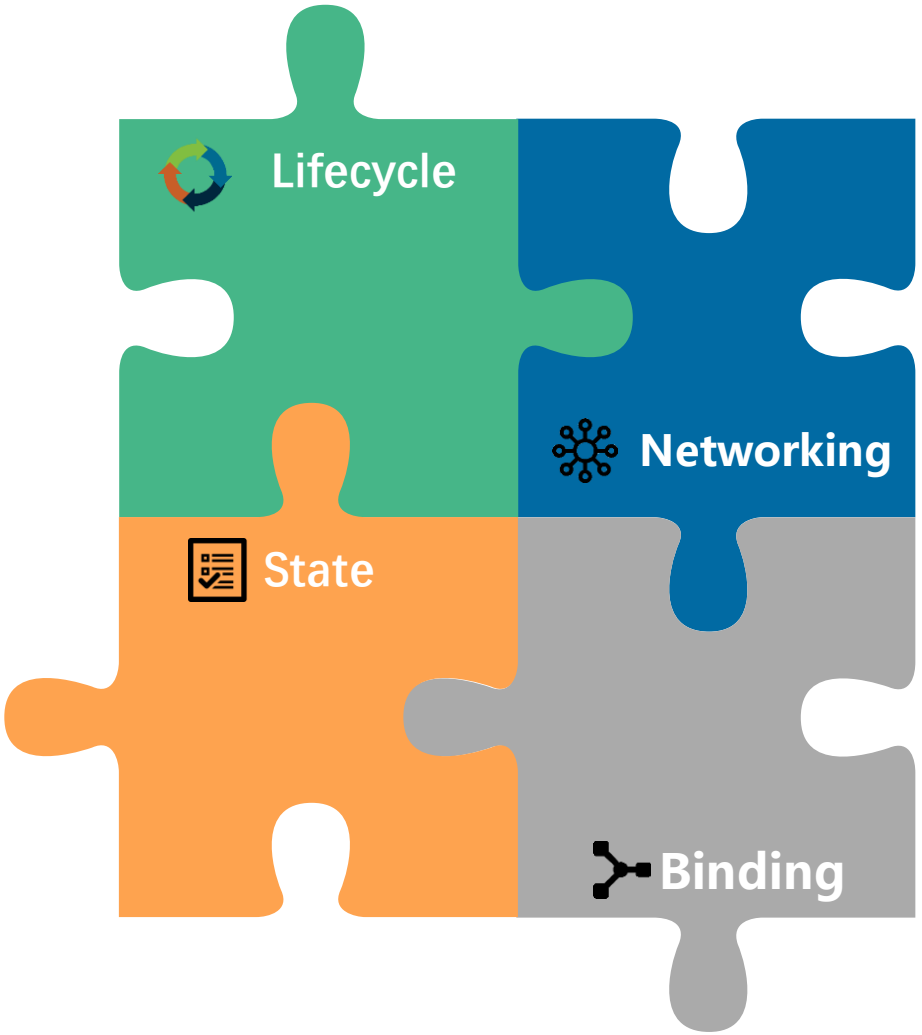
ServiceMesh的创新：引入 Sidecar 模式



分布式应用存在更多的需求： Multi-Runtime归结为四大类

生命周期

- Package
- Health check
- Deployment
- Scaling
- Configuration



网络

- Service discovery
- A/B testing, canary rollouts
- Retry, timeout, circuit breaker
- **Point-to-Point**, pub/sub
- Security, observability

状态

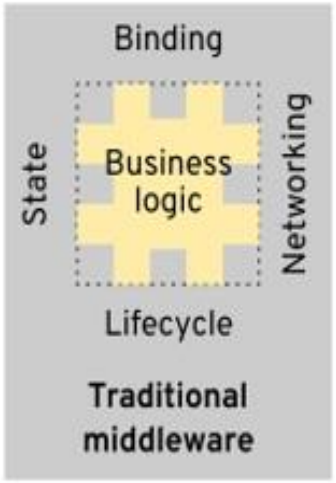
- Workflow mgmt.
- Idempotency
- Temporal scheduling
- Caching
- Application state

绑定

- Connectors
- Protocol conversion
- Message transformation
- Message routing
- Transnationality

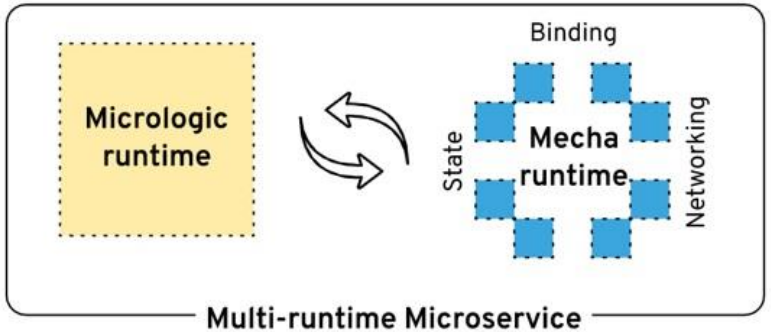
理论推导：效仿Servicemesh，能力外移并整合为Runtime

传统中间件模式



步骤1：能力外移到各种Runtime

步骤2：多个runtime整合

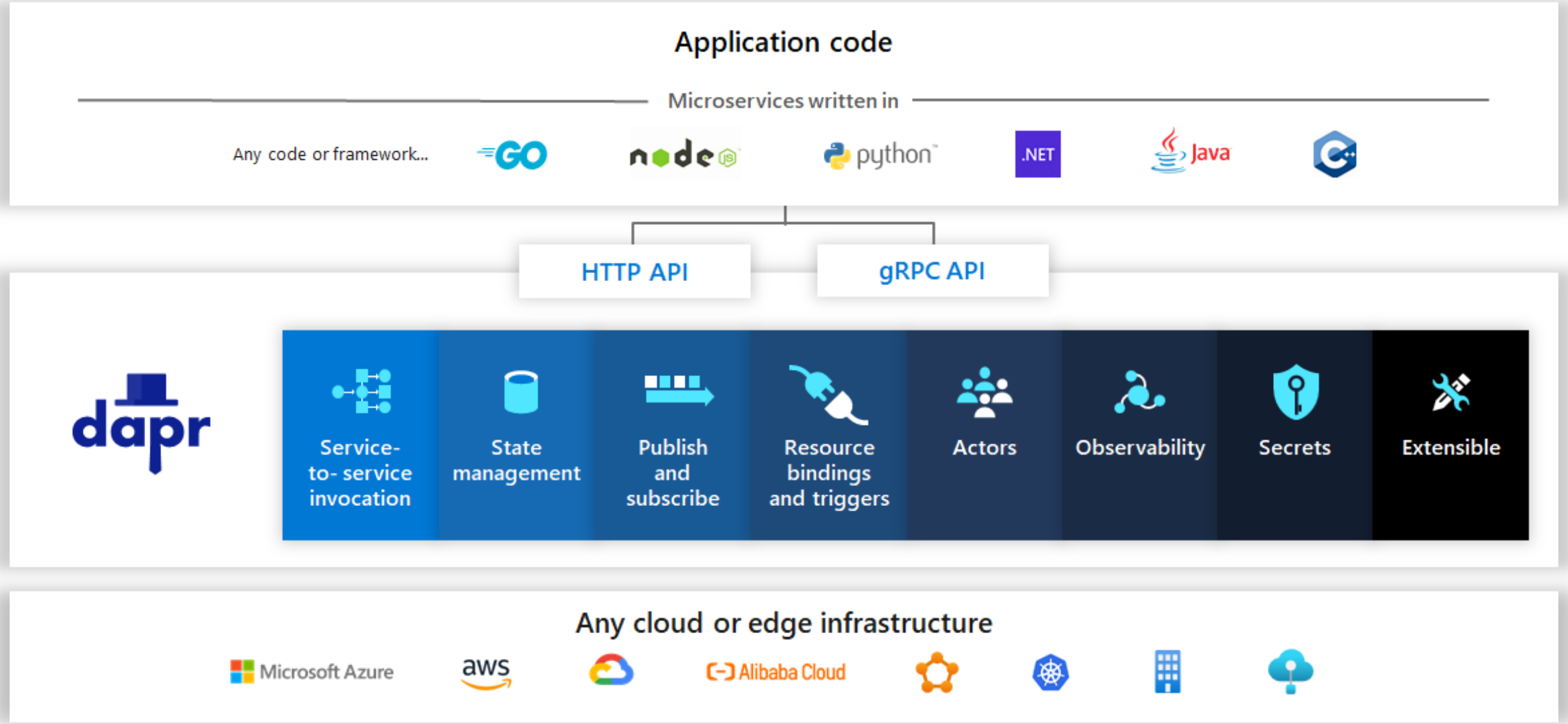


Micrologic	Mecha
• Developed in-house	• Off-the-shelf mechanincs
• Custom business logic	• Configurable capabilities
• Higher-level language	• Declarative (YAML, JSON)
• HTTP/gRPC, CloudEvents	• OpenAPI, AsyncAPI, SQL

Mecha Runtime 和应用 Runtime 共同组成微服务

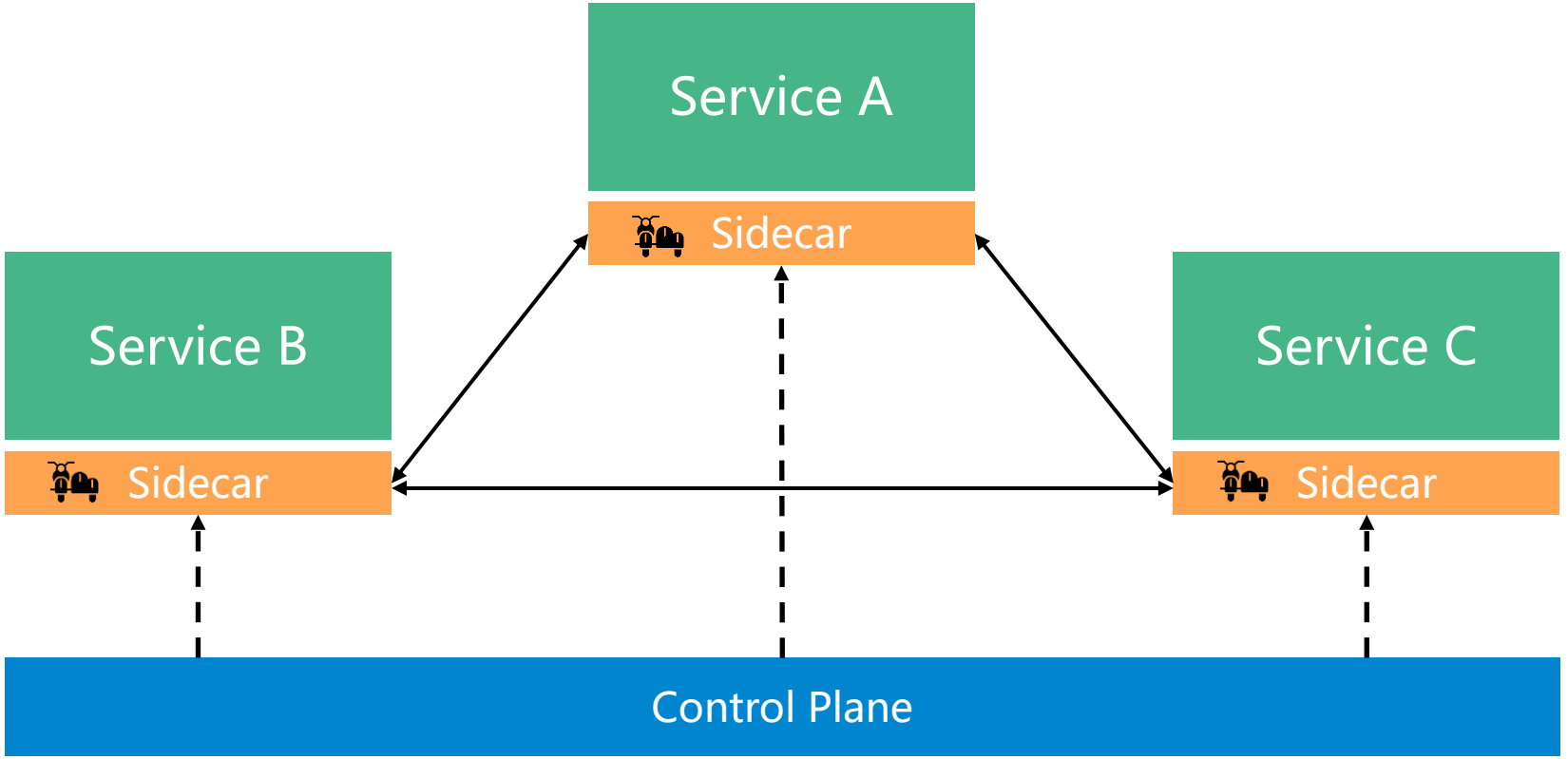
Dapr 项目： Any language, any framework, anywhere

注意：划重点，后面要展开

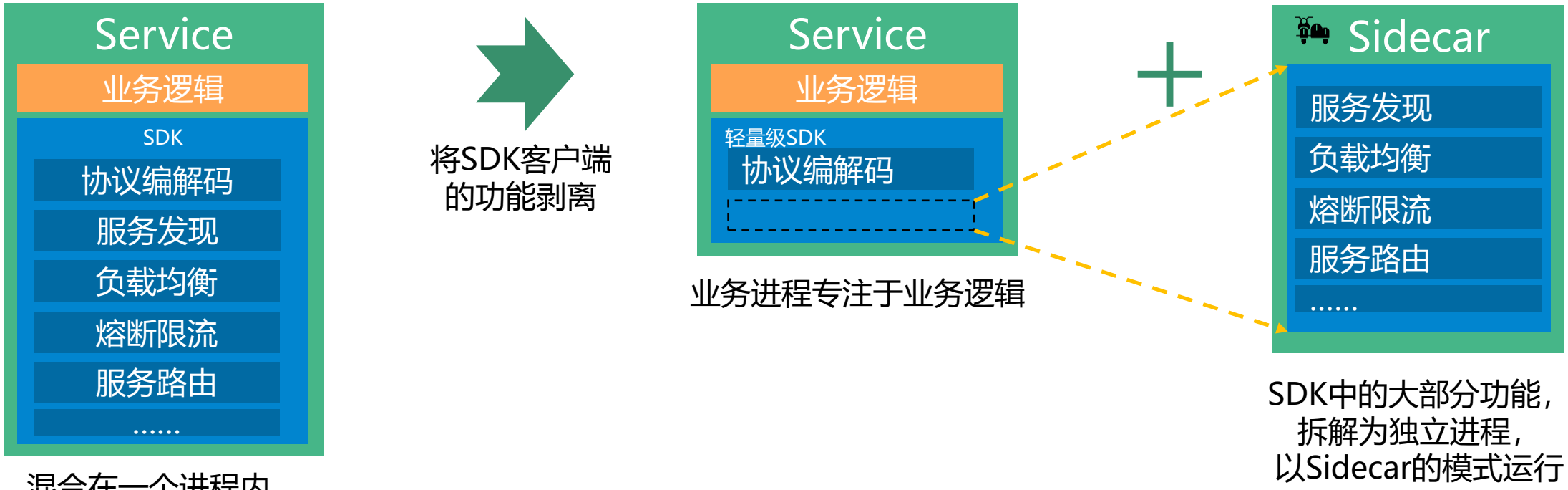


一、本质差别：Dapr vs ServiceMesh

相同点：以 Sidecar 模式为核心



基本思路一致：关注点分离 + 独立维护

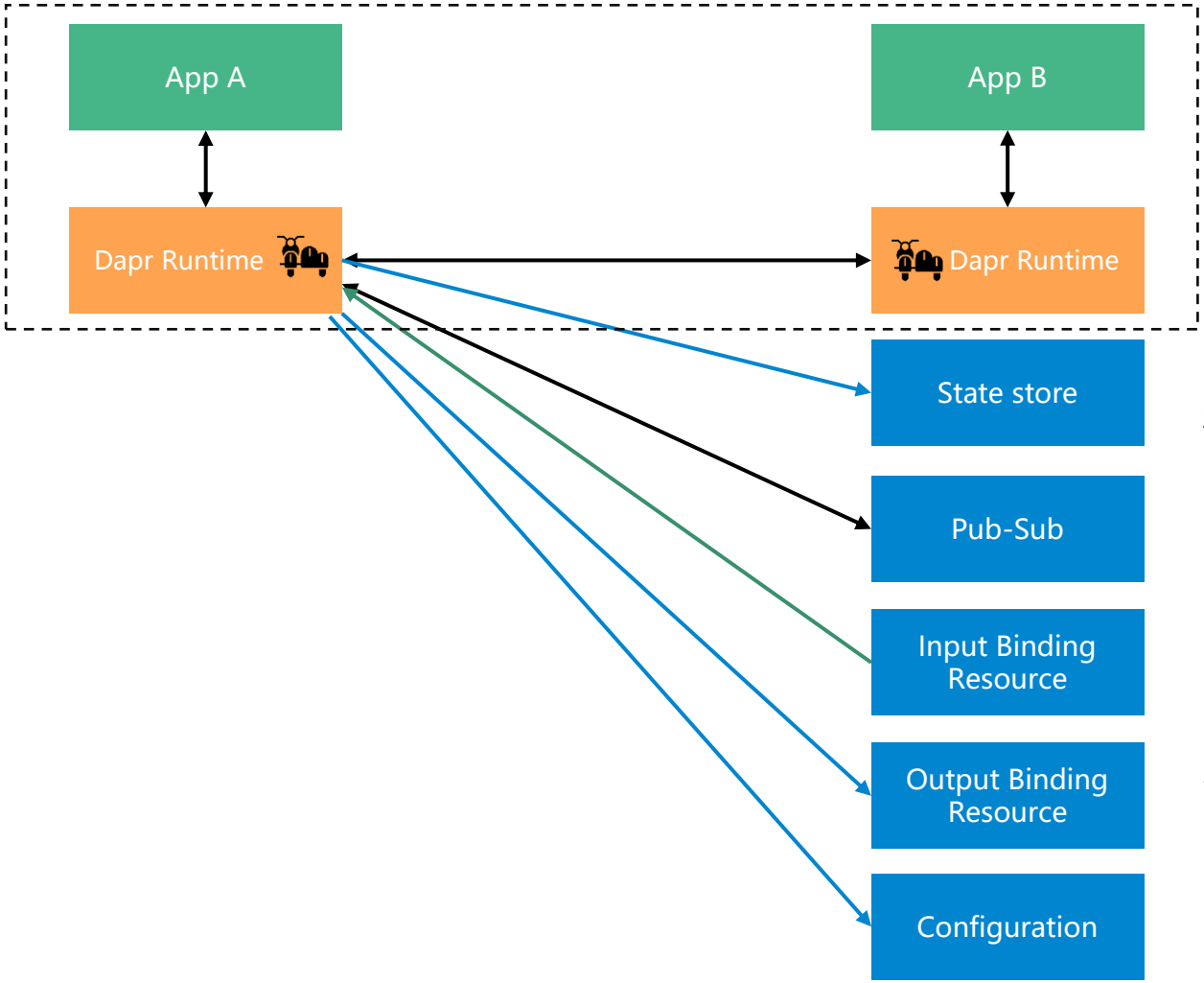


混合在一个进程内，
应用既有业务逻辑，
也有各种非业务的功能

业务进程专注于业务逻辑

SDK中的大部分功能，
拆解为独立进程，
以Sidecar的模式运行

最明显的不同：Dapr 的场景比 ServiceMesh 要复杂



Redis / consul / cassandra / Memcached / mongodb / zookeeper / Gcp.firestore / aws dynamodb

Redis / kafka / rabbimq / pulsar / mqtt / hazelcast / Gcp.pubsub / azure servicebus

kafka / rabbimq
Azure eventbus, aws kinesis

Alicloud.oss / aws sqs / Gcp.bucket/ aws.s3 / Mysql / postgres / twitter / http / smtp

Redis / consul / Appolo / Nacos

洞察跨组件和服务的调用

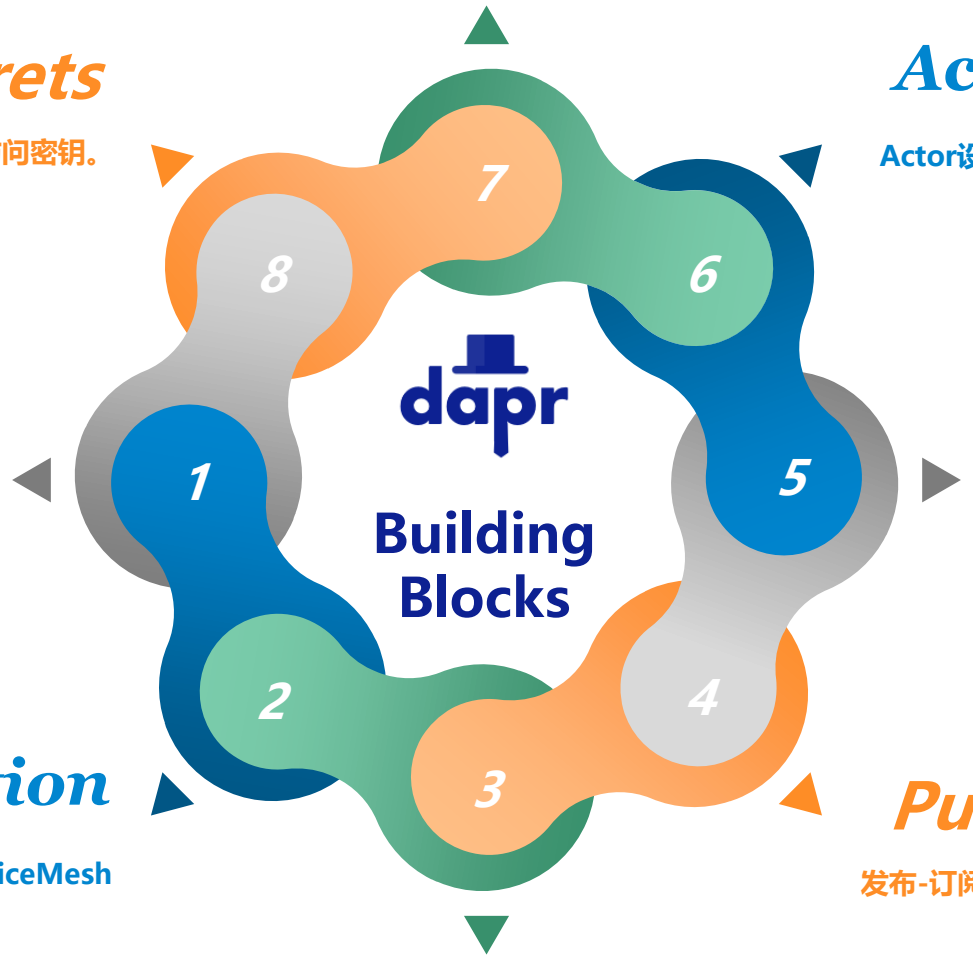
Observability

Secrets

让应用可以安全的访问密钥。

Actors

Actor设计模型，以可重用的actor对象的方式封装代码和数据



Configuration

获取应用的动态配置，进行中……

Resource Binding

资源绑定

- Input Binding: 外部输入源通过事件触发代码
- Output Binding: 绑定到外部资源如数据、消息队列等

Service Invocation

服务调用，服务间点对点通讯，类ServiceMesh

Pub-Sub

发布-订阅。服务间的消息通讯，异步，1对多

State Management

管理应用的状态，简化有状态服务的开发

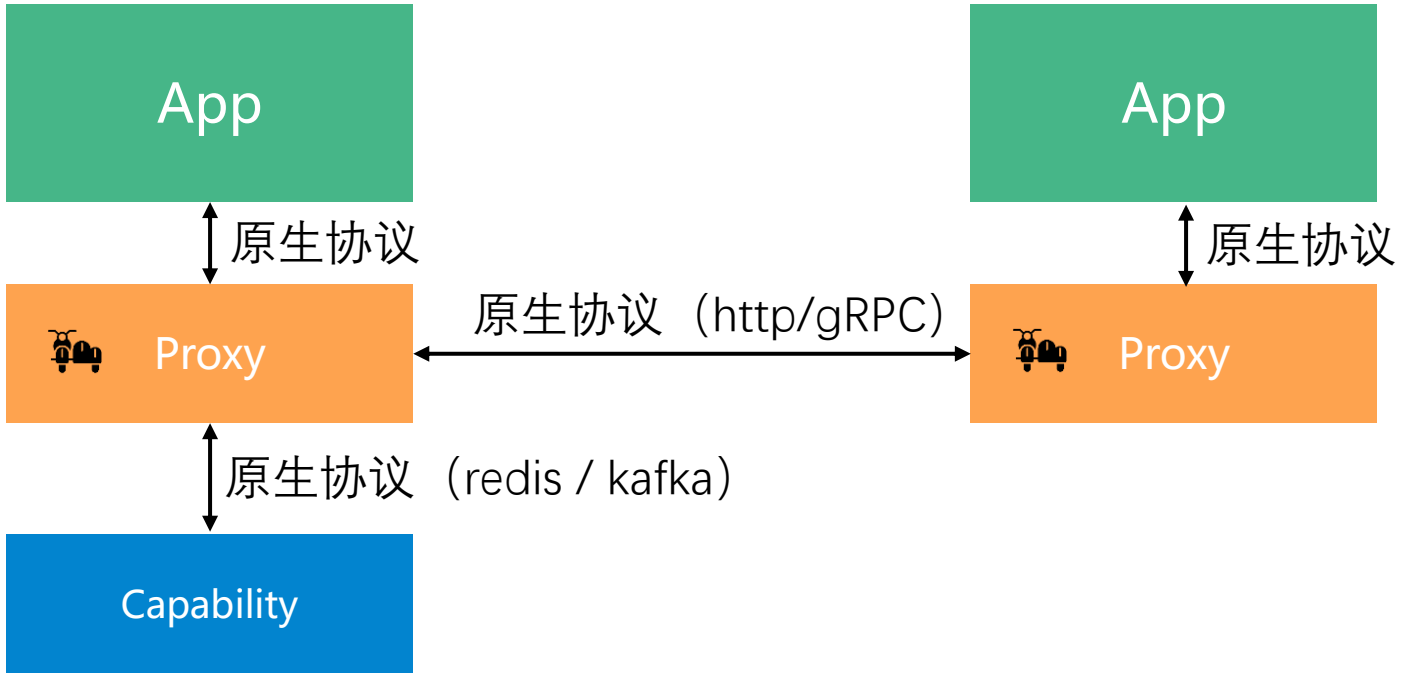
那么，问题来了：

如果 Servicemesh 也提供同样的能力，是不是就和 Dapr 一样了?

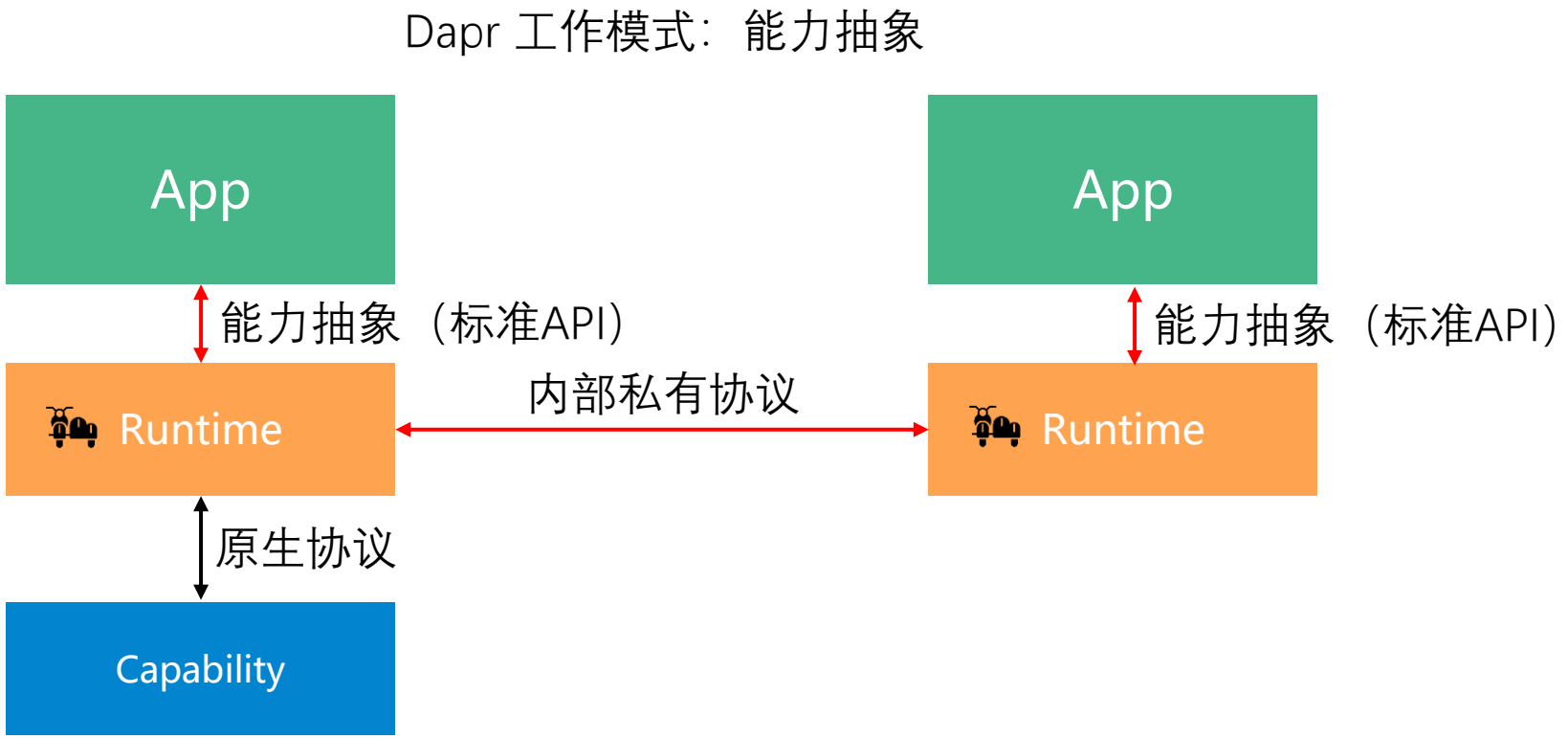
原生支持HTTP / gRPC、
原生支持Kafka、Redis 等

本质差异在于工作模式：Servicemesh是原协议转发

Servicemesh 工作模式：原协议转发

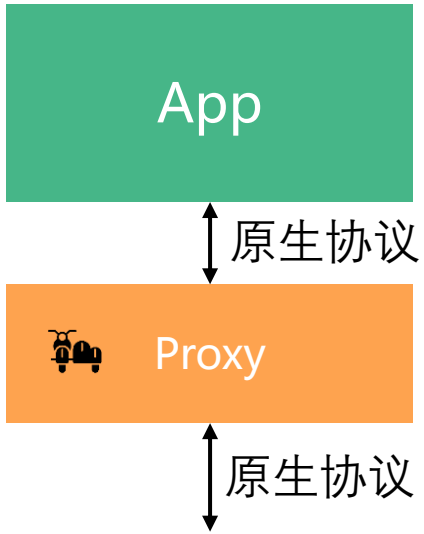


本质差异在于工作模式：Dapr是能力抽象



工作模式不同背后的设计目标

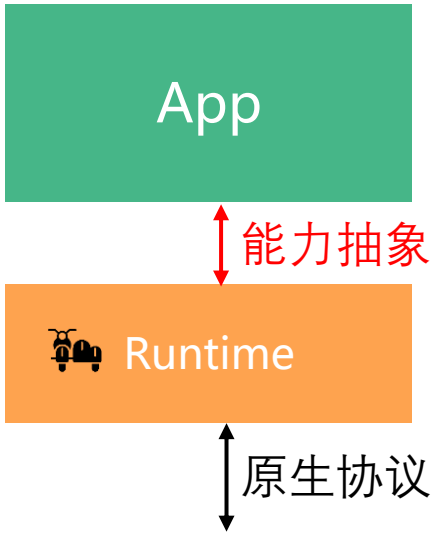
Servicemesh



工作模式：原协议转发（流量劫持）

设计目标：低侵入（甚至无侵入）

Dapr



工作模式：能力抽象（标准API）

设计目标：可移植性（跨云跨平台无厂商绑定）

Any language, any framework, anywhere

Dapr

- 标准API + 语言SDK + Runtime
- 需要应用适配（老应用需改造）

- 更适合 Green Field
- 不适合 Brown Field

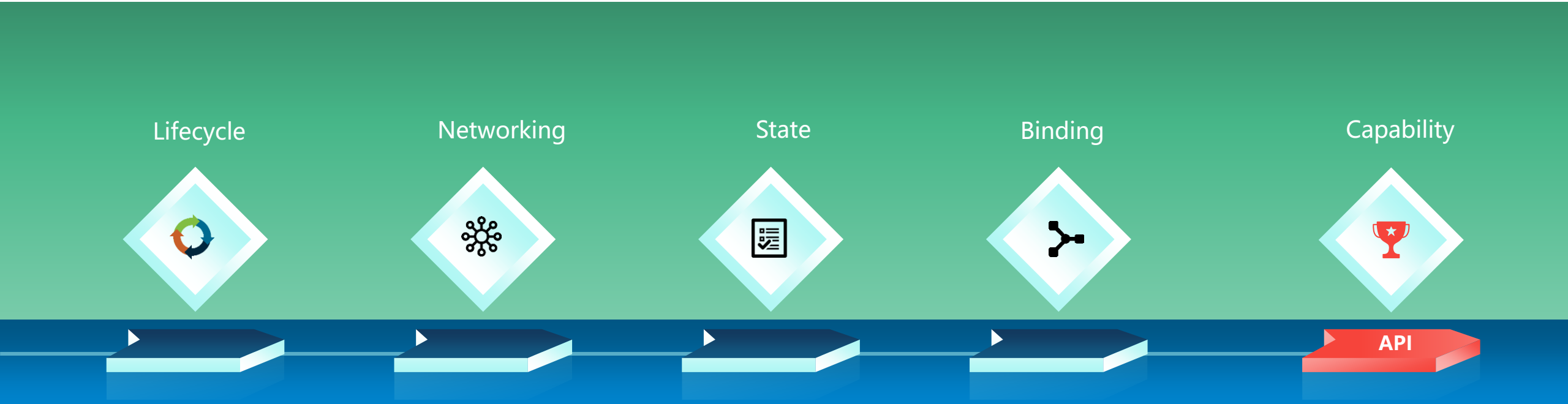
Servicemesh

- 原协议转发 + 流量劫持
- 强调对应用无侵入

- 适合 Green Field
- 更适合 Brown Field

二、死生之地：API标准化的价值

Dapr的本质：面向云原生应用的分布式能力抽象层



生命周期

- Package
- Health check
- Deployment
- Scaling
- Configuration

网络

- Service discovery
- A/B testing, canary rollouts
- Retry, timeout, circuit breaker
- Point-to-Point, pub/sub
- Security, observability

状态

- Workflow mgmt.
- Idempotency
- Temporal scheduling
- Caching
- Application state

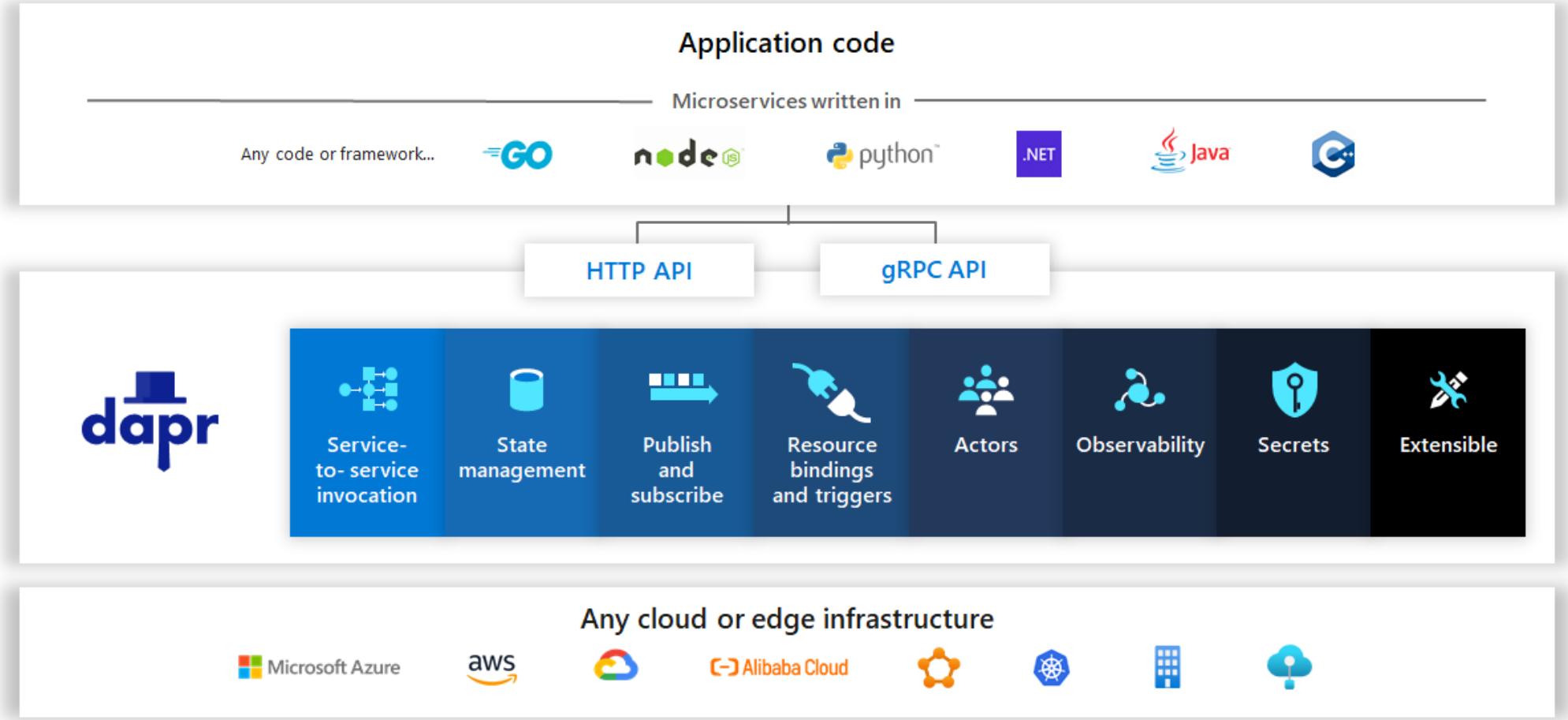
绑定

- Connectors
- Protocol conversion
- Message transformation
- Message routing
- Transnationality

抽象

- 将能力抽象为API
- 为每种能力提供多种实现
- 开发时：面对能力编程
- 运行时：通过配置选择实现

可移植性是Dapr的重要目标和核心价值

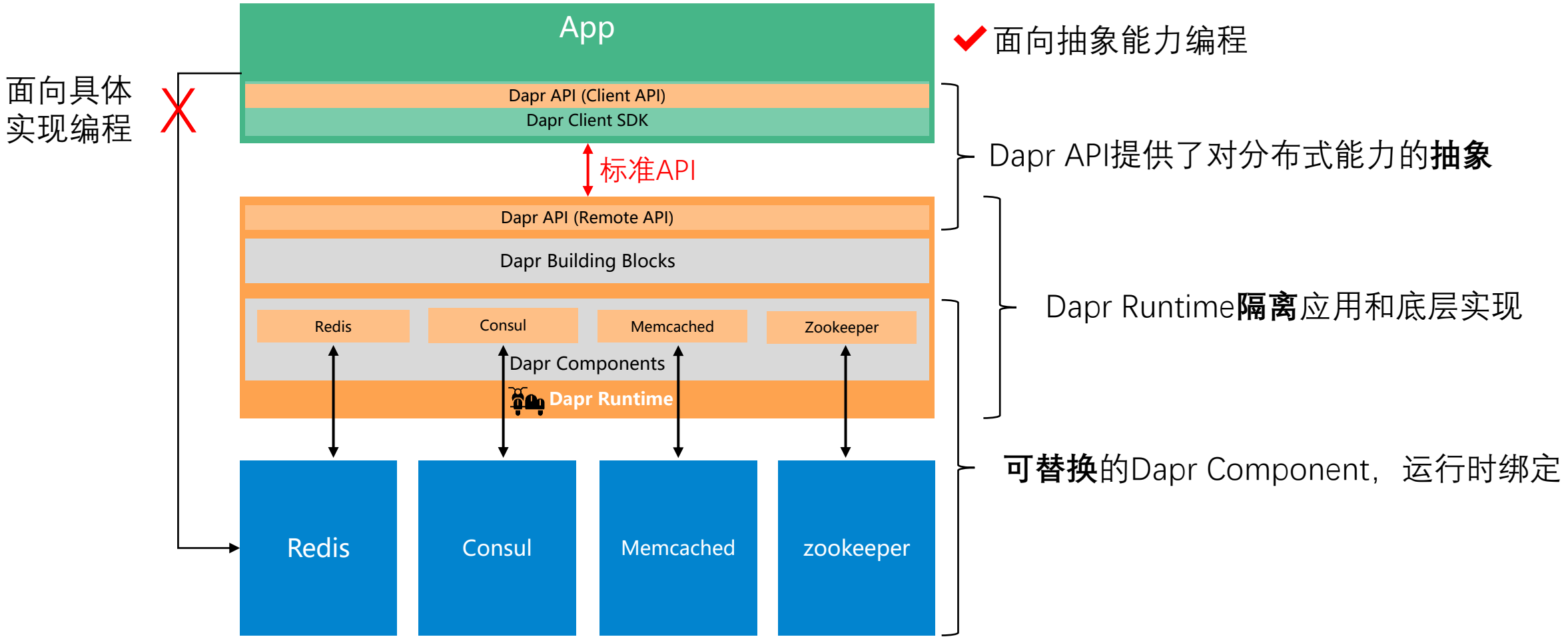


- 公有云
- 私有云
- 混合云
- 边缘网络
- 无厂商绑定

Dapr 愿景： any language, any framework, **anywhere**

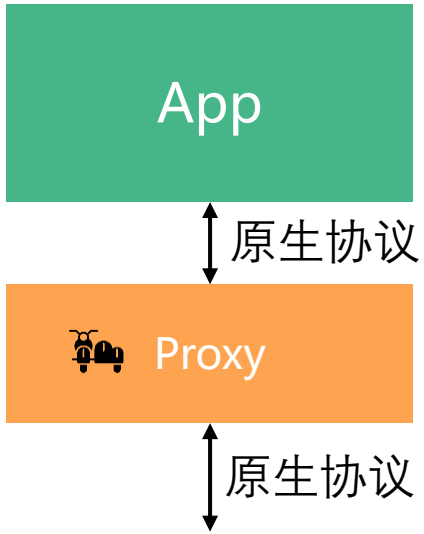
Dapr 可移植性的基石：标准API + 可拔插可替换的组件

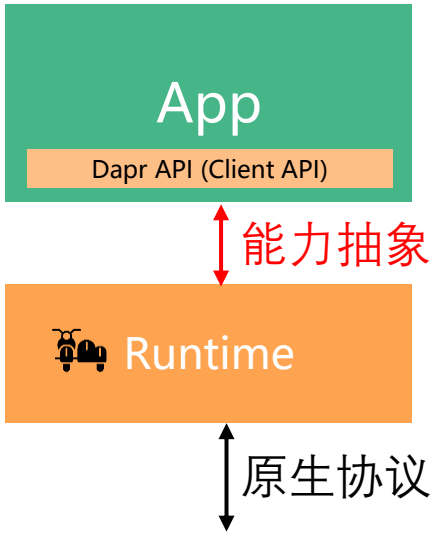
Dapr的精髓：抽象/隔离/可替换 → 解耦能力和实现 → 可移植性



Dapr落地时无可回避的问题：应用改造是有成本的

Servicemesh vs Dapr





- 应用使用Dapr API
- 新应用全新开发
- 老应用需要改造

工作模式：原协议转发（流量劫持）
设计目标：低侵入（甚至无侵入）

工作模式：能力抽象（标准API）
设计目标：可移植性（跨云跨平台无厂商绑定）

API标准化是Dapr成败的关键：建立良性循环

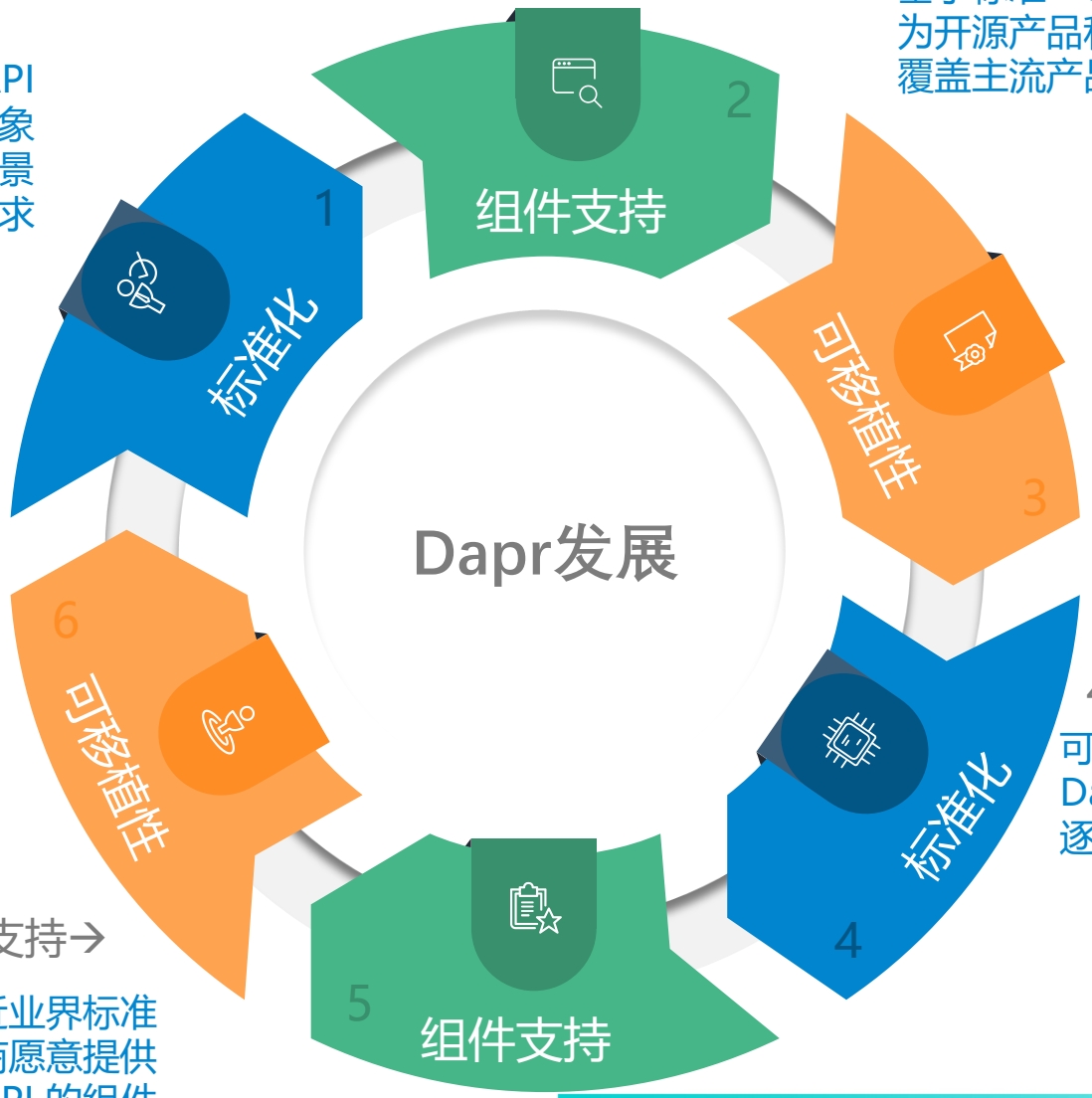
云原生社区 Cloud Native Community

1. API 标准化→

定义 Dapr API
对某一个分布式能力进行良好的抽象
覆盖日常使用的大部分场景
满足应用的常见需求

2. 提供组件支持→

基于标准 Dapr API
为开源产品和公有云商业产品提供支持组件
覆盖主流产品和厂商



3. 具备可移植性→

基于标准 Dapr API 开发的应用
可以在主流开源产品和公有云商业产品
之间自行选择适合的组件
不受平台和厂商的限制

4. API 得到更多认可→

可移植性为 Dapr 构建核心价值
Dapr API得到更多的认可
逐渐成为业界的事实标准

5. 更广泛的组件支持→

Dapr API 越接近业界标准
就会有越多的产品和厂商愿意提供
支持 Dapr API 的组件

6. 可移植性更强→

越多的组件支持
覆盖更多的开源产品和厂商
更接近 anywhere 的愿景

三、左右为难：取舍之间何去何从

（以 Dapr State API为例）

Dapr State API概述

State本质上说是key-value存储：但理论上非kv存储也可以实现State的功能，比如mysql

```
// Gets the state for a specific key.
rpc GetState(GetStateRequest) returns (GetStateResponse) {}

// Gets a bulk of state items for a list of keys
rpc GetBulkState(GetBulkStateRequest) returns (GetBulkStateResponse) {}

// Saves the state for a specific key.
rpc SaveState(SaveStateRequest) returns (google.protobuf.Empty) {}

// Deletes the state for a specific key.
rpc DeleteState(DeleteStateRequest) returns (google.protobuf.Empty) {}

// Deletes a bulk of state items for a list of keys
rpc DeleteBulkState(DeleteBulkStateRequest) returns (google.protobuf.Empty) {}

// Executes transactions for a specified store
rpc ExecuteStateTransaction(ExecuteStateTransactionRequest) returns (google.protobuf.Empty) {}
```

除了基于key的CRUD基本操作外，还有批量操作和事务操作

Dapr API 深入探讨：以 GetState()为例

高级特性：consistency / etag / expire / bulk

```proto
// Gets the state for a specific key.
rpc GetState(GetStateRequest) returns (GetStateResponse) {}
```

```proto
// GetStateRequest is the message to get key-value states from specific state store.
message GetStateRequest {
  // The name of state store.
  string store_name = 1;

  // The key of the desired state
  string key = 2;

  // The read consistency of the state store.
  common.v1.StateOptions.StateConsistency consistency = 3;

  // The metadata which will be sent to state store components.
  map<string,string> metadata = 4;
}
```

```proto
// GetStateResponse is the response conveying the state value and etag.
message GetStateResponse {
  // The byte array data
  bytes data = 1;

  // The entity tag which represents the specific version of data.
  // ETag format is defined by the corresponding data store.
  string etag = 2;

  // The metadata which will be sent to app.
  map<string,string> metadata = 3;
}
```

Metadata提供扩展性：提供实现个性化功能（而不是通用功能）的扩展途径

State API高级特性：consistency/数据一致性

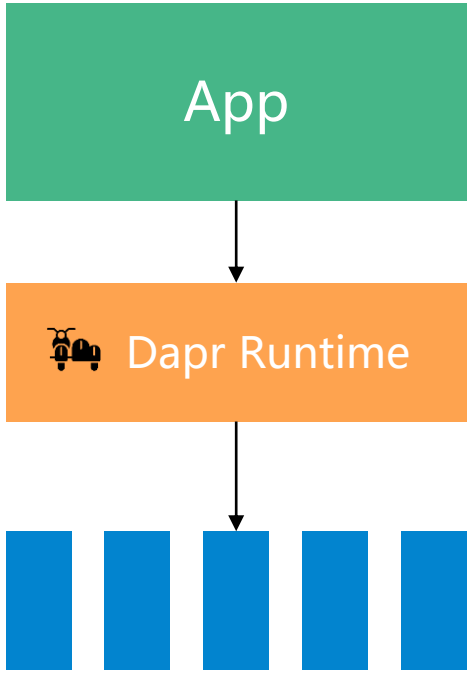
```
// Enum describing the supported consistency for state.
enum StateConsistency {
  CONSISTENCY_UNSPECIFIED = 0;
  CONSISTENCY_EVENTUAL = 1;
  CONSISTENCY_STRONG = 2;
}
```

可选参数：

- eventual: 最终一致性
- strong: 强一致性

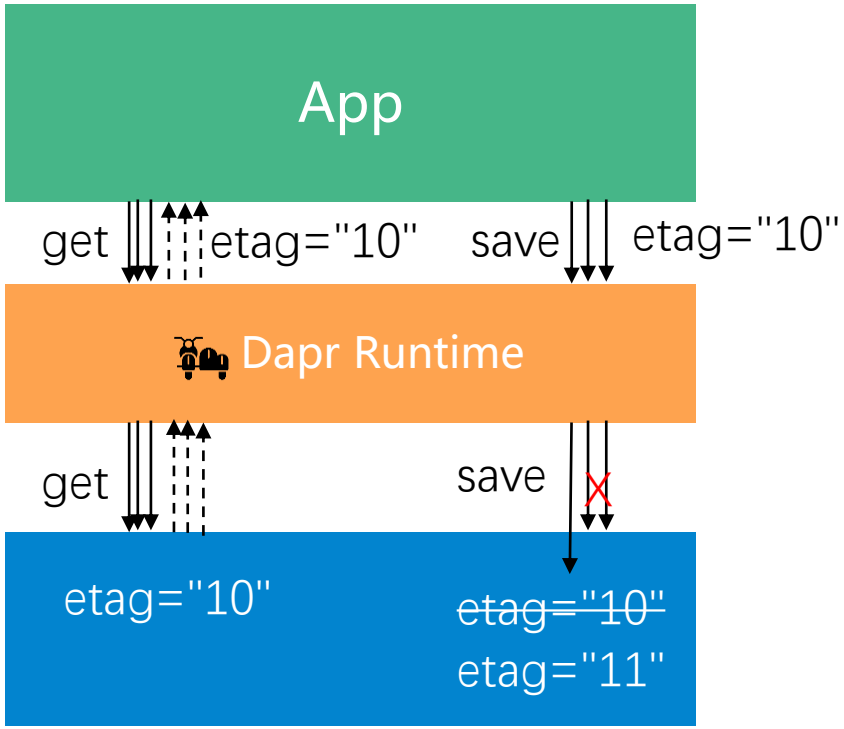
适用方法：

- Get
- Save
- Delete



State API高级特性：concurrency/并发（乐观锁）

```
// Enum describing the supported concurrency for state.
enum StateConcurrency {
  CONCURRENCY_UNSPECIFIED = 0;
  CONCURRENCY_FIRST_WRITE = 1;
  CONCURRENCY_LAST_WRITE = 2;
}
```



可选参数：

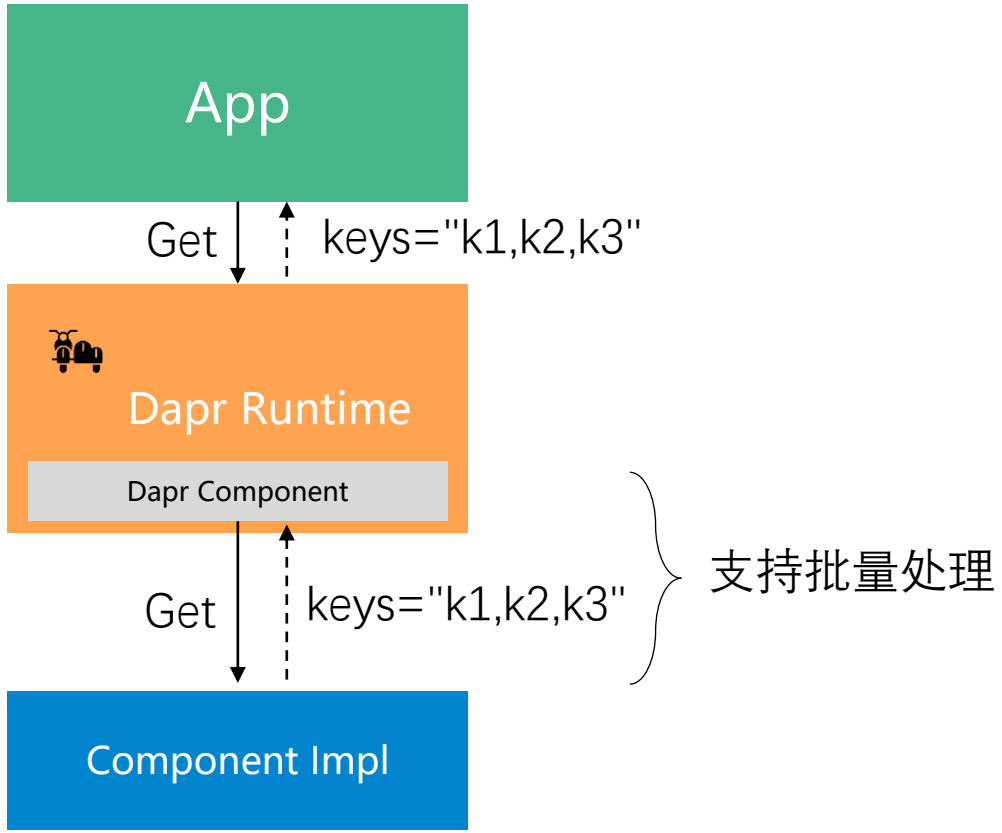
- first_write: 乐观锁
- last_write: 简单覆盖

适用方法：

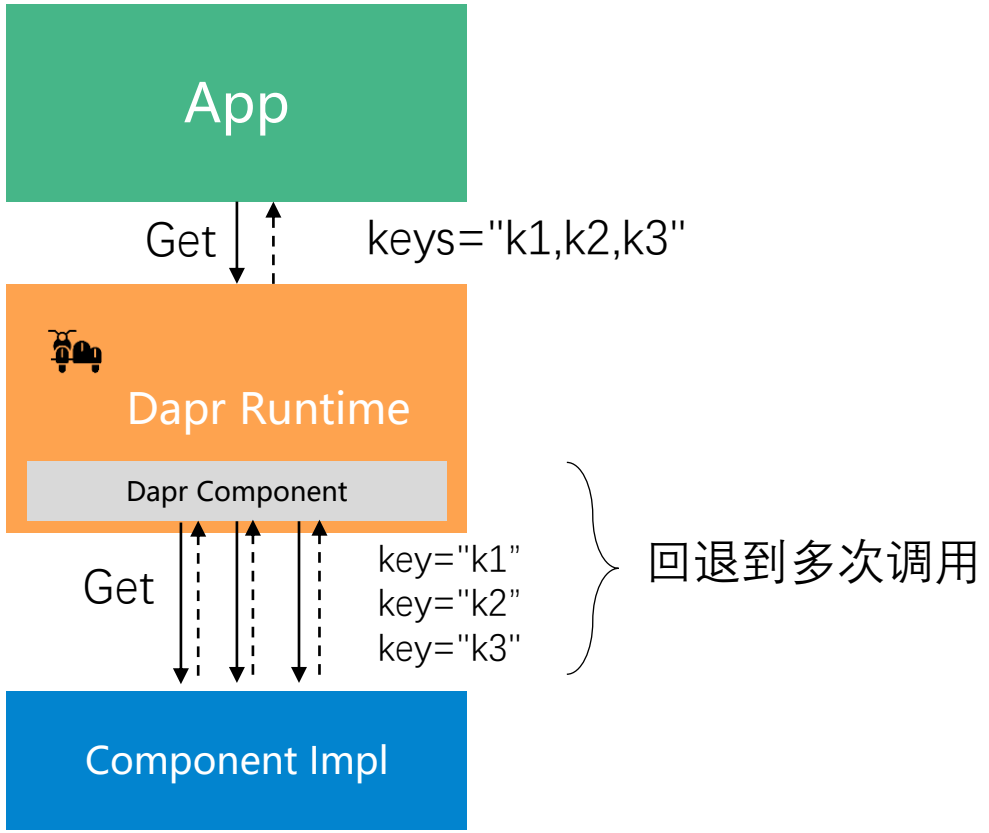
- Save

State API高级特性：批量操作

Component 原生支持批量



Component 原生不支持批量



GetBulkState() 的实现逻辑：优化 + 兜底

云原生社区 Cloud Native Community

```go
func (a *api) GetBulkState(ctx context.Context, in *runtimev1pb.GetBulkStateRequest) (*runtimev1pb.GetBulkStateResponse, error) {

    bulkGet, responses, err := store.BulkGet(reqs)
    // if store supports bulk get
    if bulkGet {
        return bulkResp, nil
    }

    // if store doesn't support bulk get, fallback to call get() method one by one
    limiter := concurrency.NewLimiter(int(in.Parallelism))
    for i := 0; i < len(reqs); i++ {
        fn := func(param interface{}) {
            req := param.(*state.GetRequest)
            r, err := store.Get(req)
            item := &runtimev1pb.BulkStateItem{
                Key: state_loader.GetOriginalStateKey(req.Key),
            }
            bulkResp.Items = append(bulkResp.Items, item)
        }
        limiter.Execute(fn, &reqs[i])
    }

    return bulkResp, nil
}
```

State API实现中最大的挑战：很多组件无法支持事务

支持事务的component有：

- Cosmosdb
- Mongodb
- Mysql
- Postgresql
- Redis
- Rethinkdb
- Sqlserver

可用于secret

不支持事务的component有：

- Aerospike
- Aws/dynamodb
- Azure/blobstorage
- Azure/tablestorage
- Cassandra
- Cloudstate
- Couchbase
- Gcp/firestore
- Hashicorp/consul
- hazelcase
- memcached
- zookeeper

State Components被分成两类：是否支持事务

云原生社区 Cloud Native Community

```go
// NewRedisStateStore returns a new redis state store
func NewRedisStateStore(logger logger.Logger) *StateStore {
    s := &StateStore{
        features: []state.Feature{state.FeatureETag, state.FeatureTransactional},
    }
    return s
}
```

1. Component在初始化时
指明是否支持事务

2. Daprd启动时过滤
支持事务的component

```go
transactionalStateStores := map[string]state.TransactionalStore{}
for key, store := range stateStores {
    if state.FeatureTransactional.IsPresent(store.Features()) {
        transactionalStateStores[key] = store.(state.TransactionalStore)
    }
}
```

3. Daprd在收到事务请求时
检查component是否支持事务

```go
transactionalStore, ok := a.transactionalStateStores[storeName]
if !ok {
    err := status.Errorf(codes.Unimplemented, messages.ErrStateStoreNotSupported, storeName)
    apiServerLogger.Debug(err)
    return &emptypb.Empty{}, err
}
```

Dapr API的抽象和实现：理想很美好，现实很残酷

基本操作

- 简单的KV语义
- CURD

批量

- Bulk Get
- Bulk Set
- Bulk Delete

过期

- 有效存活时间
- TtlInSecends

并发

- 乐观锁
- ETag

数据一致性

- 强一致性
- 最终一致性

事务

- 原子操作

越是高级特性，越难于让所有组件都支持

痛苦的抉择：向左？还是向右？

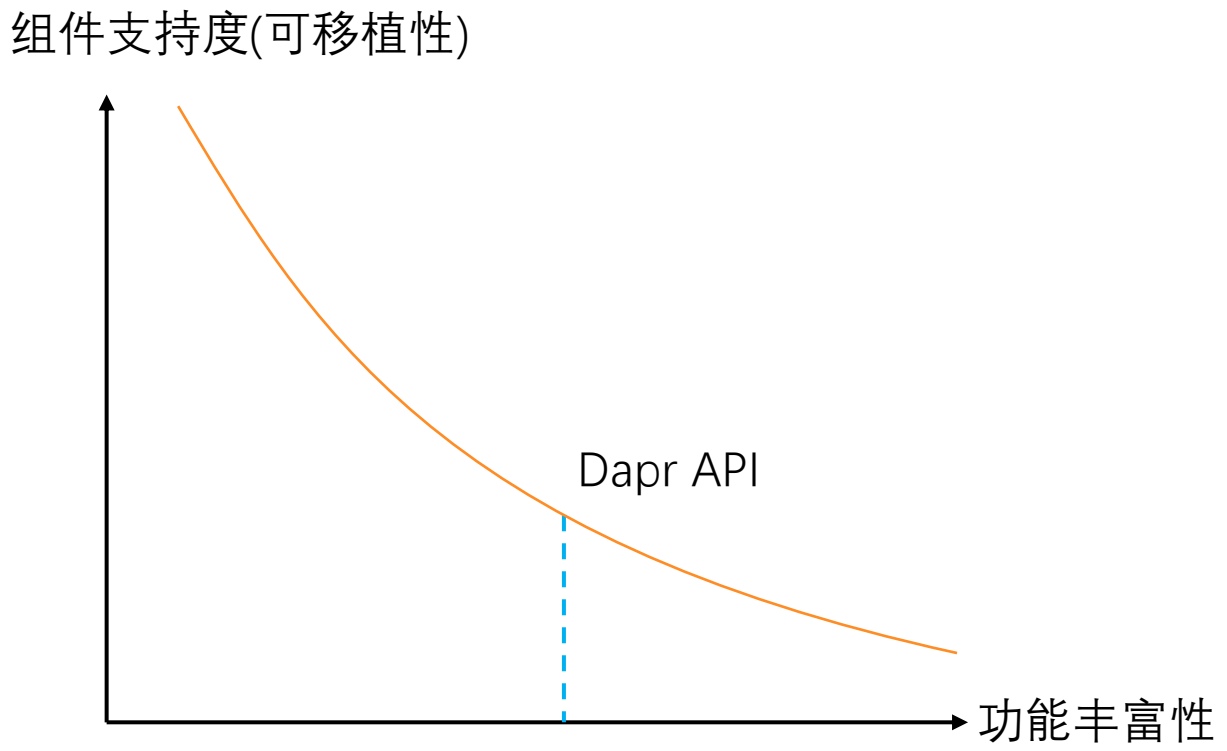
API定义的核心挑战：功能丰富性和组件可移植性难于兼顾

Dapr

- 每个Dapr构建块的 API 在初始创建时，通常会从基本功能开始，相对偏左侧
- 随着时间的推移，为了满足更多场景下的需求，会向右移动，在API中增加新功能
- 新增的功能可能会导致部分组件无法提供支持，损害可移植性

Dapr API定义时需要权衡和取舍：

- 不能过于保守：太靠近左侧，虽然可移植性得以体现，但功能的缺失会影响使用
- 不能过于激进：太靠近右侧，虽然功能非常齐备，但是组件的支持度会变差，影响可移植性



落地实践：引入请求级别的 metadata 来进行自定义扩展

云原生社区 Cloud Native Community

```proto
// GetStateRequest is the message to get key-value states from specific state store.
message GetStateRequest {
  // The name of state store.
  string store_name = 1;

  // The key of the desired state
  string key = 2;

  // The read consistency of the state store.
  common.v1.StateOptions.StateConsistency consistency = 3;

  // The metadata which will be sent to state store components.
  map<string,string> metadata = 4;
}
```

```proto
// GetStateResponse is the response conveying the state value and etag.
message GetStateResponse {
  // The byte array data
  bytes data = 1;

  // The entity tag which represents the specific version of data.
  // ETag format is defined by the corresponding data store.
  string etag = 2;

  // The metadata which will be sent to app.
  map<string,string> metadata = 3;
}
```

在不改变API定义的情况下，通过请求级别的metadata来提供自定义的功能扩展，以便使用更多的底层能力。

实践：通过Metadata实现扩展的实际案例

云原生社区 Cloud Native Community

```go
//tair
const (
    // metadata key constants
    MetadataTairKey          = "key"
    MetadataTairEtag         = "etag"
    MetadataTairTtlInSeconds = "ttlInSeconds"
    MetadataTairUsername     = "tair-username"
    MetadataTairUnit         = "tair-unit"
    MetadataTairPrefix       = "tair-prefix"
    MetadataTairOperation    = "operation"
    MetadataTairIncrease     = "increase"
    MetadataTairDecrease     = "decrease"
    MetadataTairInvalid      = "invalid"
    MetadataTairDefaultValue = "default-value"
    MetadataTairMinValue     = "min-value"
    MetadataTairMaxValue     = "max-value"
    MetadataTairKeyType      = "key-type"
    MetadataTairValueType    = "value-type"
    MetadataTairPrefixType   = "prefix-key-type"

    // data type constants for key / prefix / value
    DataTypeByteArray = "byte[]"
    DataTypeString    = "string"
    DataTypeLong      = "long"
    DataTypeInt       = "int"
    DataTypeByte      = "byte"
    DataTypeBoolean   = "boolean"
    DataTypeTimestamp = "timestamp"
    DataTypeFloat     = "float"
    DataTypeDouble    = "double"
    DataTypeObject    = "object"
)
```

```go
// rpc
const (
    MetadataRpcGroup             = "rpc-group"
    MetadataRpcVersion           = "rpc-version"
    MetadataRpcInterface         = "rpc-interface-name"
    MetadataRpcMethodName        = "rpc-method-name"
    MetadataRpcMethodParamTypes  = "rpc-method-parameter-types"
    MetadataRpcPassThrough       = "rpc-pass-through"
    MetadataRpcGeneric           = "rpc-generic"
    MetadataRpcSerializationType = "rpc-serialization-type"
    MetadataRpcTimeout           = "rpc-timeout"
    MetadataRpcAction            = "rpc-service-action"
    MetadataRpcTargetIp          = "rpc-target-ip"
    MetadataRpcTargetUnit        = "rpc-target-unit"
    MetadataRpcUserRouterID      = "rpc-user-router-id"
    MetadataRpcAttachmentPrefix  = "rpc-attachment-"
    //MetadataHsfTransformBizError = "hsf-return-error-as-result"
)
```

```go
//rocketmq
const (
    MetadataRocketmqTag           = "rocketmq-tag"
    MetadataRocketmqKey           = "rocketmq-key"
    MetadataRocketmqConsumerGroup = "rocketmq-consumerGroup"
    MetadataRocketmqType          = "rocketmq-sub-type"
    MetadataRocketmqExpression    = "rocketmq-sub-expression"
    MetadataRocketmqBrokerName    = "rocketmq-broker-name"
)

//diamond
const (
    MetadataConfigDataId   = "config-id"
    MetadataConfigGroup    = "config-group"
    MetadataConfigTimeout  = "config-timeout"
    MetadataConfigOnchange = "config-onchange"
    MetadataConfigWatch    = "config-watch"
)
```

注意：expire 的功能在 state API 中是通过名为 ttlInSeconds 的metadata 实现。

实践：通过Metadata实现扩展的实际案例

云原生社区 Cloud Native Community

```go
//tair
const (
    // metadata key constants
    MetadataTairKey          = "key"
    MetadataTairEtag         = "etag"
    MetadataTairTtlInSeconds = "ttlInSeconds"
    MetadataTairUsername     = "tair-username"
    MetadataTairUnit         = "tair-unit"
    MetadataTairPrefix       = "tair-prefix"
    MetadataTairOperation    = "operation"
    MetadataTairIncrease     = "increase"
    MetadataTairDecrease     = "decrease"
    MetadataTairInvalid      = "invalid"
    MetadataTairDefaultValue = "default-value"
    MetadataTairMinValue     = "min-value"
    MetadataTairMaxValue     = "max-value"
    MetadataTairKeyType      = "key-type"
    MetadataTairValueType    = "value-type"
    MetadataTairPrefixType   = "prefix-key-type"

    // data type constants for key / prefix / value
    DataTypeByteArray = "byte[]"
    DataTypeString    = "string"
    DataTypeLong      = "long"
    DataTypeInt       = "int"
    DataTypeByte      = "byte"
    DataTypeBoolean   = "boolean"
    DataTypeTimestamp = "timestamp"
    DataTypeFloat     = "float"
    DataTypeDouble    = "double"
    DataTypeObject    = "object"
)
```

```go
// rpc
const (
    MetadataRpcGroup             = "rpc-group"
    MetadataRpcVersion           = "rpc-version"
    MetadataRpcInterface         = "rpc-interface-name"
    MetadataRpcMethodName        = "rpc-method-name"
    MetadataRpcMethodParamTypes  = "rpc-method-parameter-types"
    MetadataRpcPassThrough       = "rpc-pass-through"
    MetadataRpcGeneric           = "rpc-generic"
    MetadataRpcSerializationType = "rpc-serialization-type"
    MetadataRpcTimeout           = "rpc-timeout"
    MetadataRpcAction            = "rpc-service-action"
    MetadataRpcTargetIp          = "rpc-target-ip"
    MetadataRpcTargetUnit        = "rpc-target-unit"
    MetadataRpcUserRouterID      = "rpc-user-router-id"
    MetadataRpcAttachmentPrefix  = "rpc-attachment-"
    //MetadataHsfTransformBizError = "hsf-return-error-as-result"
)
```

```go
//rocketmq
const (
    MetadataRocketmqTag           = "rocketmq-tag"
    MetadataRocketmqKey           = "rocketmq-key"
    MetadataRocketmqConsumerGroup = "rocketmq-consumerGroup"
    MetadataRocketmqType          = "rocketmq-sub-type"
    MetadataRocketmqExpression    = "rocketmq-sub-expression"
    MetadataRocketmqBrokerName    = "rocketmq-broker-name"
)

//diamond
const (
    MetadataConfigDataId   = "config-id"
    MetadataConfigGroup    = "config-group"
    MetadataConfigTimeout  = "config-timeout"
    MetadataConfigOnchange = "config-onchange"
    MetadataConfigWatch    = "config-watch"
)
```

注意：expire 的功能在 state API 中是通过名为 ttlInSeconds 的metadata 实现。

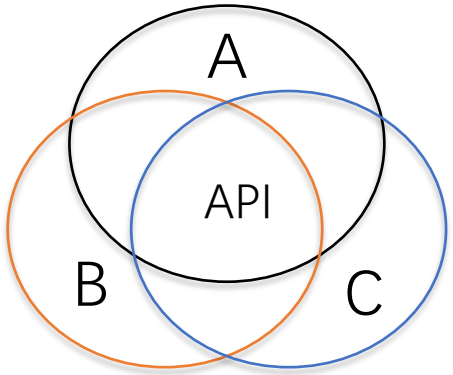
请求级别Metadata是柄双刃剑：满足需求，破坏可移植性

1. 可移植性是Dapr的核心价值

→ 定义Dapr API时应尽量满足可移植性的诉求

2. API设计时会偏功能最小集

→ 为了提供最大限度的可移植性，设计时往往会倾向于从功能最小集出发



3. 出现功能缺失→

功能最小集合意味着Dapr API只定义基本功能，自然会导致缺乏各种高级特性，落地时会遇到无法满足应用需求的情况

4. 进行自定义扩展

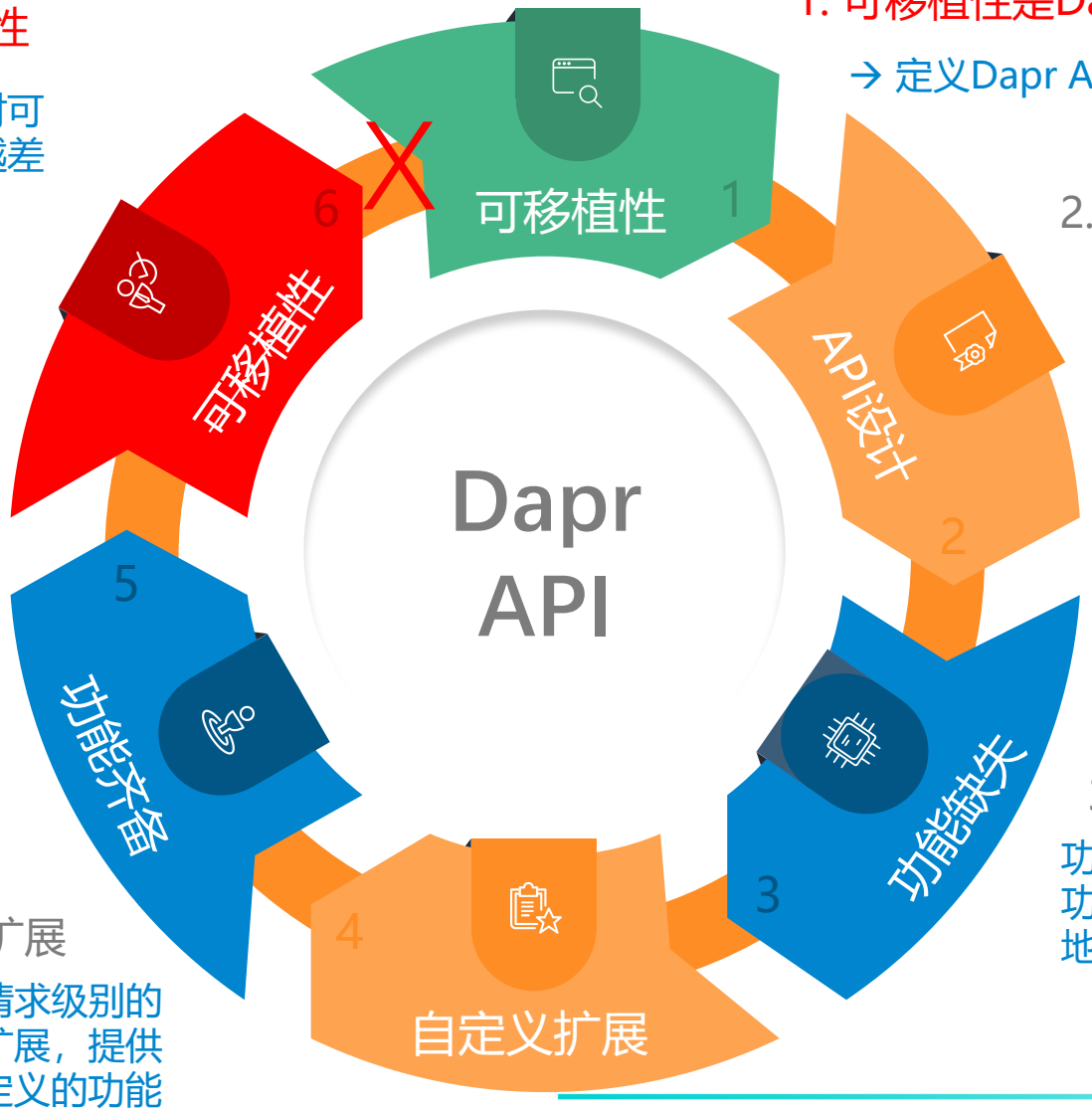
为了满足需求，使用请求级别的metadata 进行自定义扩展，提供Dapr API 没有定义的功能

5. 优点：满足功能需求

底层组件的能力得以释放

6. 缺点：严重破坏可移植性

自定义扩展越多，在迁移到其他组件时可能丢失的功能就越多，可移植性就越差



Dapr在阿里内部落地时遭遇的重大挑战

背景：来自业务团队的普遍需求——之前有的功能现在都要有

现状1：十余年打磨下来，阿里内部中间件各种五花八门的功能都有

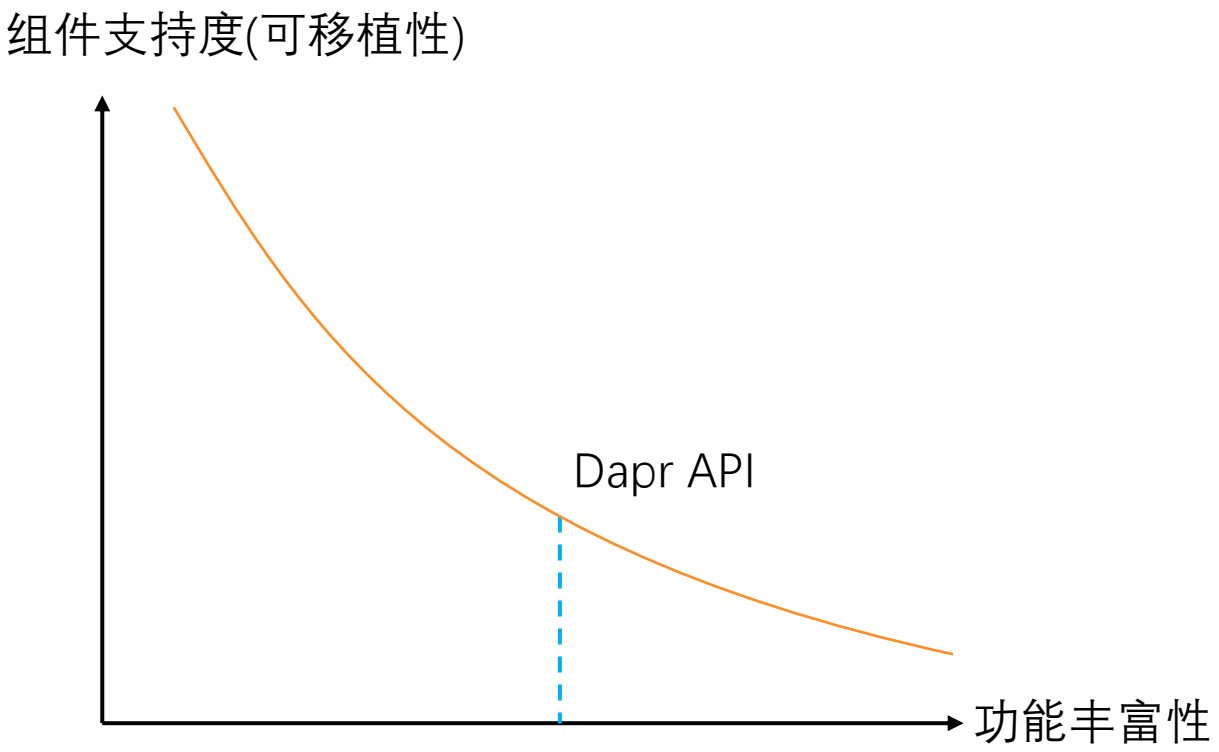
现状2：社区开源版本/其他云平台提供的产品往往没有这些功能

现状3：Dapr在内部落地时功能方面的GAP非常大

结果： 引入request级别metadata——短期满足了功能需求，长期损害可移植性

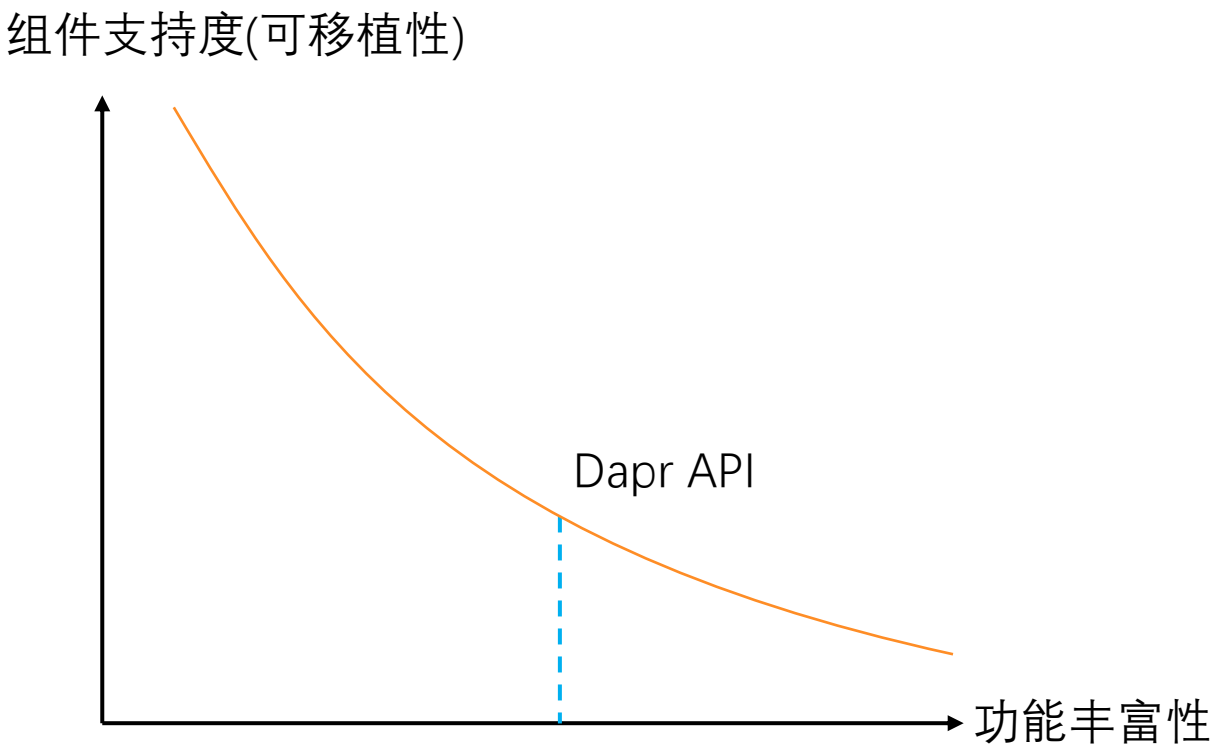
必须正视并找到解决之道

反思：左右为难，何去何从？

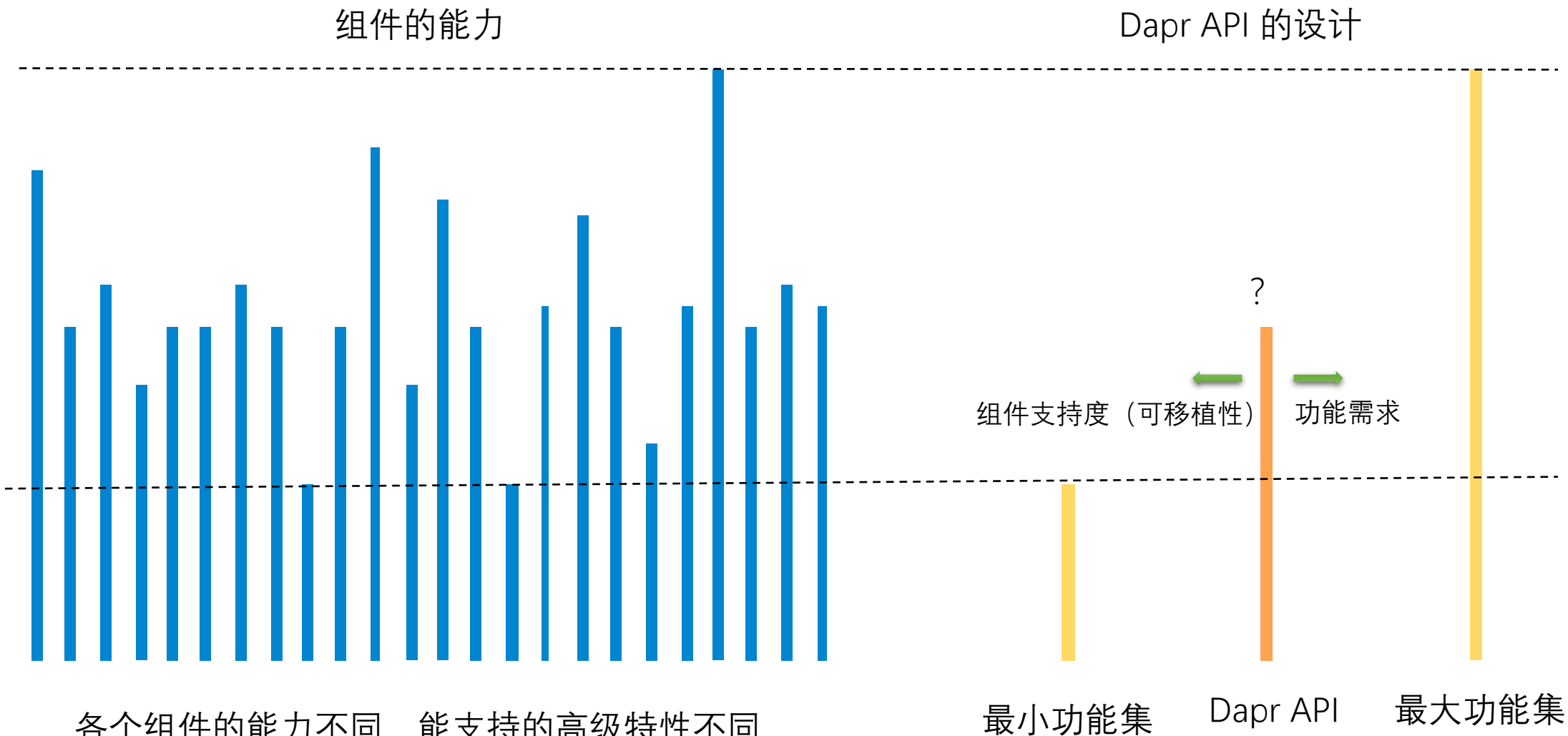


四、实践为先：在落地中探索打磨

空想无益，实践为先：以state API为例分析dapr实践



换一个角度看问题：核心在于组件提供的能力不是平齐的



解决思路一：Dapr Runtime 弥补组件缺失能力

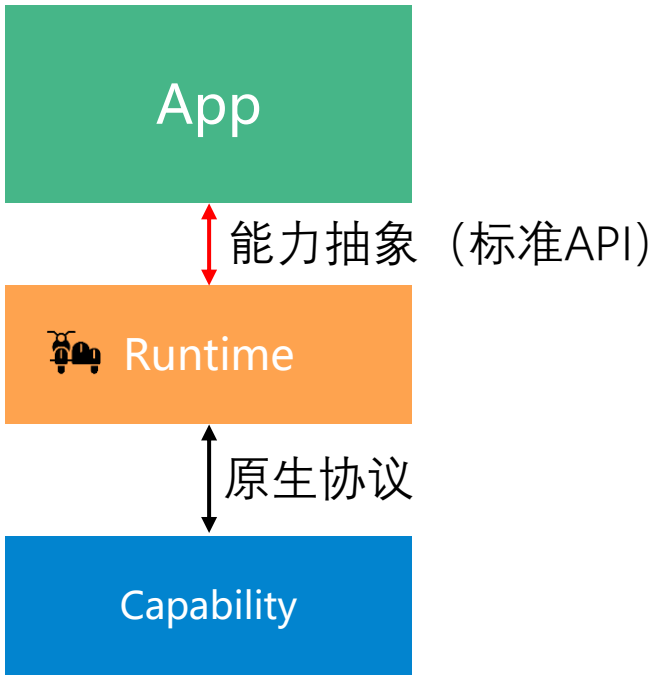
云原生社区 Cloud Native Community

```go
func (a *api) GetBulkState(ctx context.Context, in *runtimev1pb.GetBulkStateRequest) (*runtimev1pb.GetBulkStateResponse, error) {

    bulkGet, responses, err := store.BulkGet(reqs)
    // if store supports bulk get
    if bulkGet {
        return bulkResp, nil
    }

    // if store doesn't support bulk get, fallback to call get() method one by one
    limiter := concurrency.NewLimiter(int(in.Parallelism))
    for i := 0; i < len(reqs); i++ {
        fn := func(param interface{}) {
            req := param.(*state.GetRequest)
            r, err := store.Get(req)
            item := &runtimev1pb.BulkStateItem{
                Key: state_loader.GetOriginalStateKey(req.Key),
            }
            bulkResp.Items = append(bulkResp.Items, item)
        }
        limiter.Execute(fn, &reqs[i])
    }

    return bulkResp, nil
}
```

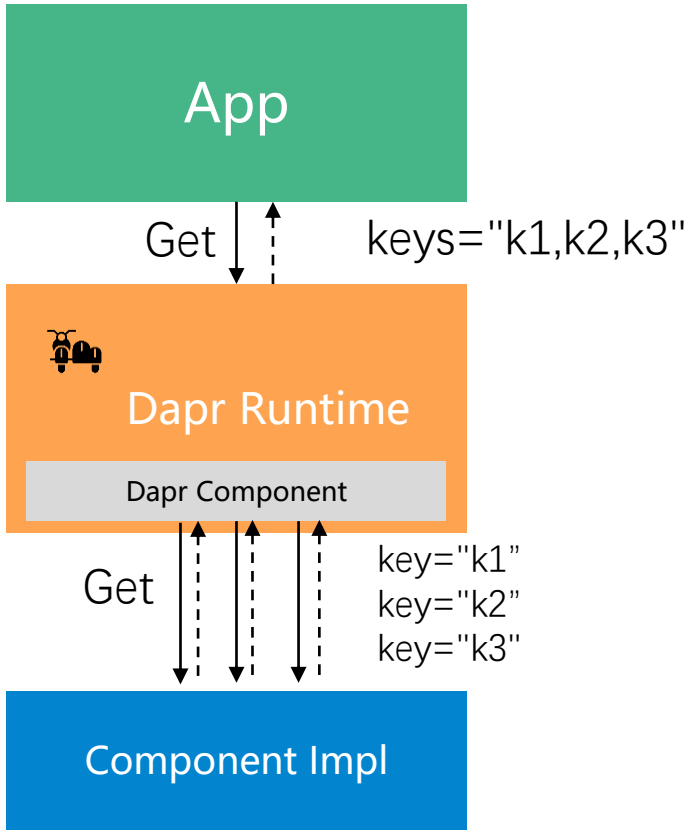
以批量操作为例

Dapr的工作模式使得Dapr有机会对组件能力进行补充

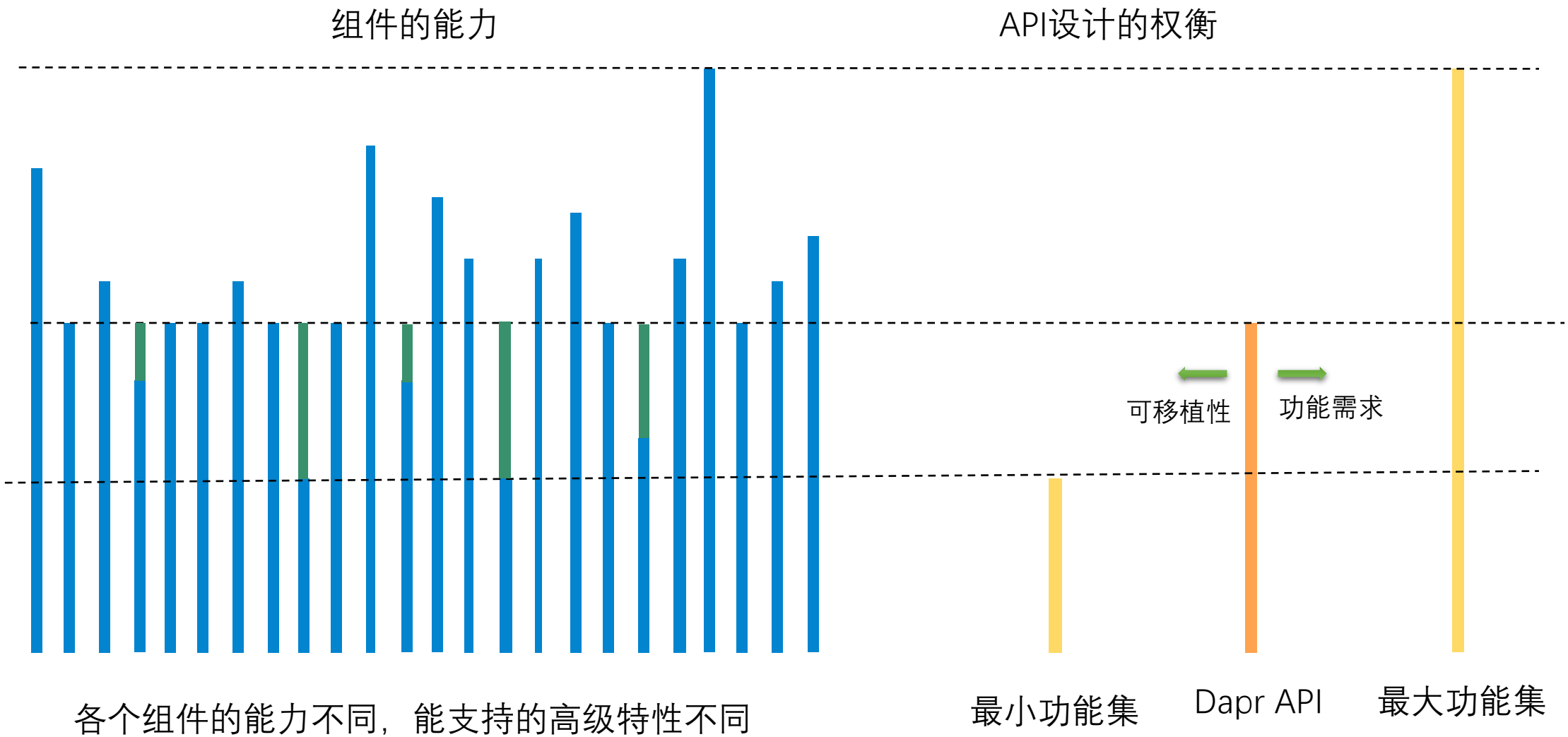
Dapr 工作模式



Dapr 干预



比较理想的解决方式：优先采纳，代价是Dapr需要做更多的工作

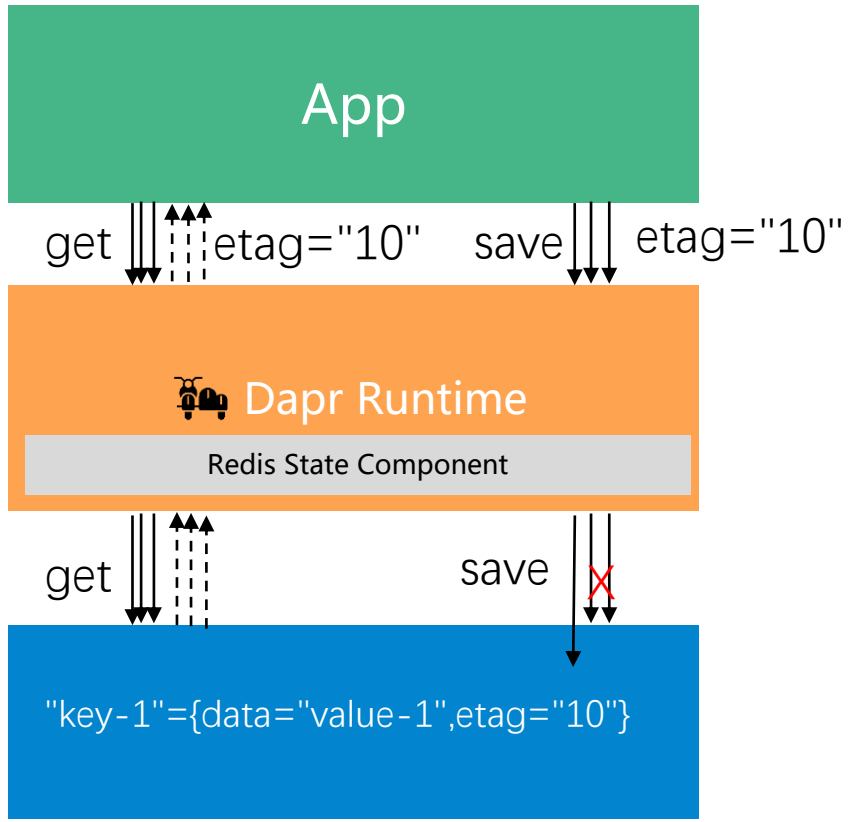


解决思路二：Dapr Component 弥补组件缺失能力

- Redis 原生不支持etag
- redis state component在实现中采用了hashmap，通过 data 和 version 两个hashmap key来实现etag
- 为了保证对 data 和 version 操作的原子性，引入了LUA脚本

```
func (r *StateStore) deleteValue(req *state.DeleteRequest) error {
    _, err := r.client.Do(r.ctx, args...: "EVAL", delQuery, 1, req.Key, *req.ETag).Result()
    return nil
}

func (r *StateStore) setValue(req *state.SetRequest) error {
    _, err = r.client.Do(r.ctx, args...: "EVAL", setQuery, 1, req.Key, ver, bt).Result()
    return nil
}
```

以并发支持为例

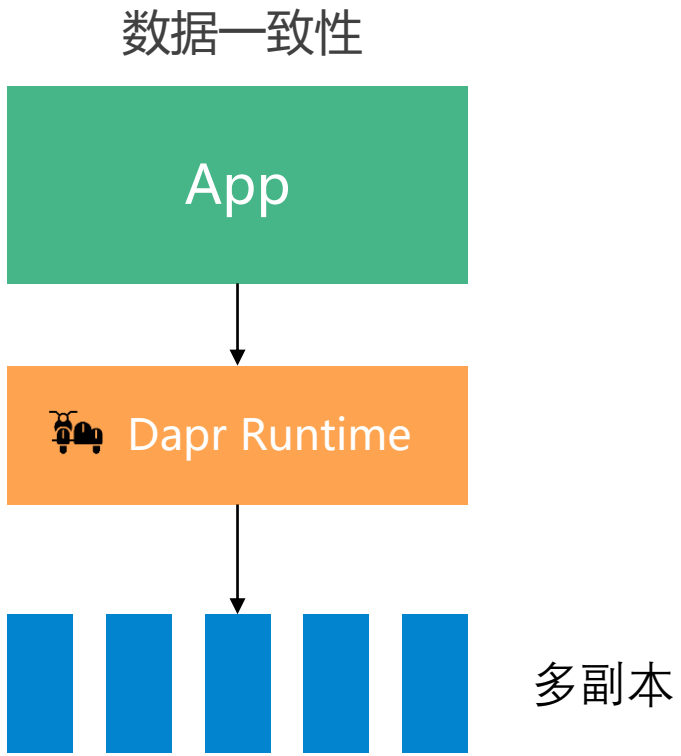


解决思路三：Dapr无法弥补，但可以模糊处理

支持强一致性的redis state实现代码：

```
if req.Options.Consistency == state.Strong && r.replicas > 0 {
    _, err = r.client.Do(r.ctx, args...: "WAIT", r.replicas, 1000).Result()
    if err != nil {
        return fmt.Errorf( format: "redis waiting for %v replicas to acknowle
    }
}
```

如果不支持强一致性，或者没有配置集群，
则可以选择忽略 consistency 参数
不实现，也不报错



解决思路四：无法弥补又不能模糊处理，认怂，让用户选

- 刚需：从需求上说，功能必须有
- 硬伤：从实现上说，很多组件不支持事务
- 弥补：dapr目前无力弥补
- 模糊：不能模糊处理，必须明确支持或者不支持

支持事务的：

- Cosmosdb
- Mongodb
- Mysql
- Postgresql
- Redis
- Rethinkdb
- Sqlserver

不支持事务的：

- Aerospike
- Aws/dynamodb
- Azure/blobstorage
- Azure/tablestorage
- Cassandra
- Cloudstate
- Couchbase
- Gcp/firestore
- Hashicorp/consul
- hazelcase
- memcached
- zookeeper

Dapr 目前的做法：

- 按照是否支持事务来区分 state components
- 用户如果需要事务支持，必须选择支持事务的组件
- 需要事务支持时，可移植性的范围被限制为支持事务的组件列表

该方案的缺陷：

- 会对可移植性造成灾难性后果
- 必须严格限制使用：只能为个别关键特性开特例，不能滥用

更大挑战：比 State API 复杂的多的 Configuration API

感兴趣的同学请浏览：

https://github.com/dapr/dapr/issues/2988

五、路阻且长：但行好事莫问前程

Dapr有美好的前景，但必然有艰难险阻

- Dapr 符合云原生的大方向，能为云原生应用带来巨大的价值
- Dapr 走在社区的最前面，我们是开拓者，我们在创造历史

- Dapr API 是 Dapr 成败的关键之一
- 云原生需要一个通用的分布式能力API
- Dapr API 需要在不断实践中补充和完善
- 这个过程注定很艰难

但行好事
莫问前程

--- 致正在为梦想奔波的人们

云原生社区
Cloud Native Community

+

感谢您的观看
THANKS!

官网：https://cloudnative.to

Twitter：CloudNativeCN

扫 码 关 注