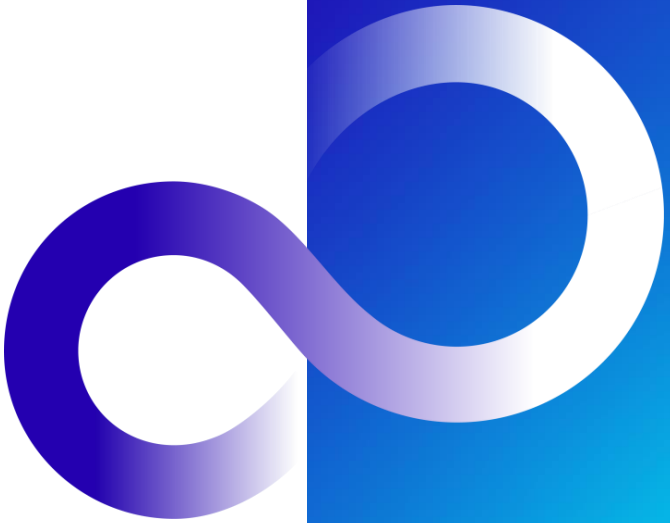
Kubernetes Meetup Tokyo #50

FUJITSU

K8s Controller Manager & KubeBuilder Deep Dive



2022/4/27

富士通株式会社

石井伴旺

本発表の目的

FUJITSU

- K8sオブジェクトの作成/更新/削除からK8sコントローラがReconcile処理を実行するまでの間にkube-controller-manager内部でどのような処理が行われているのかを説明します
  - 本発表を通してkube-controller-managerの処理がイメージしやすくなります
- ~~kube-controller-managerとkubebuilderで作成したコントローラとの間の差異について説明する~~　kube-controller-managerの分量が多くなってしまったため、割愛させて頂きます

- 前提知識
  - K8sのコンポーネント・K8sリソースの役割といった、K8sの知識がある程度ある方を対象としています
  - Goをある程度読める方を想定していますが、図を使っての説明が有るのでGoに不慣れな方でもkube-controller-managerの内部処理の感触がつかめると思います
  - K8s v1.23のコードをベースにしています

※スライド内に記載のソースコードのパスはK8sレポジトリrootからのrelative pathです

自己紹介

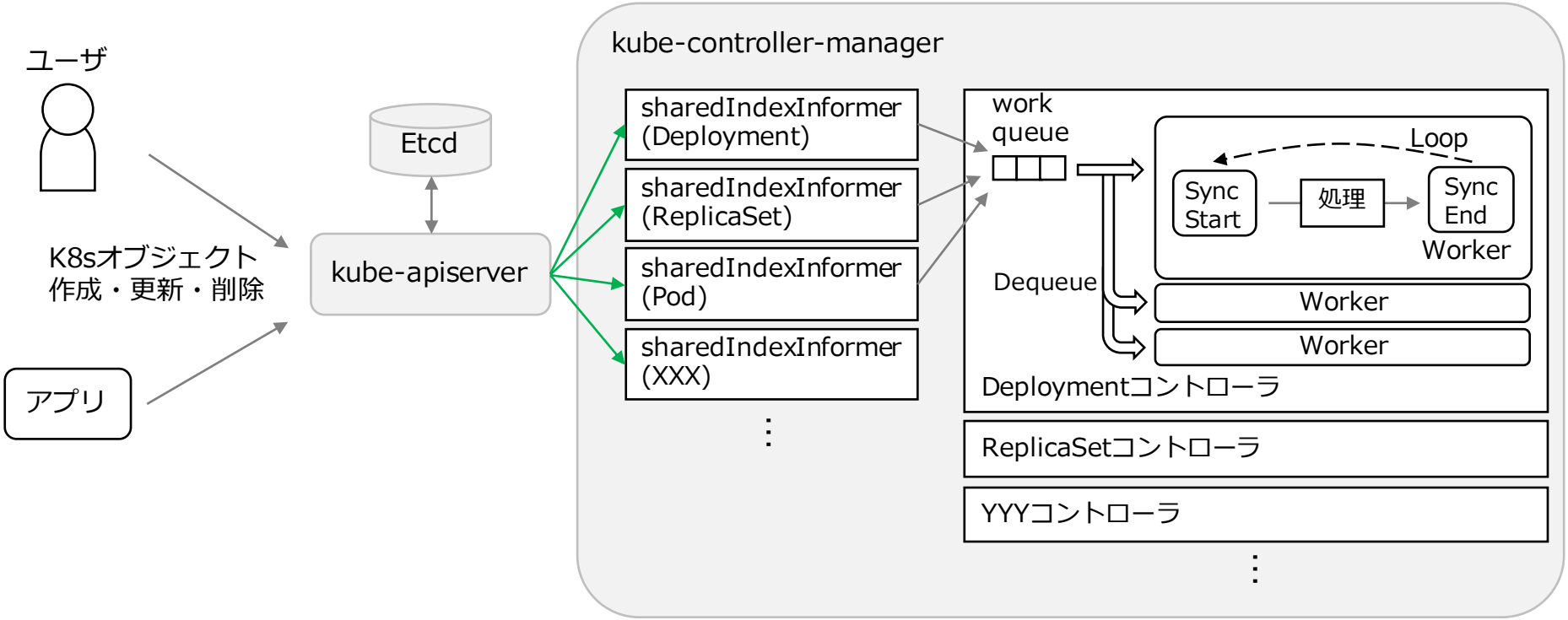
FUJITSU

- 名前
  - 石井伴旺
- 経歴
  - 2016年
    富士通株式会社に入社
  - 2016~2019年
    OpenStack・Ceph製品のセールスエンジニア
  - 2019年~
    OpenStack・K8sなどのOSSコミュニティでの開発活動に携わる

K8sクラスタでのkube-controller-managerの役割と概要

FUJITSU

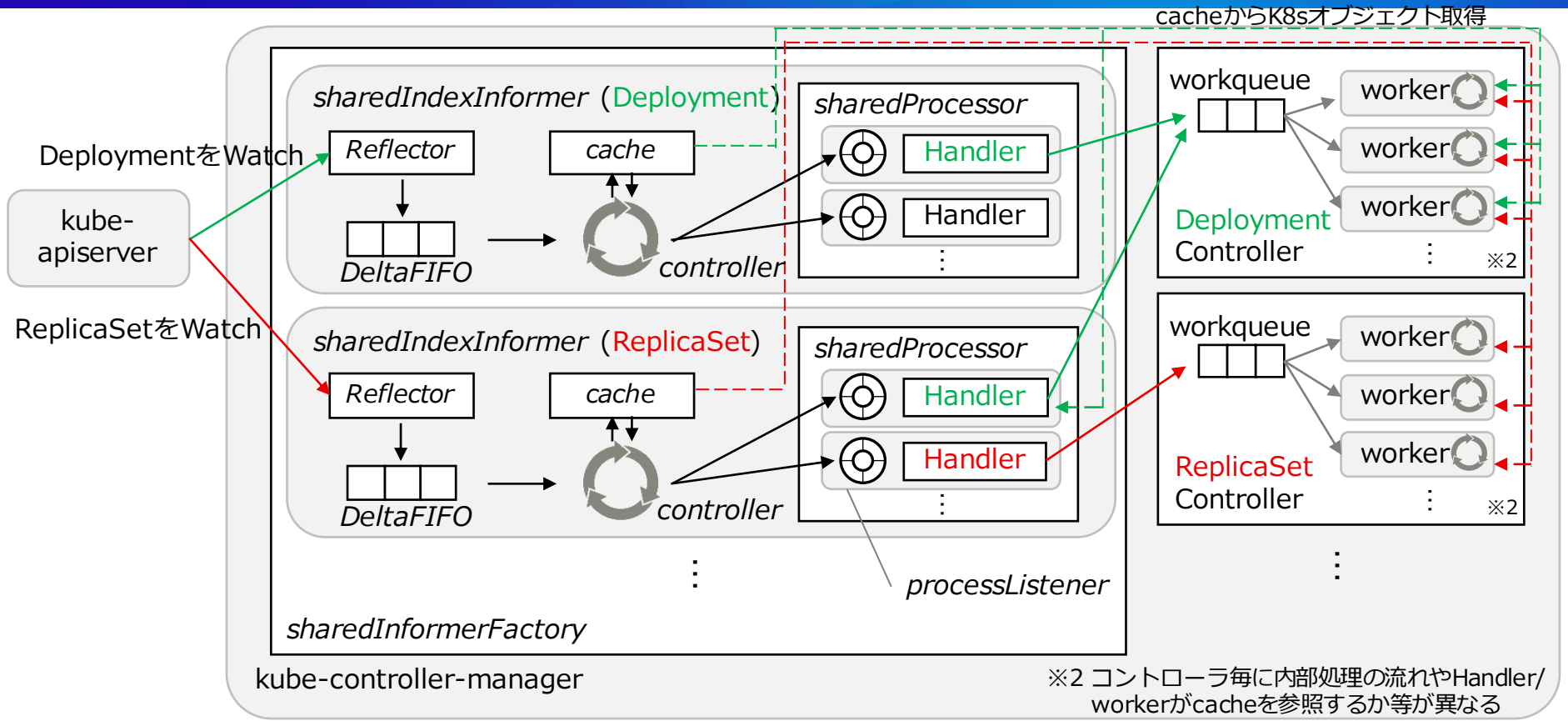
○ Etcdに保存されたK8sオブジェクトの定義に従ってK8sクラスタ内の処理を行う



K8sオブジェクトの監視(Watch)

kube-controller-managerの内部コンポーネント

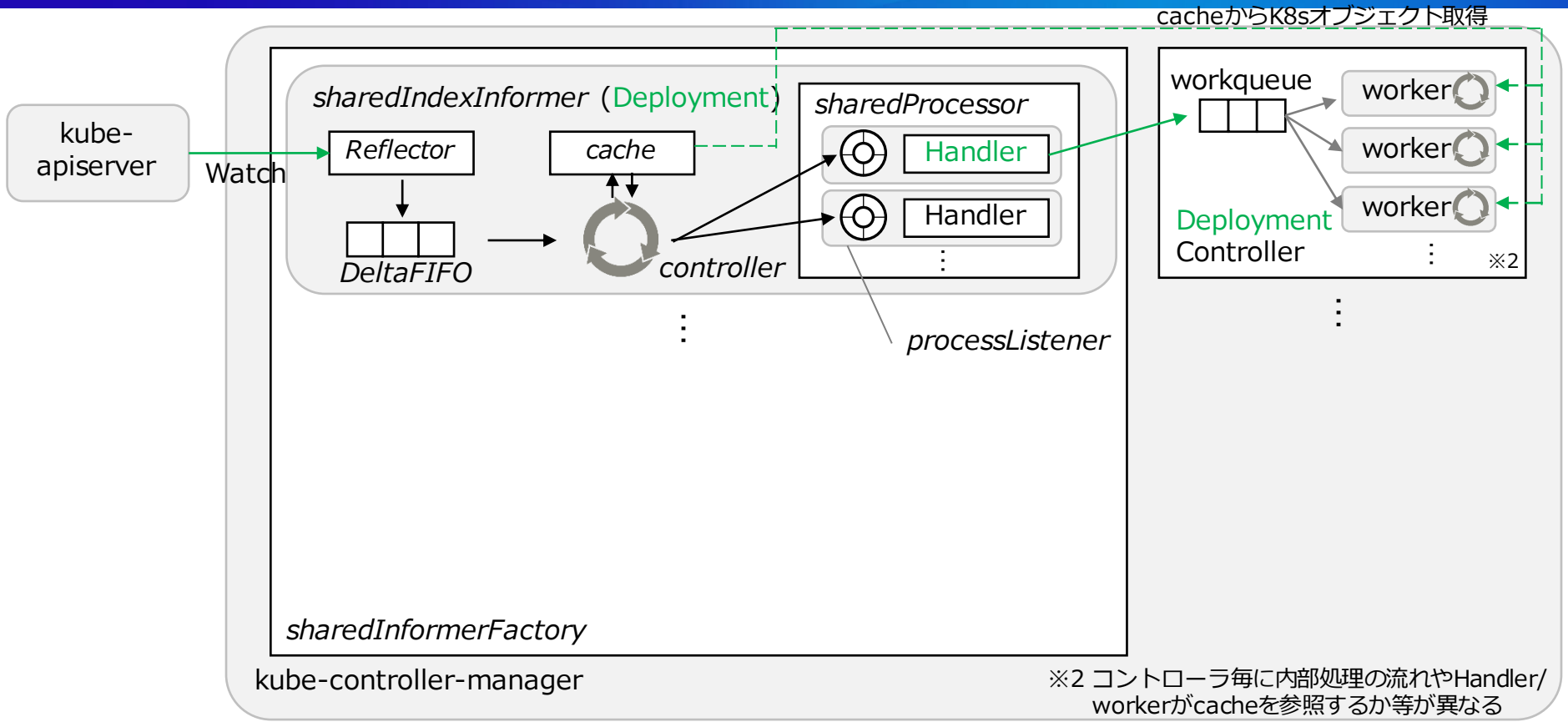
FUJITSU



※斜体はGoのType名

kube-controller-managerでのK8sオブジェクト処理フロー

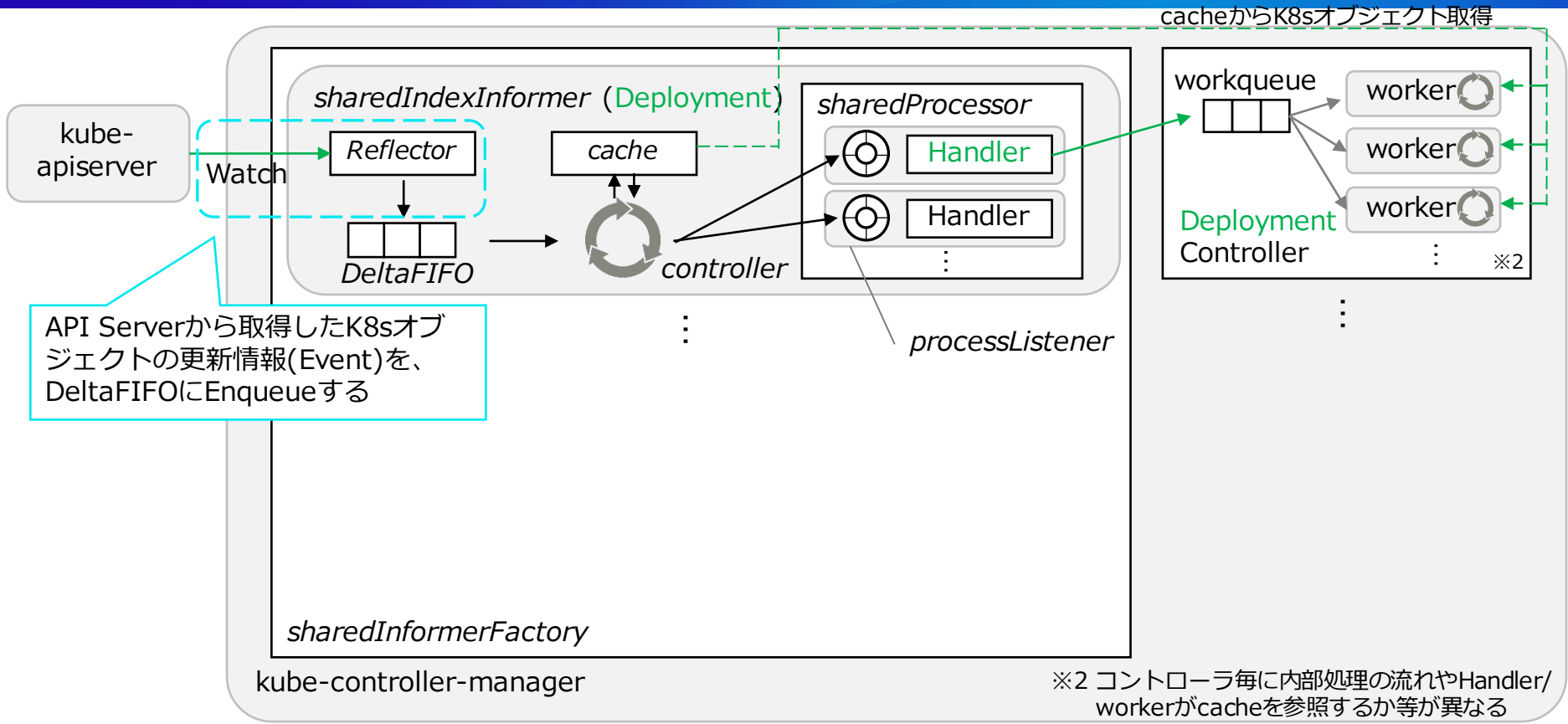
FUJITSU



※斜体はGoのType名

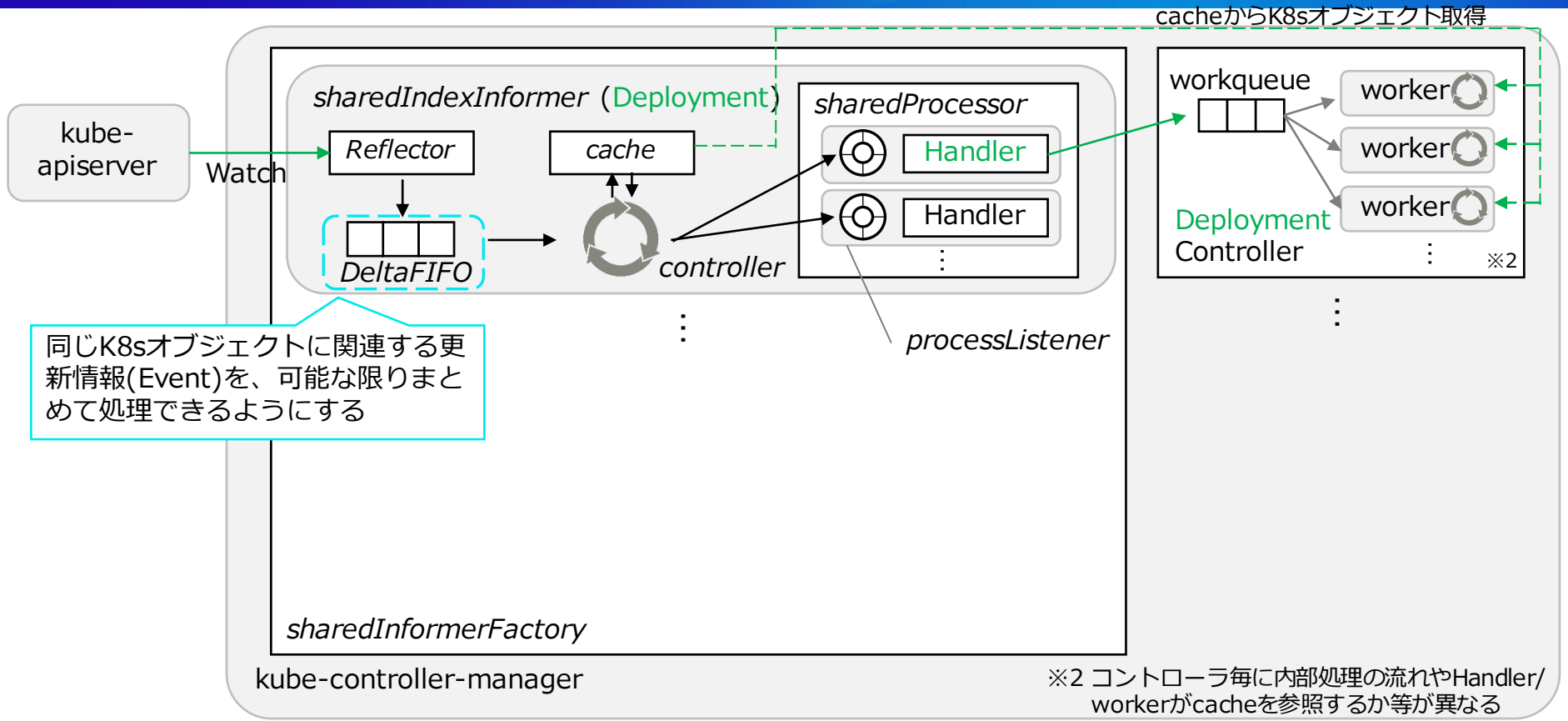
kube-controller-managerでのK8sオブジェクト処理フロー

FUJITSU



※斜体はGoのType名

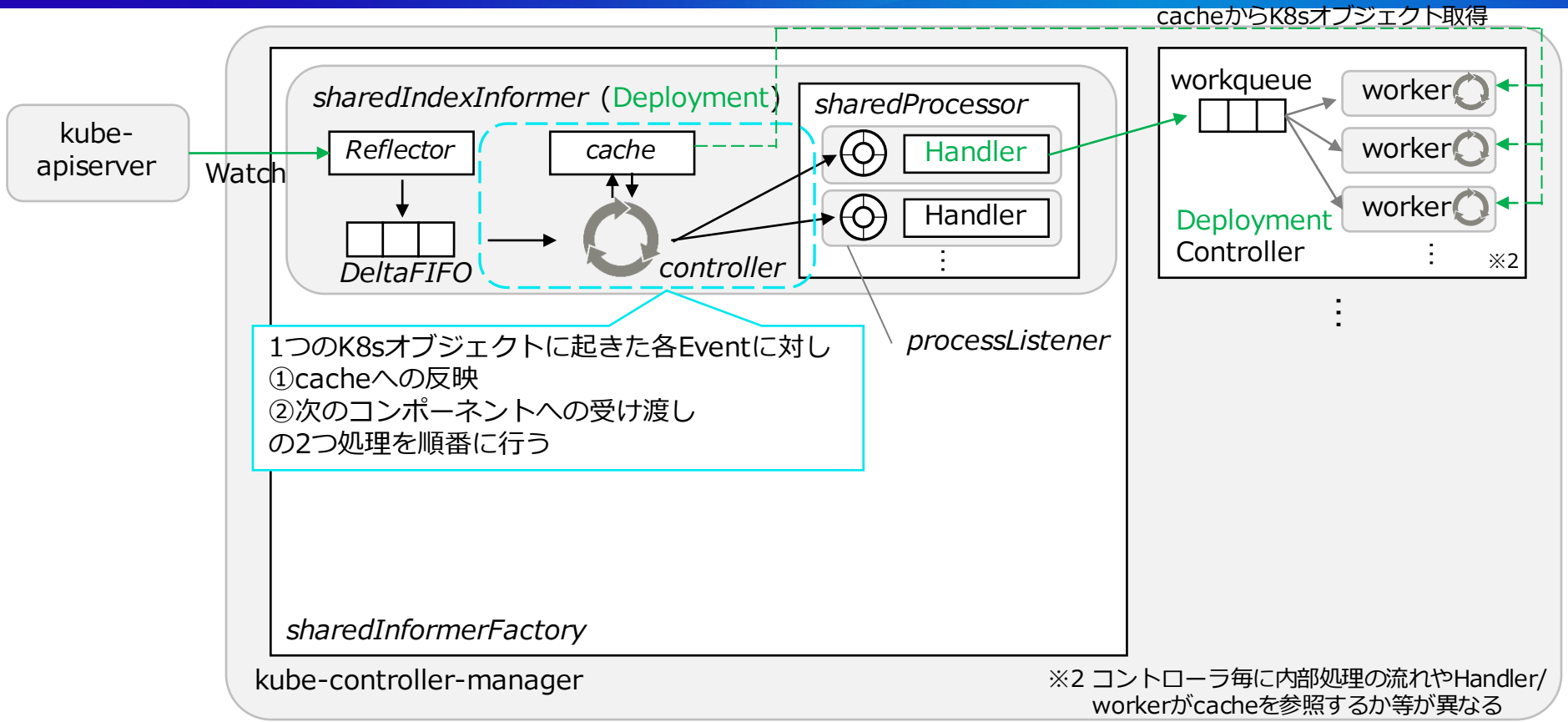
kube-controller-managerでのK8sオブジェクト処理フロー



※斜体はGoのType名

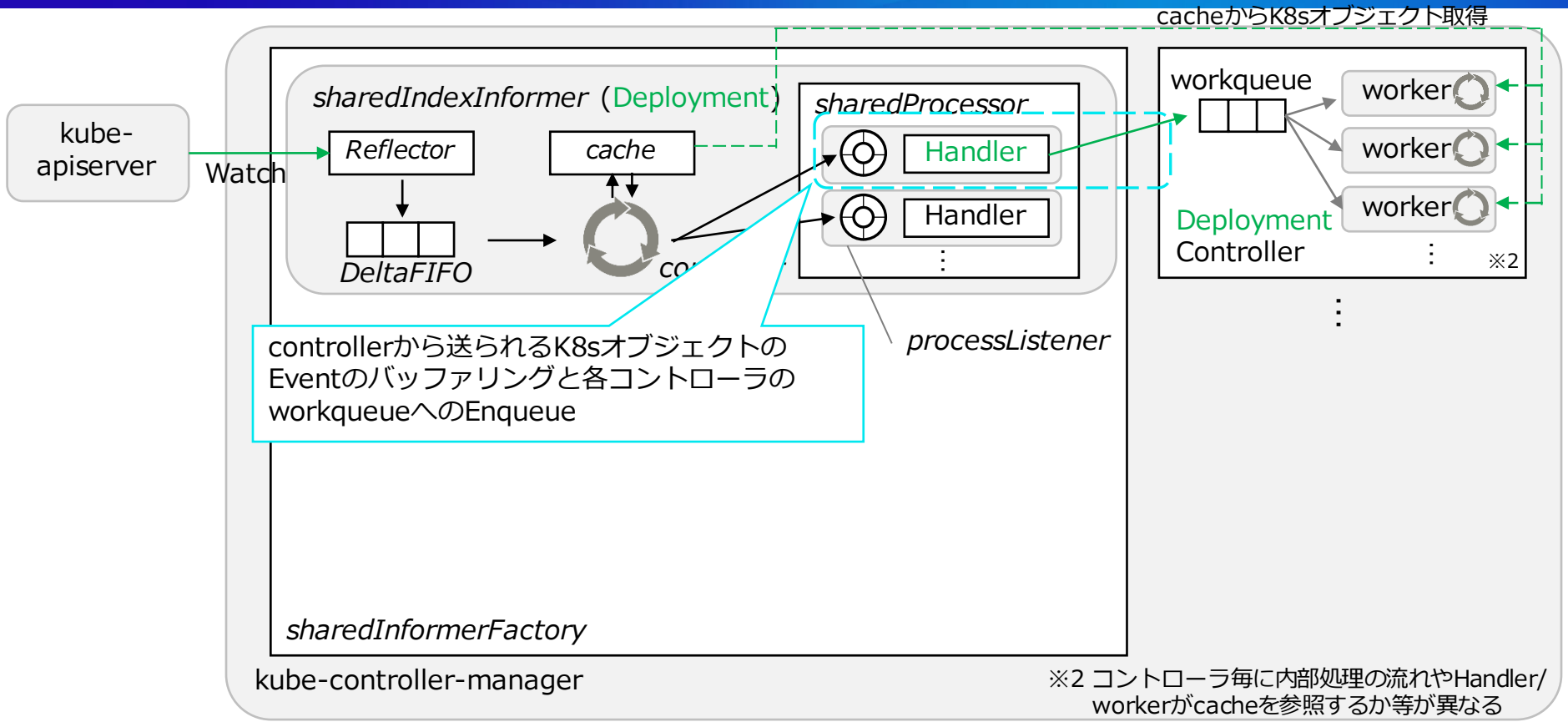
kube-controller-managerでのK8sオブジェクト処理フロー

FUJITSU



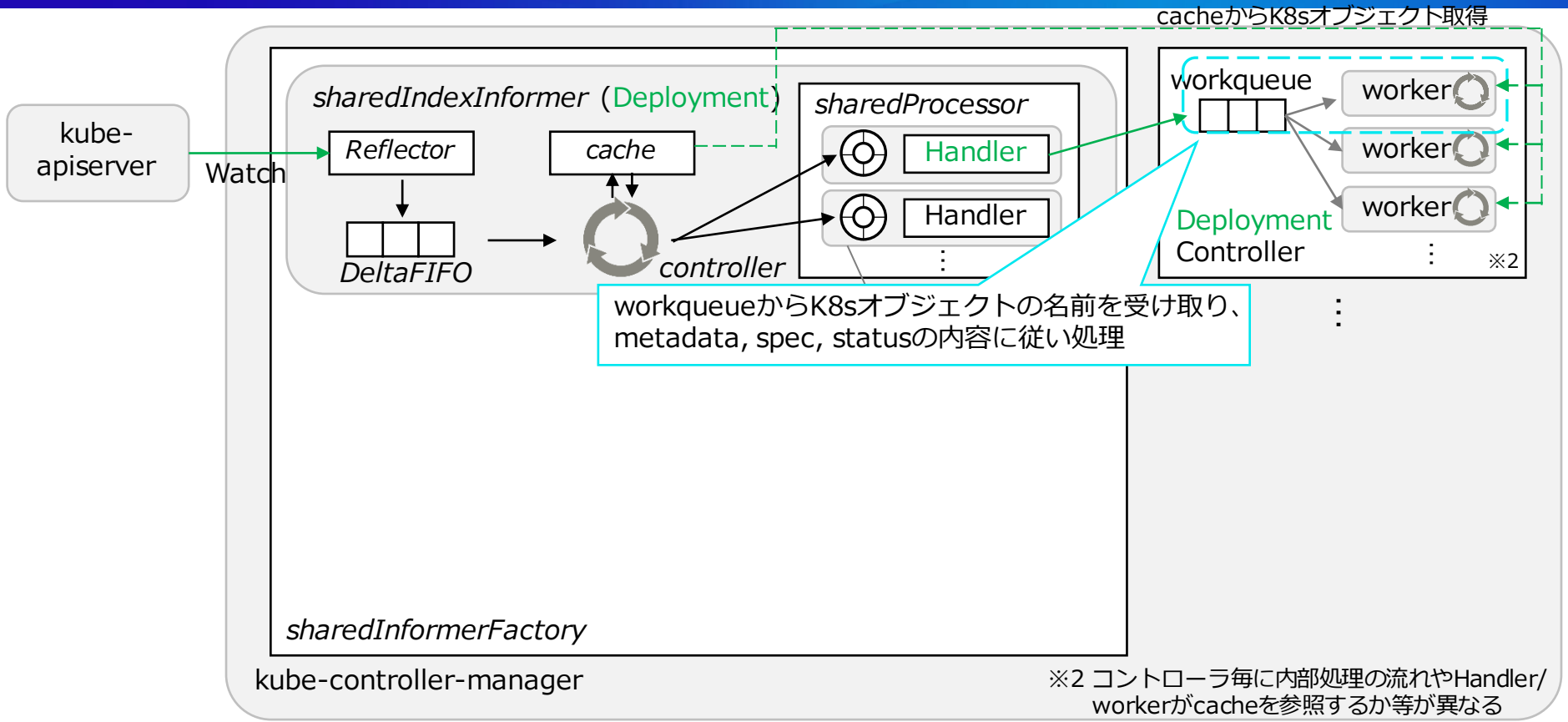
※斜体はGoのType名

kube-controller-managerでのK8sオブジェクト処理フロー



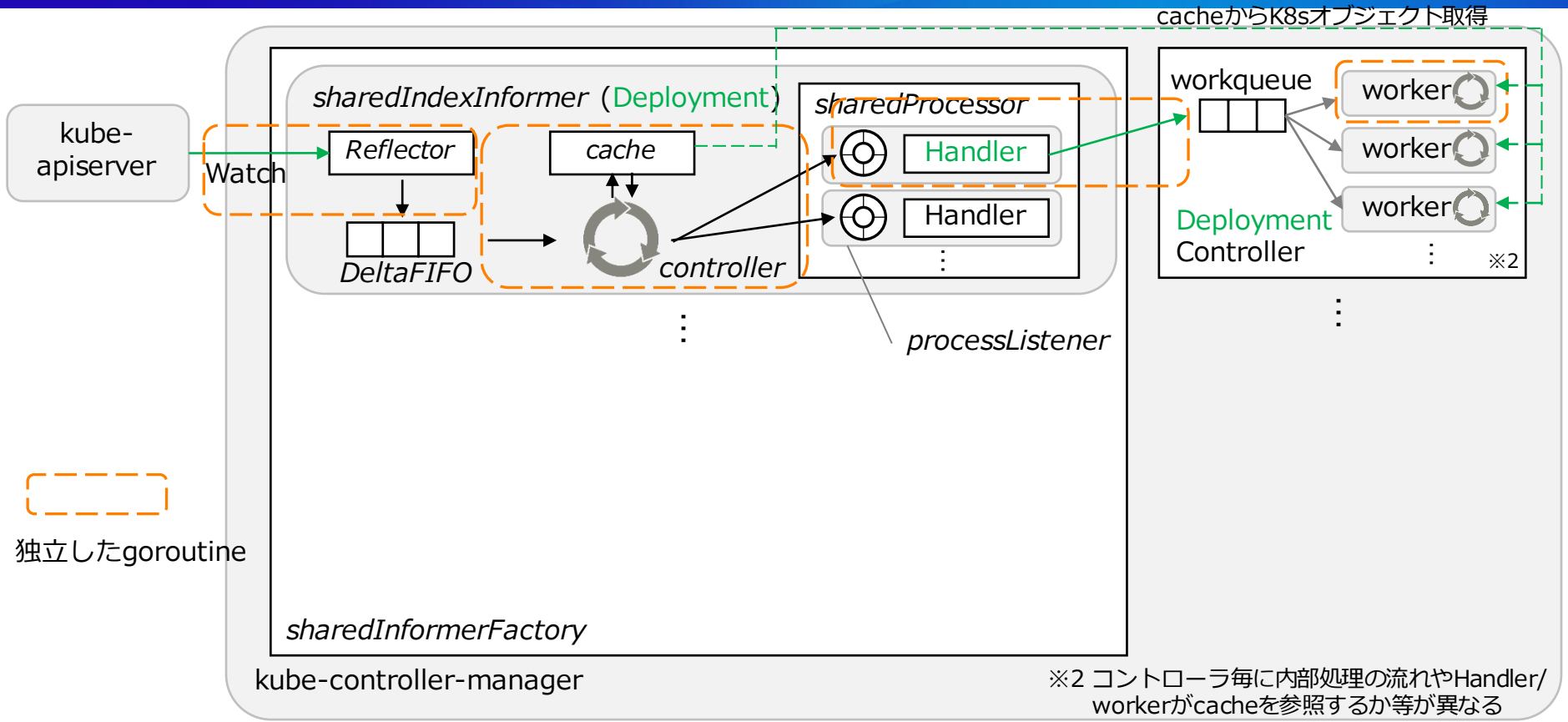
※斜体はGoのType名

kube-controller-managerでのK8sオブジェクト処理フロー



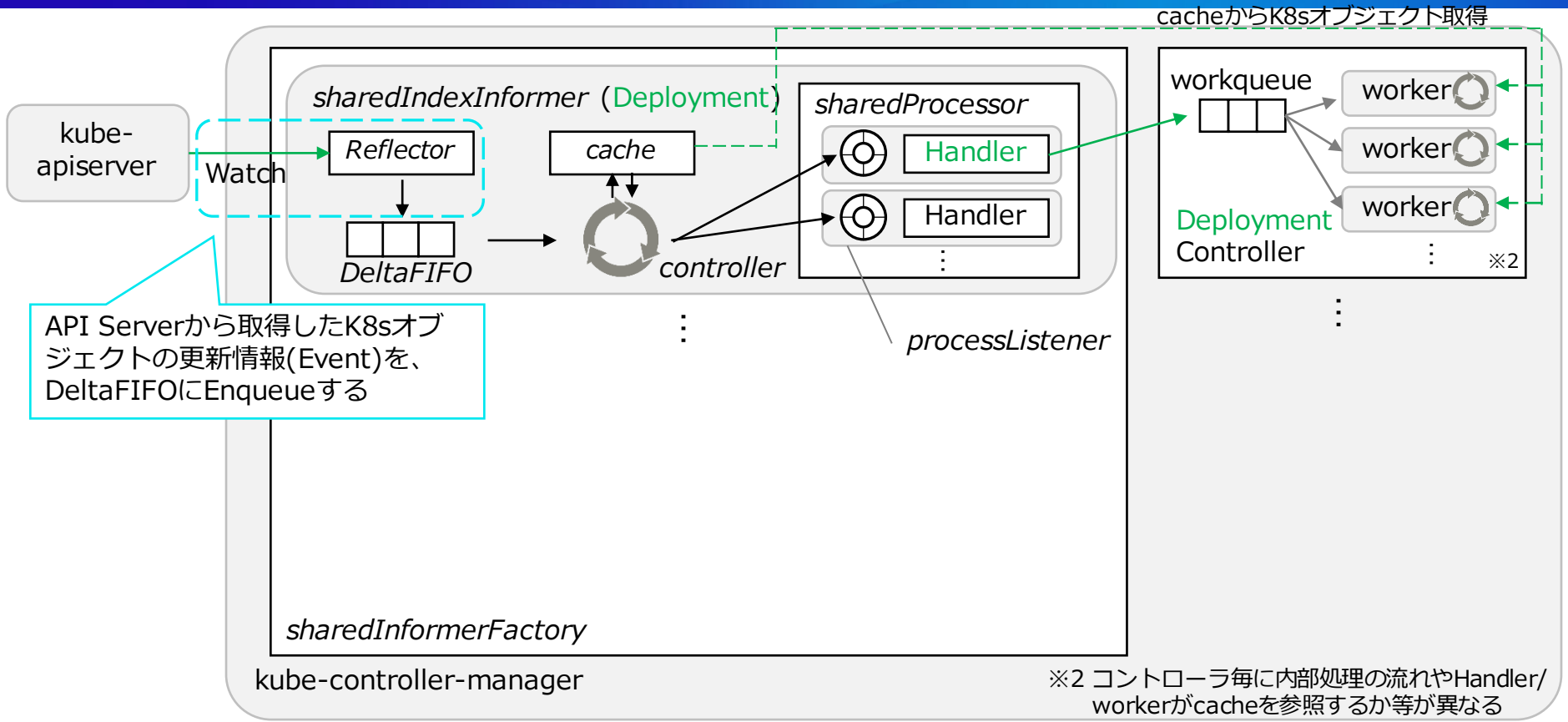
※斜体はGoのType名

kube-controller-managerでのK8sオブジェクト処理フロー



※斜体はGoのType名

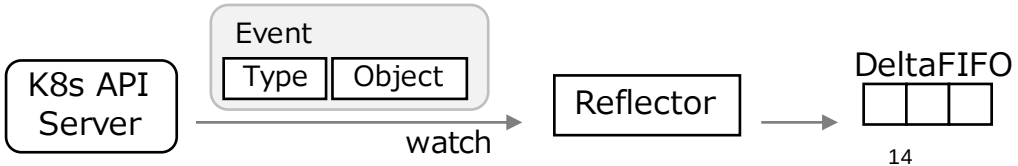
Reflectorについて



※斜体はGoのType名

Reflectorの役割

- 「K8s APIサーバから流れてくるK8sリソースのEvent」をDeltaFIFOにエンキューする
  - (例) DeploymentのsharedIndexInformer内のReflectorは、K8sクラスタ内の全Deploymentオブジェクトの各EventをK8s APIサーバから取得・DeltaFIFOにエンキューしている
- EventはEventTypeとオブジェクト情報から成る
  - オブジェクト情報：Event発生時のEtcd内のオブジェクトデータ
  - EventTypeは3種類
    - Added
      K8sクラスタ内で新しいK8sオブジェクトが作成された際にK8s APIサーバから送られる
    - Modified
      K8sクラスタ内の既存K8sオブジェクトが更新された時にK8s APIサーバから送られる
    - Deleted
      K8sクラスタ内のK8sオブジェクトが削除された時にK8s APIサーバから送られる



Event

```
{
  "type": "MODIFIED",      ← EventType
  "object": {
    "kind": "Deployment",
    "apiVersion": "apps/v1",
    "metadata": {
      "name": “dep1",
      "namespace": "default",
      "uid": "6520f999-3691-4402-af4c-d227a2da27bb",
      …
    },
    "spec": {
      "replicas": 1,
      "selector": {
        …
      },
      …
    },
    "status": {}
  }
}
```

Event発生時にEtcdが保持しているDeployment/dep1データ

Reflectorのコードについて

FUJITSU

*1 staging/src/k8s.io/client-go/tools/cache/shared_informer.go
*2 staging/src/k8s.io/client-go/tools/cache/controller.go
*3 staging/src/k8s.io/client-go/tools/cache/reflector.go

○ Reflector作成の契機

*sharedIndexInformer.Run
- DeltaFIFOの作成
- K8s APIをWatchする際に使用するlistWatcherの指定

(*1)

呼出

*controller.Run
- *Reflector*の設定
- *Reflector*の作成

(*2)

呼出

*Reflector.Run
- listWatcherからのEvent取得
- EventTypeを見てDeltaFIFOへエンキュー

(*3)

○ Eventの取得とDeltaFIFOへのエンキュー

```
func (r *Reflector) watchHandler(…) {
    …
    for {
        select {
            case event, ok := <-w.ResultChan():
                …
                switch event.Type {
                case watch.Added:
                    err := r.store.Add(event.Object)
                case watch.Modified:
                    err := r.store.Update(event.Object)
                case watch.Deleted:
                    err := r.store.Delete(event.Object)
                }
    …
```

(*3)

Reflector.Runが内部で呼び出すメソッド

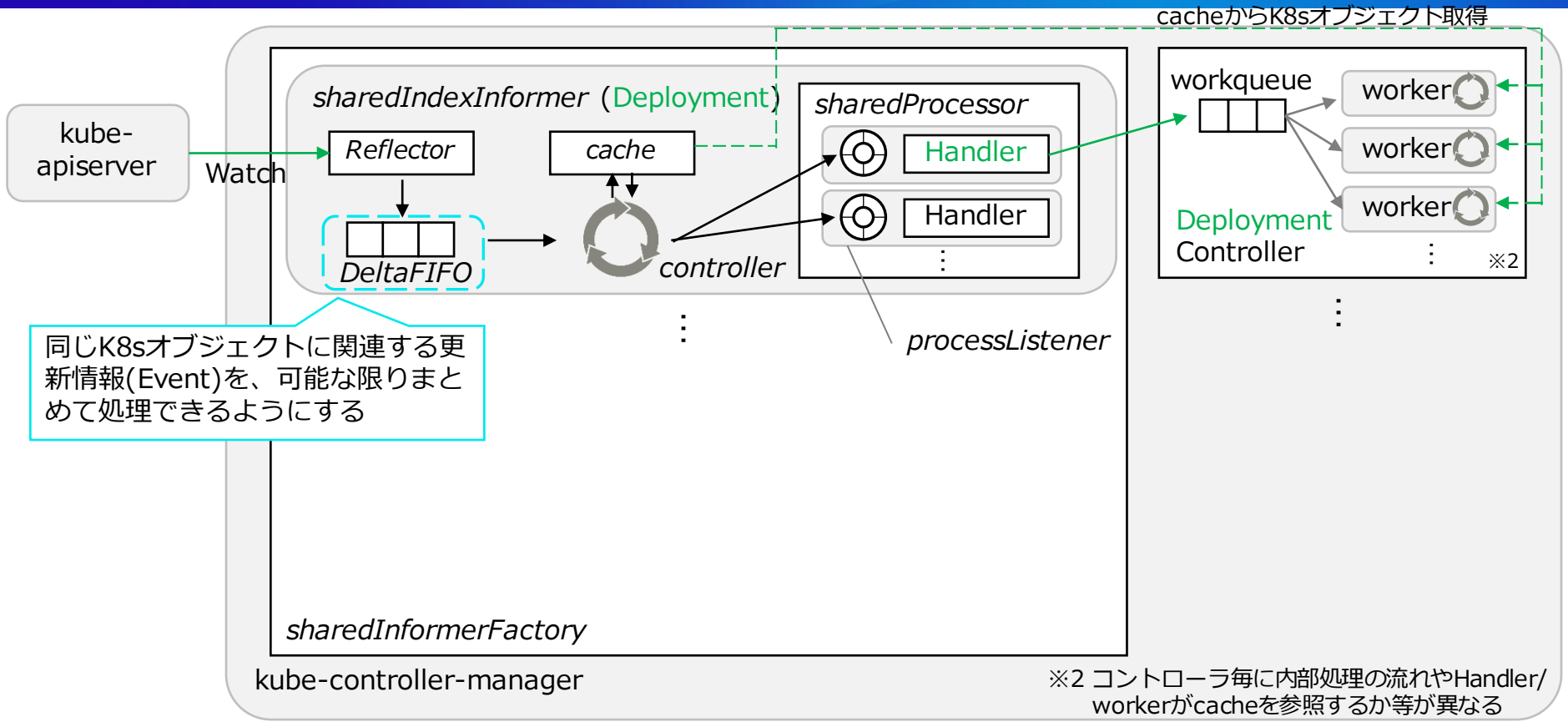
listWatcherからのEvent取得

EventTypeを見てDeltaFIFOへエンキュー

```
type Reflector {
    name string
    store Store  //DeltaFIFO
    listWatcher ListWatcher
    …
}
```

(*3)

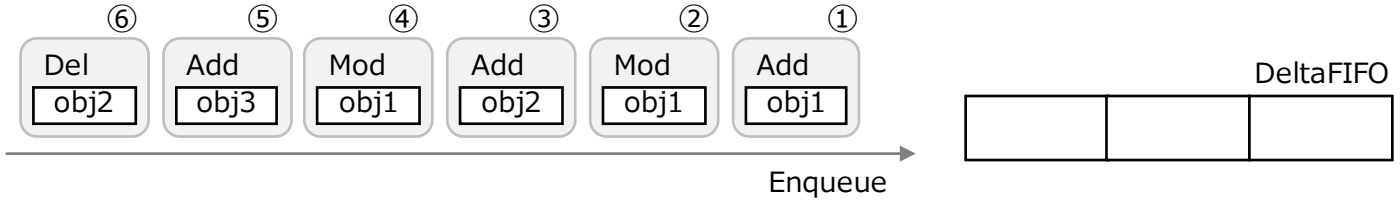
DeltaFIFOについて

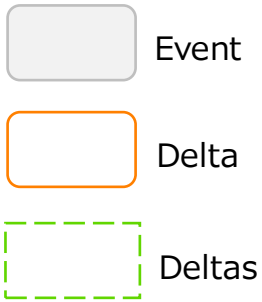


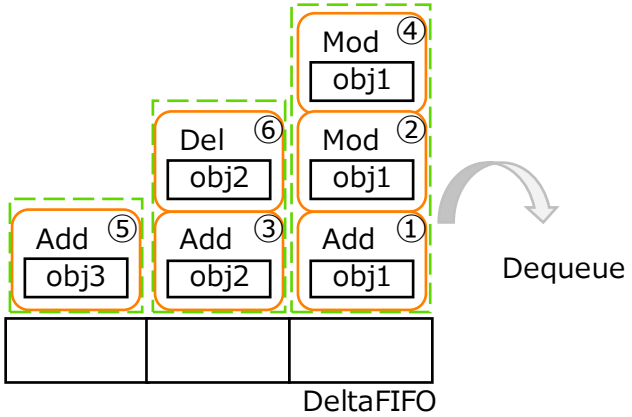
※斜体はGoのType名

DeltaFIFOの役割

- あるK8sオブジェクトに対する連続したEventをまとめて処理できるようにする
  - 同じオブジェクトに対するEventをDeltaタイプに変換して積み上げ、まとめてデキューする







DeltaFIFOのコードについて

*1 staging/src/k8s.io/client-go/tools/cache/shared_informer.go
*2 staging/src/k8s.io/client-go/tools/cache/delta_fifo.go

○ DeltaFIFO作成の契機

```
func (s *sharedIndexInformer) Run(…) {
    …
    fifo := NewDeltaFIFOWithOption(DeltaFIFOOptions{
        KnownObjects: s.indexer,
        EmitDelayTypeReplaced: true,
    })
…                                                    (*1)
```

○ DeltaFIFO内部構造

```
type DeltaFIFO struct {
    items map[string]Deltas
    queue []string
    keyFunc KeyFunc
    …
}
```
(*2)

Key:　K8sオブジェクトの <*namespace*>/<*name*> ペア
Value: 積み上げたEvent(Deltas)

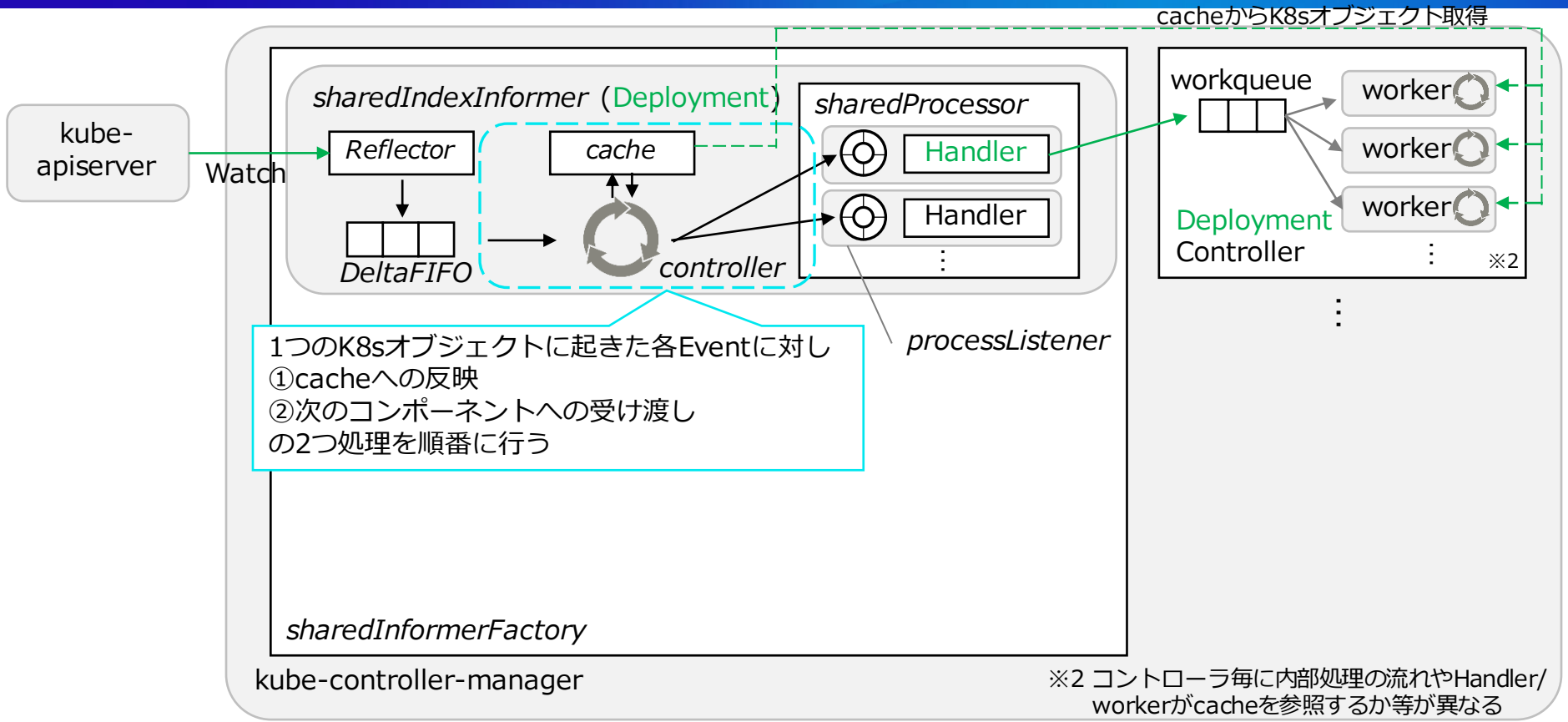
FIFOキューの機能を提供するため、エンキューされたK8sオブジェクトの順番を保持

*items*のKey, *queue*のelementとして使用している文字列 <*namespace*>/<*name*> を K8sオブジェクトから作成する関数

```
type Deltas []Delta
type Delta struct {
    Type DeltaType  // Added/Updated/Deleted
    Object interface{}  // K8s Object
}
```
(*2)

(*DeltaFIFO)のAdd, Update, DeleteメソッドにK8s objectを引数として渡し
- DeltaFIFOへのエンキュー
- ADDED, MODIFIED, DELETED EventのDeltaデータ型への変換・積み上げ を行う

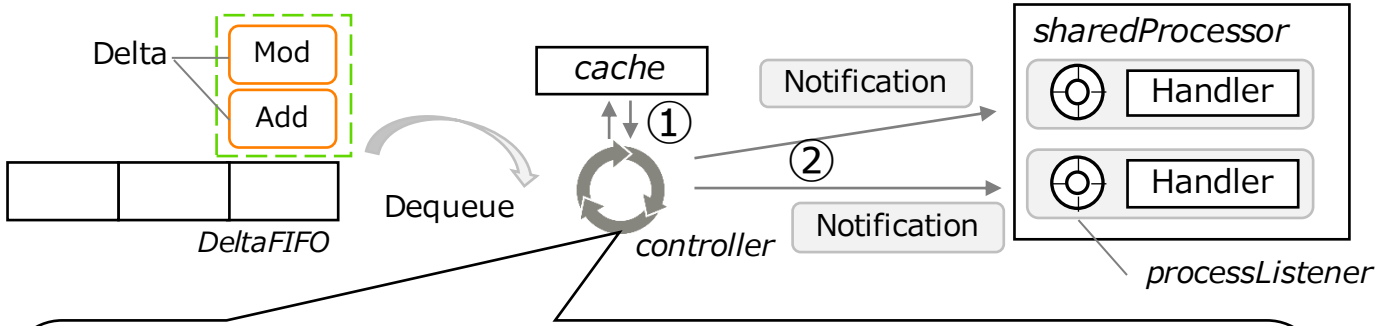
controllerについて



※斜体はGoのType名

controllerの役割

- DeltaFIFOから取得したK8sオブジェクトの状態をcacheに反映した後、次のコンポーネント(processListener)にK8sオブジェクトの情報(Notification)を渡す
  - DeltaFIFOから取り出したK8sオブジェクトに関連する各Deltaに対しcache反映・processListnerへの引き渡しを行う



DeltaFIFOからDequeueした各Deltaに対し以下2つの処理を順番に行う
① 各DeltaのDeltaTypeが
・Added/Updatedのときは、Delta内に保持されているK8sオブジェクトをcacheに保存
・Deletedのときは、Deltaに紐づいているK8sオブジェクトをcacheから削除
② Deltaが保持するK8sオブジェクトとcacheが保持しているK8sオブジェクト情報から
Notificationというデータを作成し、processListenerに渡す

図の例では、Addに対し①②の処理を行ったあと、Modに対し①②の処理を行う

controllerのコードについて

*1 staging/src/k8s.io/client-go/tools/cache/shared_informer.go
*2 staging/src/k8s.io/client-go/tools/cache/controller.go

○ controller作成・動作契機

*sharedIndexInformer.Run
- controllerの作成

(*1)

呼出

*controller.Run
- controller.processLoopを呼び以下実行
  * DeltaFIFOからのDequeue
  * cacheへの更新
  * processListenerへのNotification発行

(*2)

```
type controller struct {
    config Config
    …
}
```

(*2)

```
type Config struct {
    Queue
    Process
    …
}
```

(*2)

DeltaFIFO

sharedIndexInformer.HandleDeltasメソッドがセットされる

○ controller.processLoopの処理

```
func (c *controller) processLoop() {
    for {
        obj, err := c.config.Queue.Pop(PopProcessFunc(c.config.Process))
    …
}
```

(*2)

以下の処理をループする
DeltaFIFOからDequeueしたDeltasタイプの変数をsharedIndexInformerのHandleDeltasメソッドに引数として渡す

controllerのコードについて

FUJITSU

*1 staging/src/k8s.io/client-go/tools/cache/shared_informer.go

○ controller.processLoopの処理

```
func (s *sharedIndexInformer) HandleDeltas(obj interface{}) error {
    …
    for _, d := range obj.(Deltas) {
        switch d.Type {
        case Sync, Replaced, Added, Updated:
            …
            if old, exists, err := s.indexer.Get(d.Object); err == nil && exists {
                if err := s.indexer.Update(d.Object); err != nil {
                    …
                    }
                    …
                    s.processor.distribute(updateNotification{oldObj: old, newObj: d.Object}, isSync)
            } else {
                    if err := s.indexer.Add(d.Object); err != nil {
                        return err
                    }
                    s.processor.distribute(addNotification{newObj: d.Object}, false)
            }
        case Deleted:
                    if err := s.indexer.Delete(d.Object); err != nil {
                        return err
                    }
                    s.processor.distribute(deleteNotification{oldObj: d.Object}, false)
```

(*1)

sharedIndexInformer.indexerが*cache*

*processListener*にNotificationを送るためにprocessListenerを管理しているsharedProcessorのdistributeメソッドを実行

controllerのコードについて

FUJITSU

○ controller.processLoopでの次のコンポーネント(processListener)にNotificationを送る処理

(*1)
```
func NewSharedIndexInformer(…) SharedIndexInformer {
    realClock := &clock.RealClock{}
    sharedIndexInformer := &sharedIndexInformer{
        processor:      &sharedProcessor{clock: realClock},
        …
    }
    return sharedIndexInformer
}
```

(*2)
```
type sharedProcessor struct {
    listenersStarted bool
    listenersLock    sync.RWMutex
    listeners        []*processorListener
    syncingListeners []*processorListener
    clock            clock.Clock
    wg               wait.Group
}
```

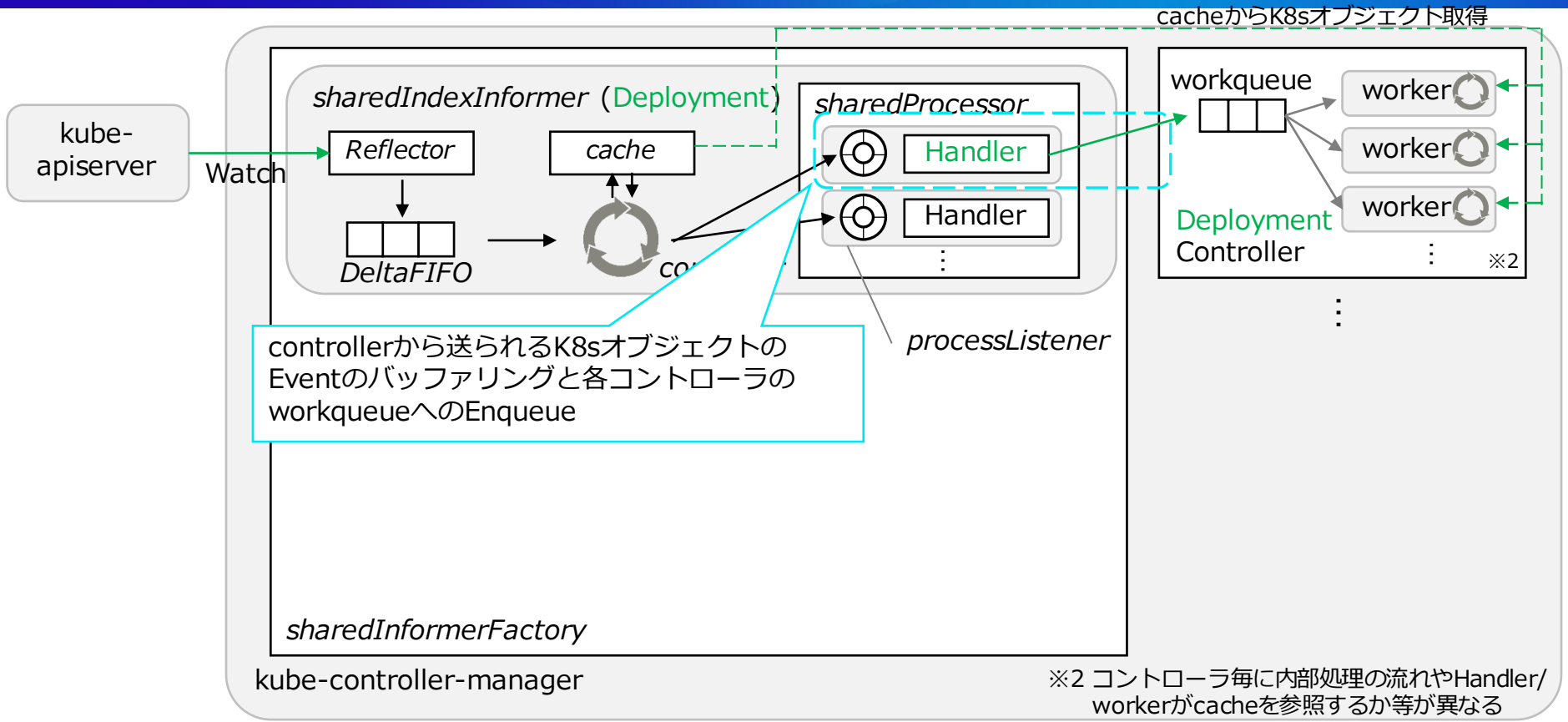
sharedProcessorが管理しているprocessListener

```
func (p *sharedProcessor) distribute(obj interface{}, sync bool) {
    …
    if sync {
        …
    } else {
        for _, listener := range p.listeners {
        listener.add(obj)
        }
    }
}
```
(*2)

各processListenerのaddメソッドを呼んでNotification(*obj*)を渡している

*1 staging/src/k8s.io/client-go/tools/cache/reflector.go
*2 staging/src/k8s.io/client-go/tools/cache/shared_informer.go

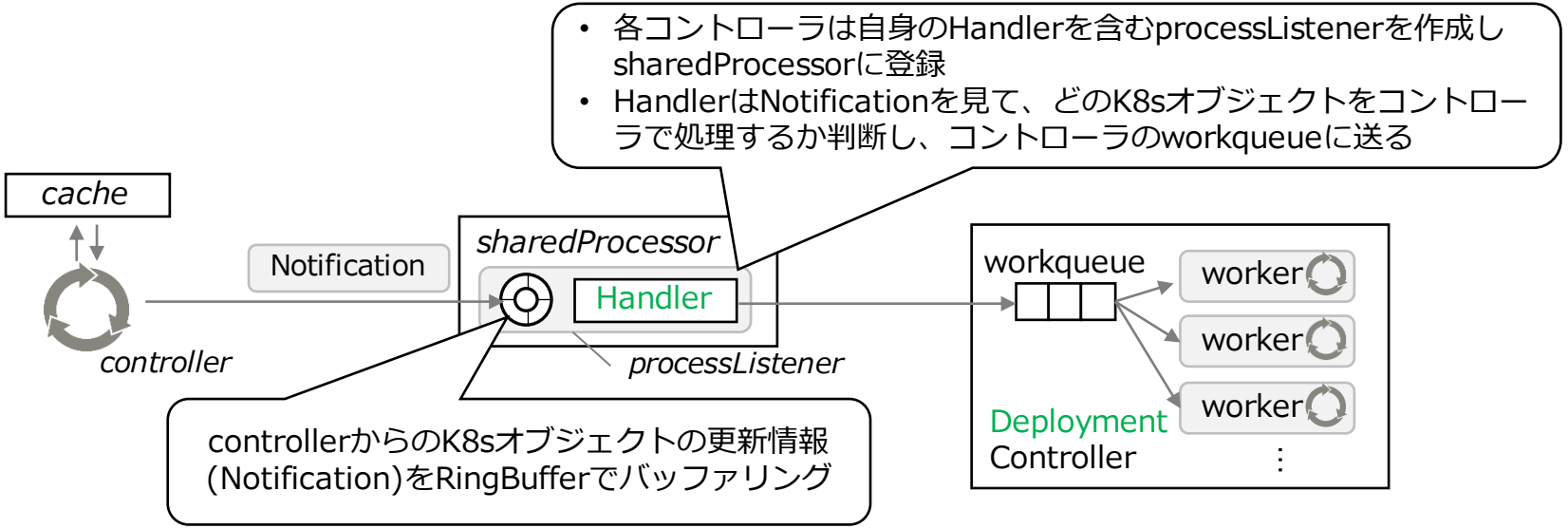
processListenerについて



※斜体はGoのType名

processListenerの役割

- controllerから渡されるNotificationのバッファリングと、Notificationから各コントローラが実際に処理するK8sオブジェクトの抽出・コントローラworkqueueへのエンキューを行う



※ コントローラ毎にコントローラ内部処理の流れやHandler/workerがcacheを参照するか等が異なる

processListenerのコードについて

FUJITSU

*1 pkg/controller/deployment/deployment_controller.go

*2 staging/src/k8s.io/client-go/tools/cache/shared_informer.go

- processListener作成の契機
  - 各コントローラ作成時に自身のHandlerをsharedIndexInformer.AddEventHandlerを呼んで登録

```
func NewDeploymentController(dInformer appsinformers.DeploymentInformer, …) (*DeploymentController, error) {
    …
    dInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
        AddFunc:    dc.addDeployment,
        UpdateFunc: dc.updateDeployment,
        DeleteFunc: dc.deleteDeployment,
(*1) })
```

Handler

Deployment sharedIndexInformerにDeploymentコントローラが自身のHandlerを登録している箇所

```
func (s *sharedIndexInformer) AddEventHandler(handler ResourceEventHandler) {
    s.AddEventHandlerWithResyncPeriod(handler, s.defaultEventHandlerResyncPeriod)
}
…
func (s *sharedIndexInformer) AddEventHandlerWithResyncPeriod(handler ResourceEventHandler, …) {
    …
    listener := newProcessListener(handler, …)
    …
    if !s.started {
        s.processor.addListener(listener)
        return
    }
    …
    s.processor.addListener(listener)
```

(*2)

processListenerの作成

processListenerを管理するsharedProcessorに作成したprocessListenerを登録

Notificationからコントローラで処理するK8sオブジェクトを判断するための処理
- AddFunc:
  addNotificationを受け取った際に実行
- UpdateFunc:
  updateNotificationを受け取った際に実行
- DeleteFunc:
  deleteNotificationを受け取った際に実行

processListenerのコードについて

FUJITSU

*1 staging/src/k8s.io/client-go/tools/cache/shared_informer.go

○ processListenerの動作契機

*sharedIndexInformer.Run
- processListenerを管理するsharedProcessorをrun

(*1)

呼出

*sharedProcessor.run
- 管理するprocessListenerのpop・run各メソッドを独立goroutineで実行
- sharedProcessor.run実行後にsharedProcessor.addListenerで追加されたprocessListenerのpop・runメソッドはaddListener実行時に実行される

(*1)

```
func (p *sharedProcessor) run(stopCh <-chan struct{}) {
    func() {
        …
        for _, listener := range p.listeners {
            p.wg.Start(listener.run)
            p.wg.Start(listener.pop)
        }
(*1)    …
```

sharedProcessorが管理する各processListenerのrun, popメソッドを個別のgoroutineで実行

○ 受け取ったNotificationのprocessListenerでの処理

```
func (p *sharedProcessor) distribute(obj interface{}, …) {
    …
        for _, listener := range p.listeners {
        listener.add(obj)
        }
(*1)    …
```

```
func (p *processorListener) add(notification interface{}) {
      p.addCh <- notification
}                                                    (*1)
```

processListenerは自身のGoチャンネル(*addCh*)を通してNotificationを受け取る

processListenerのコードについて

FUJITSU

*1 staging/src/k8s.io/client-go/tools/cache/shared_informer.go

○ processListenerでのNotificationのバッファリング

```
func (p *processorListener) pop() {
    …
    for {
        select {
        case nextCh <- notification:
            var ok bool
            notification, ok = p.pendingNotifications.ReadOne()
            if !ok { // Nothing to pop
                nextCh = nil // Disable this select case
            }
        case notificationToAdd, ok := <-p.addCh:
            if !ok {
                return
            }
            if notification == nil {
                notification = notificationToAdd
                nextCh = p.nextCh
            } else { // There is already a notification waiting to be dispatched
                p.pendingNotifications.WriteOne(notificationToAdd)
            }
        }
    }
}
```

p.nextCh (Goチャンネル)経由で次の処理(run)にNotificationを渡す

RingBufferから受け取ったNotificationをデキュー

Notificationの受け取り

RingBufferが空の時はバッファを介さず次の処理(run)にNotificationを渡すようにする

RingBufferへ受け取ったNotificationをエンキュー

(*1)

processListenerのコードについて

FUJITSU

*1 staging/src/k8s.io/client-go/tools/cache/shared_informer.go

○ HandlerへのNotification内 K8sオブジェクトの受け渡し

```
func (p *processorListener) run() {
    …
    stopCh := make(chan struct{})
    wait.Until(func() {
        for next := range p.nextCh {
            switch notification := next.(type) {
            case updateNotification:
                p.handler.OnUpdate(notification.oldObj, notification.newObj)
            case addNotification:
                p.handler.OnAdd(notification.newObj)
            case deleteNotification:
                p.handler.OnDelete(notification.oldObj)
            default:
                utilruntime.HandleError(fmt.Errorf("unrecognized notification: %T", next))
            }
        }
        close(stopCh)
    }, 1*time.Second, stopCh)
}
```

popからNotificationの受け取り

Notificationの種類に応じて
Handler関数に、Notification内
のK8sオブジェクトをわたす

(*1)

processListenerのコードについて

FUJITSU

*1 pkg/controller/deployment/deployment_controller.go

○ Deployment sharedIndexInformerにDeploymentコントローラが登録したHandlerの処理

```
func NewDeploymentController(dInformer appsinformers.DeploymentInformer, …) (*DeploymentController, error) {
    …
    dInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
        AddFunc:    dc.addDeployment,
        UpdateFunc: dc.updateDeployment,
        // This will enter the sync loop and no-op, because the deployment has been deleted from the store.
        DeleteFunc: dc.deleteDeployment,
    })
```

(*1)

```
func (dc *DeploymentController) addDeployment(obj interface{}) {
    d := obj.(*apps.Deployment)
    klog.V(4).InfoS("Adding deployment", "deployment", klog.KObj(d))
    dc.enqueueDeployment(d)
}
```

(*1)

DeploymentControllerのenqueueDeployment
フィールドの中身はenqueueメソッド

```
func (dc *DeploymentController) enqueue(deployment *apps.Deployment) {
    key, err := controller.KeyFunc(deployment)
    …
    dc.queue.Add(key)
}
```

(*1)

K8sオブジェクトを一意に識別する文字列(Key)
<namespace>/<name> を作成

DeploymentコントローラのworkqueueにKeyをエンキュー

processListenerのコードについて

FUJITSU

*1 pkg/controller/replicaset/replica_set.go

○ Pod sharedIndexInformerにReplicaSetコントローラが登録したHandlerの処理

```
func NewBaseController(…, podInformer coreinformers.PodInformer, …) *ReplicaSetController {
    …
    podInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
        AddFunc: rsc.addPod,
        UpdateFunc: rsc.updatePod,
        DeleteFunc: rsc.deletePod,
    })
```

(*1)

```
func (rsc *ReplicaSetController) addPod(obj interface{}) {
    pod := obj.(*v1.Pod)
    …
    if controllerRef := metav1.GetControllerOf(pod); controllerRef != nil {
        rs := rsc.resolveControllerRef(pod.Namespace, controllerRef)
        …
        rsKey, err := controller.KeyFunc(rs)
        …
        rsc.queue.Add(rsKey)
        return
    }
```

(*1)

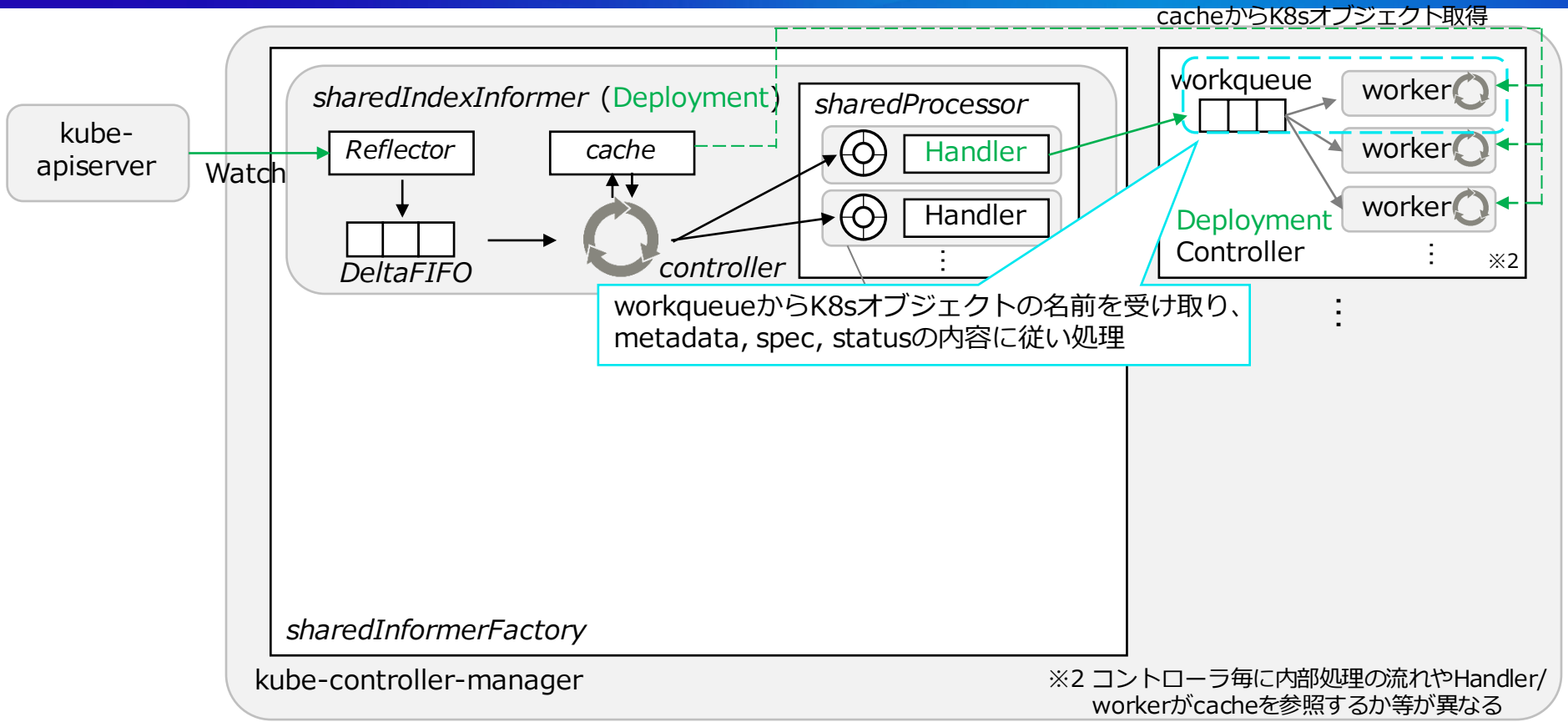
Podに紐づいたReplicaSetの情報を取得

Podと紐づいたReplicaSetの状態をcacheから取得

K8sオブジェクトを一意に識別する文字列(Key)<*namespace*>/<*name*> を作成

ReplicaSetコントローラのworkqueueにKeyをエンキュー

コントローラについて

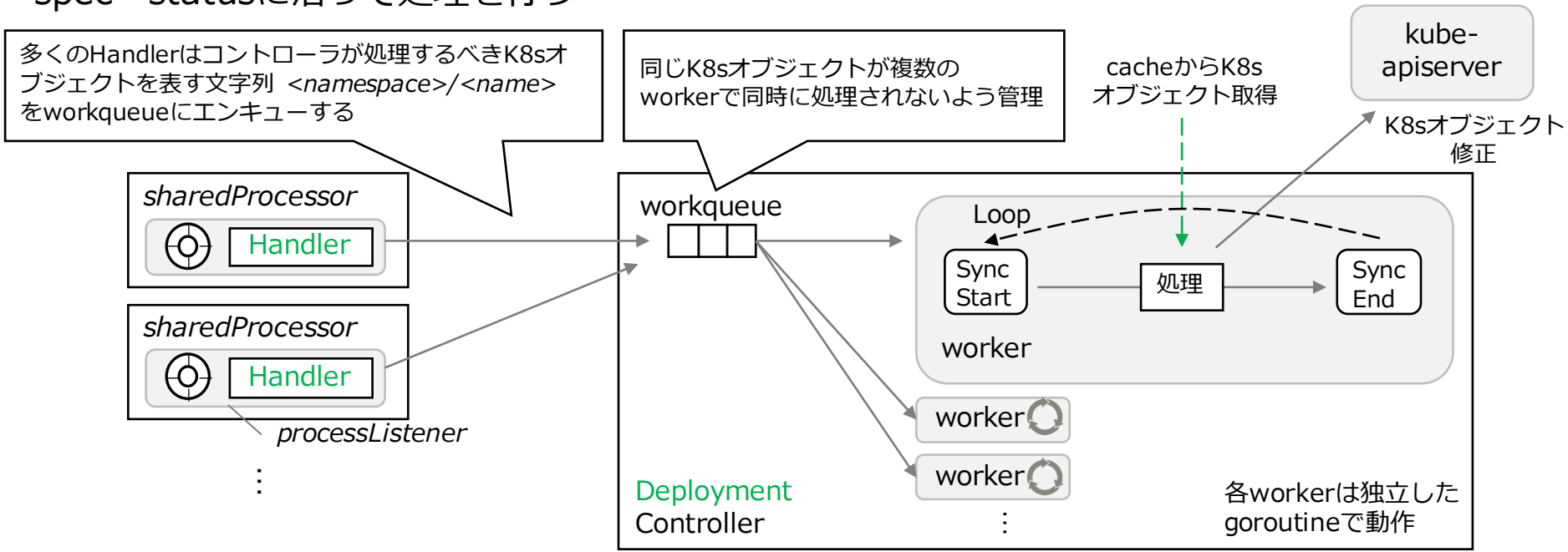


※斜体はGoのType名

コントローラの役割

FUJITSU

○ processListenerから受け取ったトリガーを契機(※)に、K8sオブジェクトのmetadata・spec・statusに沿って処理を行う



※ コントローラ毎にコントローラ内部処理の流れやHandler/workerのふるまいが異なる

コントローラのコードについて

FUJITSU

*1 cmd/kube-controller-manager/controller-manager.go
*2 cmd/kube-controller-manager/app/controllermanager.go

○ コントローラ作成契機

main() (*1)
- cobra Commandを作成し実行

呼出

Run(…) (*2)
- flag・configの適用
- コントローラの作成・実行

ServiceAccountTokentコントローラ生成関数

```
func Run(…) error {                                                                    (*2)
    …
    saTokenControllerInitFunc := serviceAccountTokenControllerStarter(…)
    …
    run := func(…, startSATokenController InitFunc, initializersFunc ControllerInitializersFunc) {
        …
        controllerInitializers := initializersFunc(controllerContext.LoopMode)
        if err := StartControllers(…, startSATokenController, controllerInitializers, …); err != nil {
            klog.Fatalf("error starting controllers: %v", err)
        }
        controllerContext.InformerFactory.Start(stopCh)
        controllerContext.ObjectOrMetadataInformerFactory.Start(stopCh)
        close(controllerContext.InformersStarted)
        select {}
    }
    …
    if !c.ComponentConfig.Generic.LeaderElection.LeaderElect {
        run(context.TODO(), saTokenControllerInitFunc, NewControllerInitializers)
        panic("unreachable")
    }
```

コントローラ生成関数を実行し
コントローラ作成・スタート

各種コントローラ生成関数

HA無しController Managerの処理実行
(HA有の場合はさらに下の箇所で処理実行)

コントローラのコードについて

FUJITSU

*1 cmd/kube-controller-manager/app/controllermanager.go

*2 cmd/kube-controller-manager/app/apps.go

○ 各コントローラの作成・実行関数

(*1)

```
func NewControllerInitializers(loopMode ControllerLoopMode) map[string]InitFunc {
    controllers := map[string]InitFunc{}
    controllers["endpoint"] = startEndpointController
    controllers["endpointslice"] = startEndpointSliceController
    controllers["endpointslicemirroring"] = startEndpointSliceMirroringController
    controllers["replicationcontroller"] = startReplicationController
    …
    controllers["deployment"] = startDeploymentController
    …
```

ServiceAccountTokenコントローラ以外のコントローラの作成・実行関数はNewControllerInitializers内にある

```
func startDeploymentController(…) (controller.Interface, bool, error) {
    dc, err := deployment.NewDeploymentController(
        controllerContext.InformerFactory.Apps().V1().Deployments(),
        controllerContext.InformerFactory.Apps().V1().ReplicaSets(),
        controllerContext.InformerFactory.Core().V1().Pods(),
        controllerContext.ClientBuilder.ClientOrDie("deployment-controller"),
    )
    if err != nil {
        return nil, true, fmt.Errorf("error creating Deployment controller: %v", err)
    }
    go dc.Run(ctx, int(controllerContext.ComponentConfig.DeploymentController.ConcurrentDeploymentSyncs))
    return nil, true, nil
}
```

コントローラの作成

コントローラの実行

(*2)

コントローラのコードについて

*1 pkg/controller/deployment/deployment_controller.go
*2 staging/src/k8s.io/client-go/util/workqueue/queue.go

○ Deploymentコントローラの作成

(*1)

```
func NewDeploymentController(dInformer appsinformers.DeploymentInformer, …) (*DeploymentController, error) {
    …
    dc := &DeploymentController{
        client:        client,
        eventRecorder: eventBroadcaster.NewRecorder(scheme.Scheme, v1.EventSource{Component: "deployment-controller"}),
        queue:         workqueue.NewNamedRateLimitingQueue(…),
    }
    …
    dInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
        AddFunc:    dc.addDeployment,
        UpdateFunc: dc.updateDeployment,
        DeleteFunc: dc.deleteDeployment,
    })
    …
```

workqueueの作成

sharedIndexInformerへのHandler登録

○ workqueueについて

```
type Type struct {
    queue []t        ←①
    dirty set        ←②
    processing set   ←③
    …
}
```

(*2)

workqueueの中ではTypeが用いられている

①コントローラで処理する順番を保持
②コントローラで再度処理しなくてはならない、現在処理中のK8sオブジェクトのKeyを保持
③コントローラで処理中のK8sオブジェクトのKeyを保持

コントローラのコードについて

FUJITSU

*1 pkg/controller/deployment/deployment_controller.go

○ Deploymentコントローラの内部処理

```
func (dc *DeploymentController) Run(…, workers int) {
    …
    for i := 0; i < workers; i++ {
        go wait.UntilWithContext(ctx, dc.worker, …)
    }
    …
}
…
func (dc *DeploymentController) worker(ctx context.Context) {
    for dc.processNextWorkItem(ctx) {
    }
}
```

(*1)

独立したgoroutineでworker実行

```
func (dc *DeploymentController) processNextWorkItem(…) bool {
    key, quit := dc.queue.Get()
    …
    err := dc.syncHandler(ctx, key.(string))
    dc.handleErr(err, key)

    return true
}
```

(*1)

処理でエラーが出たらworkqueueにリキュー

workqueueからデキューしたKey *<namespace>/<name>* を処理メソッドに渡す

```
func (dc *DeploymentController) syncDeployment(…, key string) error {
    namespace, name, err := cache.SplitMetaNamespaceKey(key)
    …
    deployment, err := dc.dLister.Deployments(namespace).Get(name)
    …
    d := deployment.DeepCopy()
    …
    everything := metav1.LabelSelector{}
    if reflect.DeepEqual(d.Spec.Selector, &everything) {
        dc.eventRecorder.Eventf(…
        if d.Status.ObservedGeneration < d.Generation {
            d.Status.ObservedGeneration = d.Generation
            dc.client.AppsV1().Deployments(d.Namespace).UpdateStatus(ctx, d, metav1.UpdateOptions{})
        }
        …
```

(*1)

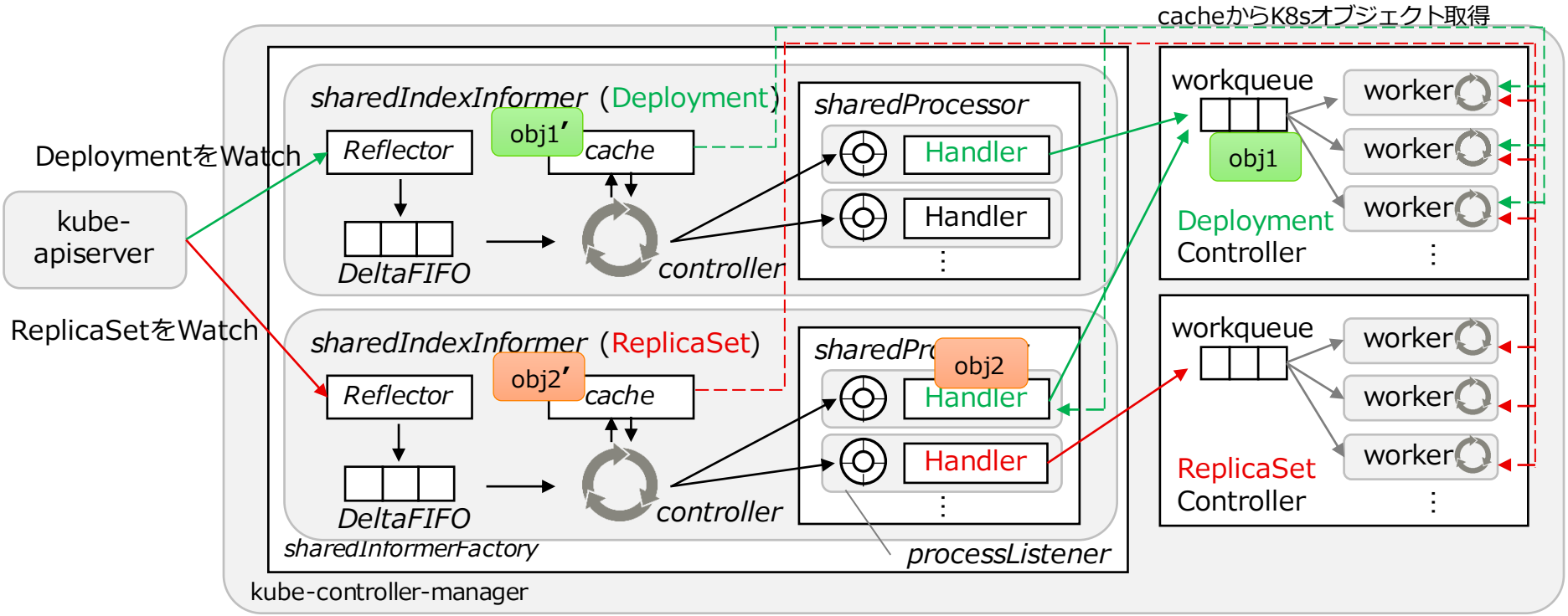
Keyから処理するDeploymentオブジェクトのNamspace・Nameを取得

cacheからDeploymentオブジェクトを取得

DeploymentオブジェクトのStatusの更新

kube-controller-manager内の非同期性について

- コントローラ処理の契機となったK8sオブジェクトとキャッシュ内部のオブジェクトのずれ
  - キャッシュ参照時に更新されたオブジェクトを見たり、契機となったオブジェクトが更新されている可能性がある



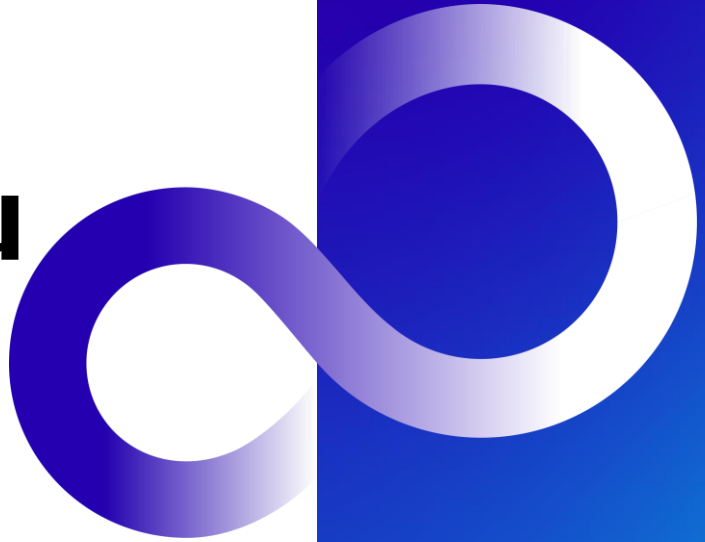
各コントローラのコードについて

FUJITSU

kube-controller-managerが管理する各コントローラの内部構造はコントローラ毎に異なる

- workqueue
  - エンキュー契機
    - K8sオブジェクトの変更をReflectorが通知し、Handlerが処理するK8sオブジェクトのKeyをエンキューする
    - 一定時間ごとにkube-apiserverからK8sオブジェクトを取得し、処理するオブジェクトのKeyをエンキュー
    - workqueueが無く、コントローラ内部で一定時間ごとにkube-apiserverからオブジェクトを取得
  - workqueueの数・実装
    - 一つのコントローラ内に複数種類のworkqueueがある
    - GoチャンネルやGo mapを用いてworkqueueを実装している
    - *<namespace>/<name>* の文字列（Key）ではなく、K8sオブジェクト自体をworkqueueにエンキュー
- コントローラの処理
  - cacheだけでなくkube-spiserverからK8sオブジェクトを取得し、metadata・spec・statusに従って処理
  - 複数個の種類の異なる処理フローが別々のgoroutineで動作している
  - processListenerを動的に作成し、sharedProcessorに登録

Thank you



FUJITSU