

HEAD 方法。

该接口的基于 Jersey 框架的实现类如下所示：

```
package com.hp.k8s.apiclient.imp;

import javax.ws.rs.core.MediaType;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import com.hp.k8s.apiclient.RestfulClient;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.config.DefaultClientConfig;
import com.sun.jersey.client.urlconnection.URLConnectionClientHandler;

public class JerseyRestfulClient implements RestfulClient {
    private static final Logger LOG = LogManager.getLogger(RestfulClient.
class.getName());
    private static final String METHOD_PATCH = "PATCH";

    private String _baseUrl = null;
    Client _client = null;

    public JerseyRestfulClient(String baseUrl) {
        DefaultClientConfig config = new DefaultClientConfig();
        config.getProperties().put(URLConnectionClientHandler.PROPERTY_HTTP_
URL_CONNECTION_SET_METHOD_WORKAROUND, true);
        _client = Client.create(config);

        this._baseUrl = baseUrl;
    }

    @Override
    public String get(Params params) {
        WebResource resource = _client.resource(_baseUrl + params.buildPath());
        String response = resource.accept(MediaType.APPLICATION_JSON_TYPE).
get(String.class);
        LOG.info("Get one resource:\n" + response);

        return response;
    }

    @Override
    public String list(Params params) {
        WebResource resource = _client.resource(_baseUrl + params.buildPath());
```

Kubernetes 权威指南：从 Docker 到 Kubernetes 实践全接触（第 2 版）

```
        LOG.info("URL: " + _baseUrl + params.buildPath());
        String response = resource.accept(MediaType.APPLICATION_JSON_TYPE).
get(String.class);

        return response;
    }

    @Override
    public String create(Params params) {
        WebResource resource = _client.resource(_baseUrl + params.buildPath());
        LOG.info("URL: " + _baseUrl + params.buildPath());
        LOG.info("Create resource: " + params.getJson());
        String response = (null == params.getJson())
            ? resource.accept(MediaType.APPLICATION_JSON).post(String.class)
            : resource.type(MediaType.APPLICATION_JSON).accept(MediaType.
APPLICATION_JSON).post(String.class,
            params.getJson());

        return response;
    }

    @Override
    public String delete(Params params) {
        WebResource resource = _client.resource(_baseUrl + params.buildPath());
        String response = resource.accept(MediaType.APPLICATION_JSON_TYPE).
delete(String.class);
        LOG.info("Delete resource " + params.getResourceType().getType() + "/"
+ params.getName() + " result:\n"
            + response);

        return response;
    }

    @Override
    public String update(Params params) {
        return updateWithMediaType(params, MediaType.APPLICATION_JSON);
    }

    @Override
    public String updateWithMediaType(Params params, String mediaType) {
        WebResource resource = _client.resource(_baseUrl + params.buildPath());
        LOG.info("URL: " + _baseUrl + params.buildPath());
        LOG.info("Patch resource: " + params.getJson());
        String response = resource.type(mediaType).accept(MediaType.APPLICATION_
JSON_TYPE).method(METHOD_PATCH, String.class,
            params.getJson());
        LOG.info("Update resource " + params.buildPath() + " result:\n" +
```

```

response);

    return response;
}

@Override
public String replace(Params params) {
    WebResource resource = _client.resource(_baseUrl + params.buildPath());
    LOG.info("URL: " + _baseUrl + params.buildPath());
    LOG.info("Replace resource: " + params.getJson());
    String response = resource.type(MediaType.APPLICATION_JSON_TYPE).accept(
(MediaType.APPLICATION_JSON_TYPE)
        .put(String.class, params.getJson()));
    LOG.info("Replace resource " + params.buildPath() + " result:\n" +
response);

    return response;
}

@Override
public String options(Params params) {
    WebResource resource = _client.resource(_baseUrl + params.buildPath());
    String response = resource.type(MediaType.APPLICATION_JSON_TYPE).accept(
(MediaType.TEXT_PLAIN_TYPE)
        .options(String.class));
    LOG.info("Get options for resource " + params.getResourceType().getType()
+ "/" + params.getName()
        + " result:\n" + response);

    return response;
}

@Override
public String head(Params params) {
    WebResource resource = _client.resource(_baseUrl + params.buildPath());
    String response = resource.accept(MediaType.TEXT_PLAIN_TYPE).head().
getResponseStatus().toString();
    LOG.info("Get head for resource " + params.getResourceType().getType() +
"/" + params.getName() + " result:\n"
        + response);

    return response;
}

@Override
public void close() {
    _client.destroy();
}

```

```
}  
  
}
```

该对象中包含如下代码：

```
config.getProperties().put(URLConnectionClientHandler.PROPERTY_HTTP_URL_CONNECTION_SET_METHOD_WORKAROUND, true);
```

该段代码的作用是使 Jersey 客户端能够支持除标准 REST 方法外的方法，比如 PATCH 方法。该段代码能访问除 watcher 外的所有 Kubernetes API 接口，在后续的章节中我们会举例说明如何访问 Kubernetes API。

4.3.2 Fabric8

Fabric8 包含多款工具包，Kubernetes Client 只是其中之一，也是 Kubernetes 官网中提到的 Java Client API 之一。本例子代码涉及的 Jar 包如图 4.7 所示。


 dnsjava-2.1.7.jar	2015/8/31 14:23	Executable Jar File	301 KB
 fabric8-utils-2.2.22.jar	2015/8/31 14:23	Executable Jar File	134 KB
 jackson-annotations-2.6.0.jar	2015/8/31 16:27	Executable Jar File	46 KB
 jackson-core-2.6.1.jar	2015/8/31 16:28	Executable Jar File	253 KB
 jackson-databind-2.6.1.jar	2015/8/31 15:56	Executable Jar File	1,140 KB
 jackson-dataformat-yaml-2.6.1.jar	2015/8/31 15:56	Executable Jar File	313 KB
 jackson-module-jaxb-annotations-2.6.0.jar	2015/8/31 16:24	Executable Jar File	32 KB
 json-20141113.jar	2015/8/31 14:23	Executable Jar File	64 KB
 kubernetes-api-2.2.22.jar	2015/8/31 14:22	Executable Jar File	72 KB
 kubernetes-client-1.3.8.jar	2015/8/31 15:37	Executable Jar File	2,262 KB
 kubernetes-model-1.0.12.jar	2015/8/31 15:56	Executable Jar File	2,308 KB
 log4j-api-2.3.jar	2015/8/31 16:18	Executable Jar File	133 KB
 log4j-core-2.3.jar	2015/8/31 15:56	Executable Jar File	808 KB
 log4j-slf4j-impl-2.3.jar	2015/8/31 15:56	Executable Jar File	23 KB
 oauth-20100527.jar	2015/8/31 15:56	Executable Jar File	44 KB
 openshift-client-1.3.2.jar	2015/8/31 14:23	Executable Jar File	24 KB
 slf4j-api-1.7.12.jar	2015/8/31 15:56	Executable Jar File	32 KB
 sundr-annotations-0.0.25.jar	2015/8/31 15:56	Executable Jar File	146 KB
 validation-api-1.1.0.Final.jar	2015/8/31 14:23	Executable Jar File	63 KB

图 4.7 例子代码涉及的 Jar 包

因为该工具包已经对访问 Kubernetes API 客户端做了较好的封装，因此其访问代码比较简单，其具体的访问过程会在后续的章节举例说明。

Fabric 8 的 Kubernetes API 客户端工具包只能访问 Node、Service、Pod、Endpoints、Events、Namespace、PersistentVolumeclaims、PersistentVolume、ReplicationController、ResourceQuota、Secret 和 ServiceAccount 这几个资源类型，不能使用 OPTIONS 和 HEAD 方法访问资源，且不能以代理方式访问资源，但其对以 watcher 方式访问资源做了很好的支持。

4.3.3 使用说明

首先, 举例说明对 API 资源的基本访问, 也就是对资源的增、删、改、查, 以及替换资源的 `status`。其中会单独对 Node 和 Pod 的特殊接口做举例说明。表 4.3 列出了各资源对象的基本 API 接口。

表 4.3 各资源对象的基本 API 接口

资源类型	方法	URL Path	说明	备注
NODES	GET	/api/v1/nodes	获取 Node 列表	
	POST	/api/v1/nodes	创建一个 Node 对象	
	DELETE	/api/v1/nodes/{name}	删除一个 Node 对象	
	GET	/api/v1/nodes/{name}	获取一个 Node 对象	
	PATCH	/api/v1/nodes/{name}	部分更新一个 Node 对象	
	PUT	/api/v1/nodes/{name}	替换一个 Node 对象	
NAMESPACES	GET	/api/v1/namespaces	获取 Namespace 列表	
	POST	/api/v1/namespaces	创建一个 Namespace 对象	
	DELETE	/api/v1/namespaces/{name}	删除一个 Namespace 对象	
	GET	/api/v1/namespaces/{name}	获取一个 Namespace 对象	
	PATCH	/api/v1/namespaces/{name}	部分更新一个 Namespace 对象	
	PUT	/api/v1/namespaces/{name}	替换一个 Namespace 对象	
	PUT	/api/v1/namespaces/{name}/finalize	替换一个 Namespace 对象的最终方案对象	在 Fabric8 中没有实现
	PUT	/api/v1/namespaces/{name}/status	替换一个 Namespace 对象的状态	在 Fabric8 中没有实现
SERVICES	GET	/api/v1/services	获取 Service 列表	
	POST	/api/v1/services	创建一个 Service 对象	
	GET	/api/v1/namespaces/{namespace}/services	获取某个 Namespace 下的 Service 列表	
	POST	/api/v1/namespaces/{namespace}/services	在某个 Namespace 下创建列表	
	DELETE	/api/v1/namespaces/{namespace}/services/{name}	删除某个 Namespace 下的一个 Service 对象	
	GET	/api/v1/namespaces/{namespace}/services/{name}	获取某个 Namespace 下的一个 Service 对象	
	PATCH	/api/v1/namespaces/{namespace}/services/{name}	部分更新某个 Namespace 下的一个 Service 对象	
	PUT	/api/v1/namespaces/{namespace}/services/{name}	替换某个 Namespace 下的一个 Service 对象	

续表

资源类型	方法	URL Path	说明	备注
REPLICATIONCONTROLLERS	GET	/api/v1/replicationcontrollers	获取 RC 列表	
	POST	/api/v1/replicationcontrollers	创建一个 RC 对象	
	GET	/api/v1/namespaces/{namespace}/replicationcontrollers	获取某个 Namespace 下的 RC 列表	
	POST	/api/v1/namespaces/{namespace}/replicationcontrollers	在某个 Namespace 下创建一个 RC 对象	
	DELETE	/api/v1/namespaces/{namespace}/replicationcontrollers/{name}	删除某个 Namespace 下的 RC 对象	
	GET	/api/v1/namespaces/{namespace}/replicationcontrollers/{name}	获取某个 Namespace 下的 RC 对象	
	PATCH	/api/v1/namespaces/{namespace}/replicationcontrollers/{name}	部分更新某个 Namespace 下的 RC 对象	
	PUT	/api/v1/namespaces/{namespace}/replicationcontrollers/{name}	替换某个 Namespace 下的 RC 对象	
PODS	GET	/api/v1/pods	获取一个 Pod 列表	
	POST	/api/v1/pods	创建一个 Pod 对象	
	GET	/api/v1/namespaces/{namespace}/pods	获取某个 Namespace 下的 Pod 列表	
	POST	/api/v1/namespaces/{namespace}/pods	在某个 Namespace 下创建一个 Pod 对象	
	DELETE	/api/v1/namespaces/{namespace}/pods/{name}	删除某个 Namespace 下的一个 Pod 对象	
	GET	/api/v1/namespaces/{namespace}/pods/{name}	获取某个 Namespace 下的一个 Pod 对象	
	PATCH	/api/v1/namespaces/{namespace}/pods/{name}	部分更新某个 Namespace 下的一个 Pod 对象	
	PUT	/api/v1/namespaces/{namespace}/pods/{name}	替换某个 Namespace 下的一个 Pod 对象	
	PUT	/api/v1/namespaces/{namespace}/pods/{name}/status	替换某个 Namespace 下的一个 Pod 对象状态	在 Fabric8 中没有实现
	POST	/api/v1/namespaces/{namespace}/pods/{name}/binding	创建某个 Namespace 下的一个 Pod 对象的 Binding	在 Fabric8 中没有实现
	GET	/api/v1/namespaces/{namespace}/pods/{name}/exec	连接到某个 Namespace 下的一个 Pod 对象，并执行 exec	在 Fabric8 中没有实现
	POST	/api/v1/namespaces/{namespace}/pods/{name}/exec	连接到某个 Namespace 下的一个 Pod 对象，并执行 exec	在 Fabric8 中没有实现

续表

资源类型	方法	URL Path	说明	备注
	GET	/api/v1/namespaces/{namespace}/pods/{name}/log	连接到某个 Namespace 下的一个 Pod 对象，并获取 log 日志信息	在 Fabric8 中没有实现
	GET	/api/v1/namespaces/{namespace}/pods/{name}/portforward	连接到某个 Namespace 下的一个 Pod 对象，并实现端口转发	在 Fabric8 中没有实现
	POST	/api/v1/namespaces/{namespace}/pods/{name}/portforward	连接到某个 Namespace 下的一个 Pod 对象，并实现端口转发	在 Fabric8 中没有实现
BINDINGS	POST	/api/v1/bindings	创建一个 Binding 对象	
	POST	/api/v1/namespaces/{namespace}/bindings	在某个 Namespace 下创建一个 Binding 对象	
ENDPOINTS	GET	/api/v1/endpoints	获取 Endpoint 列表	
	POST	/api/v1/endpoints	创建一个 Endpoint 对象	
	GET	/api/v1/namespaces/{namespace}/endpoints	获取某个 Namespace 下的 Endpoint 对象列表	
	POST	/api/v1/namespaces/{namespace}/endpoints	在某个 Namespace 下创建一个 Endpoint 对象	
	DELETE	/api/v1/namespaces/{namespace}/endpoints/{name}	删除某个 Namespace 下的 Endpoint 对象	
	GET	/api/v1/namespaces/{namespace}/endpoints/{name}	获取某个 Namespace 下的 Endpoint 对象	
	PATCH	/api/v1/namespaces/{namespace}/endpoints/{name}	部分更新某个 Namespace 下的 Endpoint 对象	
	PUT	/api/v1/namespaces/{namespace}/endpoints/{name}	替换某个 Namespace 下的 Endpoint 对象	
SERVICEACCOUNTS	GET	/api/v1/serviceaccounts	获取 Serviceaccount 列表	
	POST	/api/v1/serviceaccounts	创建一个 Serviceaccount 对象	
	GET	/api/v1/namespaces/{namespace}/serviceaccounts	获取某个 Namespace 下的 Serviceaccount 对象列表	
	POST	/api/v1/namespaces/{namespace}/serviceaccounts	在某个 Namespace 下创建一个 Serviceaccount 对象	
	DELETE	/api/v1/namespaces/{namespace}/serviceaccounts/{name}	删除某个 Namespace 下的一个 Serviceaccount 对象	
	GET	/api/v1/namespaces/{namespace}/serviceaccounts/{name}	获取某个 Namespace 下的一个 Serviceaccount 对象	

续表

资源类型	方法	URL Path	说明	备注
	PATCH	/api/v1/namespaces/{namespace}/serviceaccounts/{name}	部分更新某个 Namespace 下的一个 Serviceaccount 对象	
	PUT	/api/v1/namespaces/{namespace}/serviceaccounts/{name}	替换某个 Namespace 下的一个 Serviceaccount 对象	
SECRETS	GET	/api/v1/secrets	获取 Secret 列表	
	POST	/api/v1/secrets	创建一个 Secret 对象	
	GET	/api/v1/namespaces/{namespace}/secrets	获取某个 Namespace 下的 Secret 列表	
	POST	/api/v1/namespaces/{namespace}/secrets	在某个 Namespace 下创建一个 Secret 对象	
	DELETE	/api/v1/namespaces/{namespace}/secrets/{name}	删除某个 Namespace 下的一个 Secret 对象	
	GET	/api/v1/namespaces/{namespace}/secrets/{name}	获取某个 Namespace 下的一个 Secret 对象	
	PATCH	/api/v1/namespaces/{namespace}/secrets/{name}	部分更新某个 Namespace 下的一个 Secret 对象	
	PUT	/api/v1/namespaces/{namespace}/secrets/{name}	替换某个 Namespace 下的一个 Secret 对象	
EVENTS	GET	/api/v1/events	获取 Event 列表	
	POST	/api/v1/events	创建一个 Event 对象	
	GET	/api/v1/namespaces/{namespace}/events	获取某个 Namespace 下的 Event 列表	
	POST	/api/v1/namespaces/{namespace}/events	在某个 Namespace 下创建一个 Event 对象	
	DELETE	/api/v1/namespaces/{namespace}/events/{name}	删除某个 Namespace 下的一个 Event 对象	
	GET	/api/v1/namespaces/{namespace}/events/{name}	获取某个 Namespace 下的一个 Event 对象	
	PATCH	/api/v1/namespaces/{namespace}/events/{name}	部分更新某个 Namespace 下的一个 Event 对象	
	PUT	/api/v1/namespaces/{namespace}/events/{name}	替换某个 Namespace 下的一个 Event 对象	
COMPONENTSTATUS	GET	/api/v1/componentstatuses	获取 ComponentStatus 列表	
	GET	/api/v1/namespaces/{namespace}/componentstatuses	获取某个 Namespace 下的 Component Status 列表	

续表

资源类型	方法	URL Path	说明	备注
	GET	/api/v1/namespaces/{namespace}/componentstatuses/{name}	获取某个 Namespace 下的一个 ComponentStatus 对象	
LIMITRANGES	GET	/api/v1/limitranges	获取 LimitRange 列表	
	POST	/api/v1/limitranges	创建一个 LimitRange 对象	
	GET	/api/v1/namespaces/{namespace}/limitranges	获取某个 Namespace 下的 LimitRange 列表	
	POST	/api/v1/namespaces/{namespace}/limitranges	在某个 Namespace 下创建一个 LimitRange 对象	
	DELETE	/api/v1/namespaces/{namespace}/limitranges/{name}	删除某个 Namespace 下的一个 LimitRange 对象	
	GET	/api/v1/namespaces/{namespace}/limitranges/{name}	获取某个 Namespace 下的一个 LimitRange 对象	
	PATCH	/api/v1/namespaces/{namespace}/limitranges/{name}	部分更新某个 Namespace 下的一个 LimitRange 对象	
	PUT	/api/v1/namespaces/{namespace}/limitranges/{name}	替换某个 Namespace 下的一个 LimitRange 对象	
RESOURCEQUOTAS	GET	/api/v1/resourcequotas	获取 ResourceQuota 列表	
	POST	/api/v1/resourcequotas	创建一个 ResourceQuota 对象	
	GET	/api/v1/namespaces/{namespace}/resourcequotas	获取某个 Namespace 下的 Resource Quota 列表	
	POST	/api/v1/namespaces/{namespace}/resourcequotas	在某个 Namespace 下创建一个 Resource Quota 对象	
	DELETE	/api/v1/namespaces/{namespace}/resourcequotas/{name}	删除某个 Namespace 下的一个 Resource Quota 对象	
	GET	/api/v1/namespaces/{namespace}/resourcequotas/{name}	获取某个 Namespace 下的一个 Resource Quota 对象	
	PATCH	/api/v1/namespaces/{namespace}/resourcequotas/{name}	部分更新某个 Namespace 下的一个 Resource Quota 对象	
	PUT	/api/v1/namespaces/{namespace}/resourcequotas/{name}	替换某个 Namespace 下的一个 Resource Quota 对象	
	PUT	/api/v1/namespaces/{namespace}/resourcequotas/{name}/status	替换某个 Namespace 下的一个 Resource Quota 对象状态	在 Fabric8 中没有实现

续表

资源类型	方法	URL Path	说明	备注
PODTEMPLATES	GET	/api/v1/podtemplates	获取 PodTemplate 列表	
	POST	/api/v1/podtemplates	创建一个 PodTemplate 对象	
	GET	/api/v1/namespaces/{namespace}/podtemplates	获取某个 Namespace 下的 PodTemplate 列表	
	POST	/api/v1/namespaces/{namespace}/podtemplates	在某个 Namespace 下创建一个 PodTemplate 对象	
	DELETE	/api/v1/namespaces/{namespace}/podtemplates/{name}	删除某个 Namespace 下的一个 PodTemplate 对象	
	GET	/api/v1/namespaces/{namespace}/podtemplates/{name}	获取某个 Namespace 下的一个 PodTemplate 对象	
	PATCH	/api/v1/namespaces/{namespace}/podtemplates/{name}	部分更新某个 Namespace 下的一个 PodTemplate 对象	
	PUT	/api/v1/namespaces/{namespace}/podtemplates/{name}	替换某个 Namespace 下的一个 PodTemplate 对象	
PERSISTENTVOLUMES	GET	/api/v1/persistentvolumes	获取 PersistentVolume 列表	
	POST	/api/v1/persistentvolumes	创建一个 PersistentVolume 对象	
	DELETE	/api/v1/persistentvolumes/{name}	删除一个 PersistentVolume 对象	
	GET	/api/v1/persistentvolumes/{name}	获取一个 PersistentVolume 对象	
	PATCH	/api/v1/persistentvolumes/{name}	部分更新一个 PersistentVolume 对象	
	PUT	/api/v1/persistentvolumes/{name}	替换一个 PersistentVolume 对象	
	PUT	/api/v1/persistentvolumes/{name}/status	替换一个 PersistentVolume 对象状态	在 Fabric8 中没有实现
PERSISTENTVOLUMECLAIMS	GET	/api/v1/persistentvolumeclaims	获取 PersistentVolumeClaim 列表	
	POST	/api/v1/persistentvolumeclaims	创建一个 PersistentVolumeClaim 对象	
	GET	/api/v1/namespaces/{namespace}/persistentvolumeclaims	获取某个 Namespace 下的 PersistentVolumeClaim 列表	
	POST	/api/v1/namespaces/{namespace}/persistentvolumeclaims	在某个 Namespace 下创建一个 PersistentVolumeClaim 对象	
	DELETE	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}	删除某个 Namespace 下的一个 PersistentVolumeClaim 对象	
	GET	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}	获取某个 Namespace 下的一个 PersistentVolumeClaim 对象	
	PATCH	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}	部分更新某个 Namespace 下的一个 PersistentVolumeClaim 对象	

续表

资源类型	方法	URL Path	说明	备注
	PUT	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}	替换某个 Namespace 下的一个 Persistent VolumeClaim 对象	
	PUT	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}/status	替换某个 Namespace 下的一个 Persistent VolumeClaim 对象状态	在 Fabric8 中没有实现

首先，举例说明如何通过 API 接口来创建资源对象。我们需要创建访问 API Server 的客户端，基于 Jersey 框架的代码如下：

```
RestfulClient _restfulClient = new JerseyRestfulClient("http://192.168.1.128:8080/api/v1");
```

其中，http://192.168.1.128:8080 为 API Server 的地址。基于 Fabric8 框架的代码如下：

```
Config _conf = new Config();
KubernetesClient _kube = new DefaultKubernetesClient("http://192.168.1.128: 8080");
```

分别通过上面的两个客户端创建 Namespace 资源对象，基于 Jersey 框架的代码如下：

```
private void testCreateNamespace() {
    Params params = new Params();
    params.setResourceType(ResourceType.NAMESPACES);
    params.setJson(Utills.getJson("namespace.json"));

    LOG.info("Result: " + _restfulClient.create(params));
}
```

其中，“namespace.json”为创建 Namespace 资源对象的 JSON 定义，代码如下：

```
{
  "kind": "Namespace",
  "apiVersion": "v1",
  "metadata": {
    "name": "ns-sample"
  }
}
```

基于 Fabric8 框架的代码如下：

```
private void testCreateNamespace() {
    Namespace ns = new Namespace();
    ns.setApiVersion(ApiVersion.V_1);
    ns.setKind("Namespace");
    ObjectMeta om = new ObjectMeta();
    om.setName("ns-fabric8");
    ns.setMetadata(om);

    _kube.namespaces().create(ns);
}
```

```
LOG.info(_kube.namespaces().list().getItems().size());
}
```

由于 Fabric8 框架对 Kubernetes API 对象做了很好的封装，对其中的大量对象都做了定义，所以用户可以通过其提供的资源对象去定义 Kubernetes API 对象，例如上面例子中的 Namespace 对象。Fabric8 框架中的 `kubernetes-model` 工具包用于 API 对象的封装。在上面的例子中，通过 Fabric8 框架提供的类创建了一个名为“`ns-fabric8`”的命名空间对象。

接下来我们会通过基于 Jeysey 框架的代码去创建两个 Pod 资源对象。在两个例子中，一个是在上面创建的“`ns-sample`”Namespace 中创建 Pod 资源对象，另一个是为后续创建“`cluster service`”而创建的 Pod 资源对象。由于基于 Fabric8 框架创建 Pod 资源对象的方法很简单，因此不再用 Fabric8 框架对上述两个例子做说明。通过基于 Jersey 框架创建这两个 Pod 资源对象的代码如下：

```
private void testCreatePod() {
    Params params = new Params();
    params.setResourceType(ResourceType.PODS);
    params.setJson(Utils.getJson("podInNs.json"));
    params.setNamespace("ns-sample");
    LOG.info("Result: " + _restfulClient.create(params));

    params.setJson(Utils.getJson("pod4ClusterService.json"));
    LOG.info("Result: " + _restfulClient.create(params));
}
```

其中，`podInNs.json` 和 `pod4ClusterService.json` 是创建两个 Pod 资源对象的定义。`podInNs.json` 文件的内容如下：

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "pod-sample-in-namespace",
    "namespace": "ns-sample"
  },
  "spec": {
    "containers": [{
      "name": "mycontainer",
      "image": "kubeguide/redis-master"
    }]
  }
}
```

`pod4ClusterService.json` 文件的内容如下：

```
{
  "kind": "Pod",
```

```

"apiVersion":"v1",
"metadata":{
  "name":"pod-sample-4-cluster-service",
  "namespace": "ns-sample",
  "labels":{
    "k8s-cs": "kube-cluster-service",
    "k8s-test": "kube-cluster-test",
    "k8s-sample-app": "kube-service-sample",
    "kkk": "bbb"
  }
},
"spec":{
  "containers":[{
    "name":"mycontainer",
    "image":"kubeguide/redis-master"
  }]
}
}

```

下面的例子代码用于获取 Pod 资源列表,其中第 1 部分代码用于获取所有的 Pod 资源对象,第 2、3 部分代码主要是列举如何使用标签选择 Pod 资源对象,最后一部分代码用于举例说明如何使用 field 选择 Pod 资源对象。代码如下:

```

private void testGetPodList() {
    Params params = new Params();
    params.setResourceType(ResourceType.PODS);
    LOG.info("Result: " + _restfulClient.list(params));

    Map<String, String> labels = new HashMap<String, String>();
    labels.put("k8s-cs", "kube-cluster-service");
    labels.put("k8s-sample-app", "kube-service-sample");
    params.setLabels(labels);
    LOG.info("Result: " + _restfulClient.list(params));
    params.setLabels(null);

    Map<String, List<String>> inLabels = new HashMap<String, List<String>>();
    List list = new ArrayList<String>();
    list.add("kube-cluster-service");
    list.add("kube-cluster");
    inLabels.put("k8s-cs", list);
    params.setInLabels(inLabels);
    LOG.info("Result: " + _restfulClient.list(params));
    params.setInLabels(null);

    Map<String, String> fields = new HashMap<String, String>();
    fields.put("metadata.name", "pod-sample-4-cluster-service");
    params.setNamespace("ns-sample");
}

```

```
params.setFields(fields);
LOG.info("Result: " + _restfulClient.list(params));
}
```

接下来的例子代码用于替换一个 Pod 对象，在通过 Kubernetes API 替换一个 Pod 资源对象时需要注意两点：

(1) 在替换该资源对象前，先从 API 中获取该资源对象的 JSON 对象，然后在该 JSON 对象的基础上修改需要替换的部分；

(2) 在 Kubernetes API 提供的接口中，PUT 方法（replace）只支持替换容器的 image 部分。

代码如下：

```
private void testReplacePod() {
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setJson(Utills.getJson("pod4Replace.json"));
    params.setResourceType(ResourceType.PODS);

    LOG.info("Result: " + _restfulClient.replace(params));
}
```

其中，pod4Replace.json 的内容如下：

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "pod-sample-in-namespace",
    "namespace": "ns-sample",
    "selfLink": "/api/v1/namespaces/ns-sample/pods/pod-sample-in-namespace",
    "uid": "084ff63e-59d3-11e5-8035-000c2921ba71",
    "resourceVersion": "45450",
    "creationTimestamp": "2015-09-13T04:51:01Z"
  },
  "spec": {
    "volumes": [
      {
        "name": "default-token-szoje",
        "secret": {
          "secretName": "default-token-szoje"
        }
      }
    ],
    "containers": [
      {
        "name": "mycontainer",
```

```

    "image": "centos",
    "resources": {},
    "volumeMounts": [
      {
        "name": "default-token-szoje",
        "readOnly": true,
        "mountPath": "/var/run/secrets/kubernetes.io/serviceaccount"
      }
    ],
    "terminationMessagePath": "/dev/termination-log",
    "imagePullPolicy": "IfNotPresent"
  }
],
"restartPolicy": "Always",
"dnsPolicy": "ClusterFirst",
"serviceAccountName": "default",
"serviceAccount": "default",
"nodeName": "192.168.1.129"
},
"status": {
  "phase": "Running",
  "conditions": [
    {
      "type": "Ready",
      "status": "True"
    }
  ]
},
"hostIP": "192.168.1.129",
"podIP": "10.1.10.66",
"startTime": "2015-09-11T15:17:28Z",
"containerStatuses": [
  {
    "name": "mycontainer",
    "state": {
      "running": {
        "startedAt": "2015-09-11T15:17:30Z"
      }
    },
    "lastState": {},
    "ready": true,
    "restartCount": 0,
    "image": "kubeguide/redis-master",
    "imageID":
"docker://5630952871a38cddffda9ec611f5978ab0933628fcd54cd7d7677ce6b17de33f",
    "containerID": "docker://7bf0d454c367418348711556e667fd1ef6a04d7153d
24bfcac2e2e06da634a9f"
  }
]
}

```

```
    ]  
  }  
}
```

接下来的两个例子实现了 4.2.4 节中提到的两种 Merge 方式：Merge Patch 和 Strategic Merge Patch。

第 1 种 Merge 方式的示例如下：

```
private void testUpdatePod1() {  
    Params params = new Params();  
    params.setNamespace("ns-sample");  
    params.setName("pod-sample-in-namespace");  
    params.setJson(Utils.getJson("pod4MergeJsonPatch.json"));  
    params.setResourceType(ResourceType.PODS);  
  
    LOG.info("Result: " + _restfulClient.updateWithMediaType(params,  
"application/merge-patch+json"));  
}
```

其中，pod4MergeJsonPatch.json 的内容如下：

```
{  
  "metadata":{  
    "labels":{  
      "k8s-cs": "kube-cluster-service",  
      "k8s-test": "kube-cluster-test",  
      "k8s-sa555mple-app": "kube-service-sample",  
      "kkk": "bbb4444"  
    }  
  }  
}
```

第 2 种 Merge 方式（Strategic Merge Patch）的示例如下：

```
private void testUpdatePod2() {  
    Params params = new Params();  
    params.setNamespace("ns-sample");  
    params.setName("pod-sample-in-namespace");  
    params.setJson(Utils.getJson("pod4StrategicMerge.json"));  
    params.setResourceType(ResourceType.PODS);  
  
    LOG.info("Result: " + _restfulClient.updateWithMediaType(params,  
"application/strategic-merge-patch+json"));  
}
```

其中，pod4StrategicMerge.json 的内容如下：

```
{  
  "spec":{
```



```

        "containers": [{
            "name": "mycontainer",
            "image": "centos",
            "patchStrategy": "merge",
            "patchMergeKey": "name"
        }]
    }
}

```

接下来实现了修改 Pod 资源对象的状态，代码如下：

```

private void testStatusPod() {
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setSubPath("/status");
    params.setJson(Utils.getJson("pod4Status.json"));
    params.setResourceType(ResourceType.PODS);

    _restfulClient.replace(params);
}

```

其中，pod4Status.json 的内容如下：

```

{
    "kind": "Pod",
    "apiVersion": "v1",
    "metadata": {
        "name": "pod-sample-in-namespace",
        "namespace": "ns-sample",
        "selfLink": "/api/v1/namespaces/ns-sample/pods/pod-sample-in-namespace",
        "uid": "ad1d803f-59ec-11e5-8035-000c2921ba71",
        "resourceVersion": "51640",
        "creationTimestamp": "2015-09-13T07:54:35Z"
    },
    "spec": {
        "volumes": [
            {
                "name": "default-token-szoje",
                "secret": {
                    "secretName": "default-token-szoje"
                }
            }
        ],
        "containers": [
            {
                "name": "mycontainer",
                "image": "kubeguide/redis-master",
                "resources": {},

```

```
    "volumeMounts": [
      {
        "name": "default-token-szoje",
        "readOnly": true,
        "mountPath": "/var/run/secrets/kubernetes.io/serviceaccount"
      }
    ],
    "terminationMessagePath": "/dev/termination-log",
    "imagePullPolicy": "IfNotPresent"
  }
],
"restartPolicy": "Always",
"dnsPolicy": "ClusterFirst",
"serviceAccountName": "default",
"serviceAccount": "default",
"nodeName": "192.168.1.129"
},
"status": {
  "phase": "Unknown",
  "conditions": [
    {
      "type": "Ready",
      "status": "false"
    }
  ],
  "hostIP": "192.168.1.129",
  "podIP": "10.1.10.79",
  "startTime": "2015-09-11T18:21:02Z",
  "containerStatuses": [
    {
      "name": "mycontainer",
      "state": {
        "running": {
          "startedAt": "2015-09-11T18:21:03Z"
        }
      },
      "lastState": {},
      "ready": true,
      "restartCount": 0,
      "image": "kubeguide/redis-master",
      "imageID": "docker://5630952871a38cddffda9ec611f5978ab0933628fcd54cd7d7677ce6b17de33f",
      "containerID": "docker://b0e2312643e9a4b59cf1ff5fb7a8468c5777180d5a8ea5f2f0c9dfddcf3f4cd2"
    }
  ]
}
```

```
}

```

接下来实现了查看 Pod 的 log 日志功能，代码如下：

```
private void testLogPod() {
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setSubPath("/log");
    params.setResourceType(ResourceType.PODS);

    _restfulClient.get(params);
}
```

下面通过 API 访问 Node 的多种接口，代码如下：

```
private void testPoxyNode() {
    Params params = new Params();
    params.setName("192.168.1.129");
    params.setSubPath("pods");
    params.setVisitProxy(true);
    params.setResourceType(ResourceType.NODES);
    _restfulClient.get(params);

    params = new Params();
    params.setName("192.168.1.129");
    params.setSubPath("stats");
    params.setVisitProxy(true);
    params.setResourceType(ResourceType.NODES);
    _restfulClient.get(params);

    params = new Params();
    params.setName("192.168.1.129");
    params.setSubPath("spec");
    params.setVisitProxy(true);
    params.setResourceType(ResourceType.NODES);
    _restfulClient.get(params);

    params = new Params();
    params.setName("192.168.1.129");
    params.setSubPath("run/ns-sample/pod/pod-sample-in-namespace");
    params.setVisitProxy(true);
    params.setResourceType(ResourceType.NODES);
    _restfulClient.get(params);

    params = new Params();
    params.setName("192.168.1.129");
    params.setSubPath("metrics");
    params.setVisitProxy(true);
}
```

```
params.setResourceType(ResourceType.NODES);
_restfulClient.get(params);
}
```

最后，举例说明如何通过 API 删除资源对象 pod，代码如下：

```
private void testDeletePod() {
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setResourceType(ResourceType.PODS);
    LOG.info("Result: " + _restfulClient.delete(params));
}
```

通过 API 接口除了能够对资源对象实现前面列出的基本操作外，还涉及两类特殊接口，一类是 WATCH，一类是 PROXY。这两类特殊接口所包含的接口如表 4.4 所示。

表 4.4 两类特殊接口所包含的接口

资源类型	类别	方法	URL Path	说明
NODES	WATCH	GET	/api/v1/watch/nodes	监听所有节点的变化
		GET	/api/v1/watch/nodes/{name}	监听单个节点的变化
	PROXY	DELETE	/api/v1/proxy/nodes/{name}/{path:*}	代理 DELETE 请求到节点的某个子目录
		GET	/api/v1/proxy/nodes/{name}/{path:*}	代理 GET 请求到节点的某个子目录
		HEAD	/api/v1/proxy/nodes/{name}/{path:*}	代理 HEAD 请求到节点的某个子目录
		OPTIONS	/api/v1/proxy/nodes/{name}/{path:*}	代理 OPTIONS 请求到节点的某个子目录
		POST	/api/v1/proxy/nodes/{name}/{path:*}	代理 POST 请求到节点的某个子目录
		PUT	/api/v1/proxy/nodes/{name}/{path:*}	代理 PUT 请求到节点的某个子目录
		DELETE	/api/v1/proxy/nodes/{name}	代理 DELETE 请求到节点
		GET	/api/v1/proxy/nodes/{name}	代理 GET 请求到节点
		HEAD	/api/v1/proxy/nodes/{name}	代理 HEAD 请求到节点
		OPTIONS	/api/v1/proxy/nodes/{name}	代理 OPTIONS 请求到节点
		POST	/api/v1/proxy/nodes/{name}	代理 POST 请求到节点
		PUT	/api/v1/proxy/nodes/{name}	代理 PUT 请求到节点
SERVICES	WATCH	GET	/api/v1/watch/services	监听所有 Service 的变化
		GET	/api/v1/watch/namespaces/{namespace}/services	监听某个 Namespace 下所有 Service 的变化

续表

资源类型	类别	方法	URL Path	说明
		GET	/api/v1/watch/namespaces/{namespace}/services/{name}	监听某个 Service 的变化
	PROXY	DELETE	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:*}	代理 DELETE 请求到 Service 的某个子目录
		GET	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:*}	代理 GET 请求到 Service 的某个子目录
		HEAD	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:*}	代理 HEAD 请求到 Service 的某个子目录
		OPTIONS	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:*}	代理 OPTIONS 请求到 Service 的某个子目录
		POST	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:*}	代理 POST 请求到 Service 的某个子目录
		PUT	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:*}	代理 PUT 请求到 Service 的某个子目录
		DELETE	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 DELETE 请求到 Service
		GET	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 GET 请求到 Service
		HEAD	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 HEAD 请求到 Service
		OPTIONS	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 OPTIONS 请求到 Service
		POST	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 POST 请求到 Service
		PUT	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 PUT 请求到 Service
REPLICATIONCONTROLLER	WATCH	GET	/api/v1/watch/replicationcontrollers	监听所有 RC 的变化
		GET	/api/v1/watch/namespaces/{namespace}/replicationcontrollers	监听某个 Namespace 下所有 RC 的变化
		GET	/api/v1/watch/namespaces/{namespace}/replicationcontrollers/{name}	监听某个 RC 的变化
PODS	WATCH	GET	/api/v1/watch/pods	监听所有 Pod 的变化
		GET	/api/v1/watch/namespaces/{namespace}/pods	监听某个 Namespace 下所有 Pod 的变化

续表

资源类型	类别	方法	URL Path	说明
	PROXY	GET	/api/v1/watch/namespaces/{namespace}/pods/{name}	监听某个 Pod 的变化
		DELETE	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:*}	代理 DELETE 请求到 Pod 的某个子目录
		GET	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:*}	代理 GET 请求到 Pod 的某个子目录
		HEAD	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:*}	代理 HEAD 请求到 Pod 的某个子目录
		OPTIONS	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:*}	代理 OPTIONS 请求到 Pod 的某个子目录
		POST	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:*}	代理 POST 请求到 Pod 的某个子目录
		PUT	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:*}	代理 PUT 请求到 Pod 的某个子目录
		DELETE	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 DELETE 请求到 Pod
		GET	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 GET 请求到 Pod
		HEAD	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 HEAD 请求到 Pod
		OPTIONS	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 OPTIONS 请求到 Pod
		POST	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 POST 请求到 Pod
		PUT	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 PUT 请求到 Pod
		DELETE	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:*}	代理 DELETE 请求到 Pod 的某个子目录
		GET	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:*}	代理 GET 请求到 Pod 的某个子目录
		HEAD	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:*}	代理 HEAD 请求到 Pod 的某个子目录
		OPTIONS	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:*}	代理 OPTIONS 请求到 Pod 的某个子目录
		POST	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:*}	代理 POST 请求到 Pod 的某个子目录

续表

资源类型	类别	方法	URL Path	说明
		PUT	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:~}	代理 PUT 请求到 Pod 的某个子目录
		DELETE	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 DELETE 请求到 Pod
		GET	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 GET 请求到 Pod
		HEAD	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 HEAD 请求到 Pod
		OPTIONS	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 OPTIONS 请求到 Pod
		POST	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 POST 请求到 Pod
		PUT	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 PUT 请求到 Pod
ENDPOINTS	WATCH	GET	/api/v1/watch/endpoints	监听所有 Endpoint 的变化
		GET	/api/v1/watch/namespaces/{namespace}/endpoints	监听某个 Namespace 下所有 Endpoint 的变化
		GET	/api/v1/watch/namespaces/{namespace}/endpoints/{name}	监听某个 Endpoint 的变化
SERVICEACCOUNT	WATCH	GET	/api/v1/watch/serviceaccounts	监听所有 ServiceAccount 的变化
		GET	/api/v1/watch/namespaces/{namespace}/serviceaccounts	监听某个 Namespace 下所有 ServiceAccount 的变化
		GET	/api/v1/watch/namespaces/{namespace}/serviceaccounts/{name}	监听某个 ServiceAccount 的变化
SECRET	WATCH	GET	/api/v1/watch/secrets	监听所有 Secret 的变化
		GET	/api/v1/watch/namespaces/{namespace}/secrets	监听某个 Namespace 下所有 Secret 的变化
		GET	/api/v1/watch/namespaces/{namespace}/secrets/{name}	监听某个 Secret 的变化
EVENTS	WATCH	GET	/api/v1/watch/events	监听所有 Event 的变化
		GET	/api/v1/watch/namespaces/{namespace}/events	监听某个 Namespace 下所有 Event 的变化

续表

资源类型	类别	方法	URL Path	说明
		GET	/api/v1/watch/namespaces/{namespace}/events/{name}	监听某个 Event 的变化
LIMITRANGES	WATCH	GET	/api/v1/watch/limitranges	监听所有 Event 的变化
		GET	/api/v1/watch/namespaces/{namespace}/limitranges	监听某个 Namespace 下所有 Event 的变化
		GET	/api/v1/watch/namespaces/{namespace}/limitranges/{name}	监听某个 Event 的变化
RESOURCEQUOTAS	WATCH	GET	/api/v1/watch/resourcequotas	监听所有 ResourceQuota 的变化
		GET	/api/v1/watch/namespaces/{namespace}/resourcequotas	监听某个 Namespace 下所有 ResourceQuota 的变化
		GET	/api/v1/watch/namespaces/{namespace}/resourcequotas/{name}	监听某个 ResourceQuota 的变化
PODTEMPLATES	WATCH	GET	/api/v1/watch/podtemplates	监听所有 PodTemplate 的变化
		GET	/api/v1/watch/namespaces/{namespace}/podtemplates	监听某个 Namespace 下所有 PodTemplate 的变化
		GET	/api/v1/watch/namespaces/{namespace}/podtemplates/{name}	监听某个 PodTemplate 的变化
PERSISTENTVOLUMES	WATCH	GET	/api/v1/watch/persistentvolumes	监听所有 PersistentVolume 的变化
		GET	/api/v1/watch/persistentvolumes/{name}	监听某个 PersistentVolume 的变化
PERSISTENTVOLUMECLAIMS	WATCH	GET	/api/v1/watch/persistentvolumeclaims	监听所有 PersistentVolumeClaim 的变化
		GET	/api/v1/watch/namespaces/{namespace}/persistentvolumeclaims	监听某个 Namespace 下所有 PersistentVolumeClaim 的变化
		GET	/api/v1/watch/namespaces/{namespace}/persistentvolumeclaims/{name}	监听某个 PersistentVolumeClaim 的变化

下面基于 Fabric8 实现对资源对象的监听（Watch），代码如下：

```
private void testWatcher() {
    _kube.pods().watch(new io.fabric8.kubernetes.client.Watcher<Pod>() {
        @Override
        public void eventReceived(Action action, Pod pod) {
            System.out.println(action + ": " + pod);
        }
    })
}
```



```

        @Override
        public void onClose(KubernetesClientException e) {
            System.out.println("Closed: " + e);
        }
    });
}

```

接下来基于 Jersey 框架实现通过 Proxy 方式访问 Pod。由于 API Server 针对 Pod 资源提供了两种 Proxy 访问接口，所以下面分别用两段代码进行示例说明。代码如下：

```

private void testPoxyPod() {
    //访问第1种 proxy 接口
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setSubPath("/proxy");
    params.setResourceType(ResourceType.PODS);

    _restfulClient.get(params);

    //访问第2种 proxy 接口
    params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setVisitProxy(true);
    params.setResourceType(ResourceType.PODS);

    _restfulClient.get(params);
}

```